PATTERNS AND DATA

Vo Hoang Nhat Khang

HO CHI MINH CITY, FEBRUARY 2024

Contents

Chapter 1

INTRODUCTION ABOUT DESIGN PATTERNS

1.1 Design patterns - in general

Design patterns are a set of solutions and best practices that have emerged over time to address common software design problems. They provide a way to standardize the design of software applications and promote software quality, maintainability, and scalability. Design patterns are reusable solutions that can be applied to different software development scenarios and problems, and they can help developers avoid common pitfalls and reduce the overall complexity of their code.

Design patterns can be applied to all stages of the software development process, from requirements gathering and analysis to design, coding, and testing. They can also be used in a variety of software development paradigms, such as object-oriented programming, functional programming, and event-driven programming. By using design patterns, developers can create more robust, flexible, and maintainable software systems that are easier to understand and modify over time.

Some of the most widely used design patterns include the Singleton, Factory Method, Observer, Strategy, and Decorator patterns. Each of these patterns addresses a specific software design problem and provides a standard solution that can be adapted to different situations. While design patterns are not a silver bullet solution to all software design problems, they are a powerful tool that can help developers write better code and build better software systems.

1.2 Benefits of having design patterns

In general:

1. Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

2. Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, "Oh, just use a Singleton for that," and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

To be more specific, design patterns bring human lots of advantages:

1. **Reusability**: Design patterns provide reusable solutions to common software design problems. Developers can reuse these patterns in different contexts, making it easier to build and maintain software systems.

2. **Standardization**: Design patterns promote standardization of software design and coding practices, which can lead to better communication and collaboration among team members.
3. **Scalability**: Design patterns can help improve the scalability of software systems by providing a standardized approach to handling common design problems.
4. Maintainability: By promoting well-organized, modular code, design patterns can help make software systems easier to maintain and modify over time.
5. **Abstraction**: Design patterns can help developers abstract away low-level implementation details, allowing them to focus on higher-level design and architecture concerns.
6. **Code quality**: By promoting best practices and proven solutions, design patterns can help improve the overall quality of software code and reduce the risk of errors and bugs.
7. **Efficient development**: By providing pre-defined solutions to common design problems, design patterns can help reduce the time and effort required to develop software systems.

Chapter 2

DEFINITION AND CLASSIFICATION

2.1 Definition

A technical definition of design patterns can be as follows:

Design pattern

Design patterns are general, reusable solutions to commonly occurring problems in software design that embody best practices, provide proven solutions to specific design problems, and facilitate building software systems that are more maintainable, extensible, and scalable.

Design patterns typically describe the relationships between classes, objects, and interfaces, and provide a standard vocabulary for communicating and documenting software design decisions. They are not complete solutions to design problems, but rather, templates that can be adapted to specific contexts, and their effectiveness depends on the skill and experience of the software developers who use them.

What does the pattern consist of?

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

2.2 Classification

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed. I like the analogy to road construction: you can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.

The most basic and low-level patterns are often called **idioms**. They usually apply only to a single programming language.

The most universal and high-level patterns are **architectural patterns**. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

In addition, all patterns can be categorized by their intent, or purpose. Chapter 3 will describe in detail all the necessary knowledge for these patterns. For now, this report covers three main groups of patterns:

1. **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
2. **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
3. **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

There is no strict relationship between these three types of design patterns, as they address different aspects of software design. However, they can be used together to create more complex and robust software systems.

Chapter 3

DIFFERENT TYPES OF DESIGN PATTERNS

3.1 Creational patterns

Creational Patterns provide ways to create objects while hiding the creation logic, instead of instantiating objects directly using the new operator. This gives the program more flexibility in deciding which objects need to be created for a given use case.[1]

3.1.1 Factory Method

1 Definition

> Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

2 Problem

Imagine that you're creating a logistics management application.

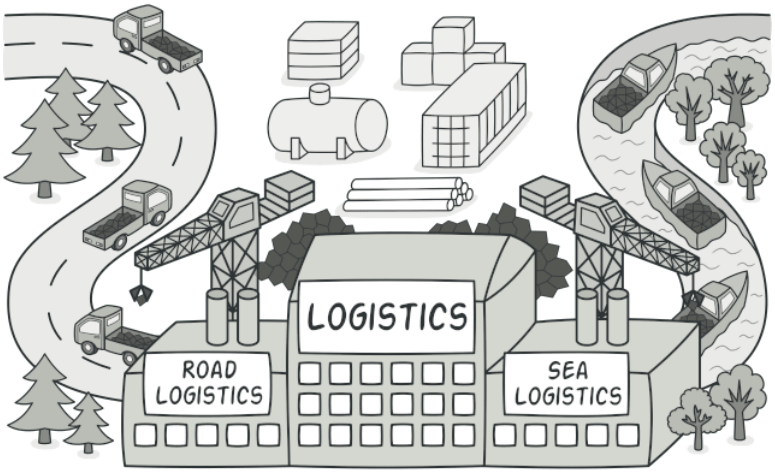


Figure 3.1: Logistics progress

Initially, your application is designed solely to manage truck transportation, and thus the majority of your code is contained within the `Truck` class. As your application gains popularity, you start receiving numerous requests from maritime transport companies to integrate sea logistics into the application.

[1] https://www.gofpatterns.com/design-patterns/module2/behavioral-creational-structural.php

Figure 3.2: Problem

Currently, a significant portion of your code is tightly connected to the `Truck` class. Integrating `Ships` into the application would necessitate modifying the entire codebase. Additionally, if you later want to incorporate another transportation type, you will likely have to redo all of these modifications. Consequently, your code will become quite unsightly, with conditional statements that alter the application's behavior based on the transportation object's class.

3 Solution

The design pattern known as the **Factory Method** proposes replacing direct calls to construct objects (using the `new` operator) with calls to a unique factory method. Though the objects are still created using the `new` operator, it is invoked within the factory method. Products are the objects returned by a factory method.

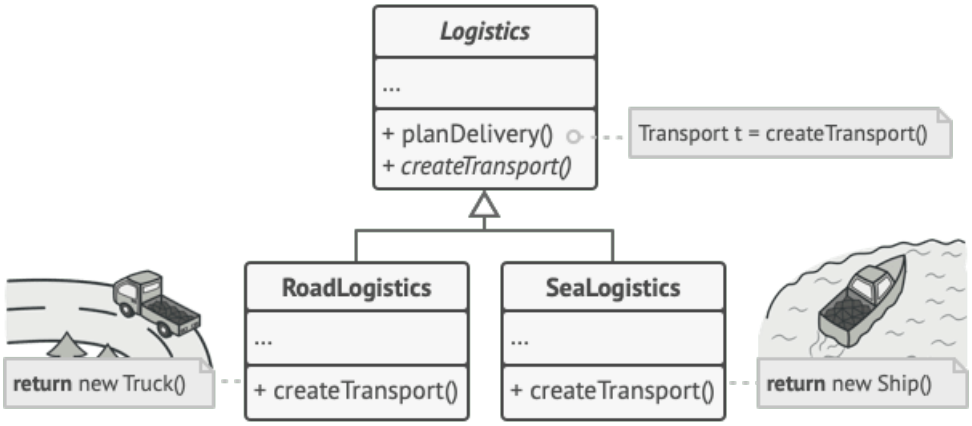


Figure 3.3: Subclasses can alter the class of objects being returned by the factory method

Initially, this modification may seem insignificant since all we did was relocate the constructor call from one section of the program to another. However, the change's significance lies in the ability to override the factory method in a subclass, enabling the product class created by the method to be altered.

Nevertheless, there is a minor constraint: Subclasses may only return diverse product types if they share a common base class or interface. Additionally, the base class's factory method should have its return type declared as the shared interface.

For example, both `Truck` and `Ship` classes should implement the `Transport` interface, which declares a method called `deliver`. Each class implements this method differently: Trucks deliver cargo by land, ships deliver cargo by sea.

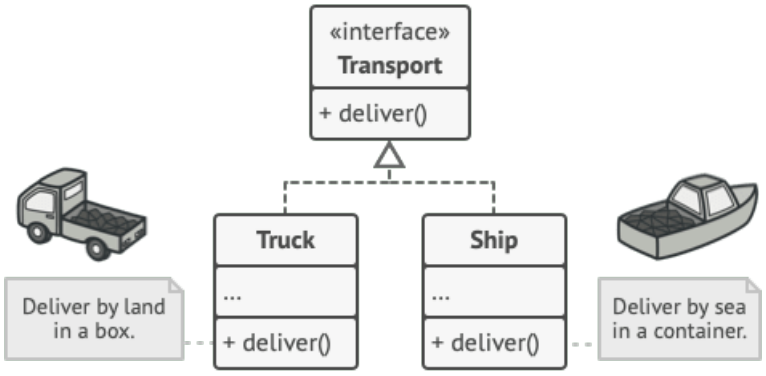


Figure 3.4: All products must follow the same interface

The factory method in the `RoadLogistics` class returns truck objects, whereas the factory method in the `SeaLogistics` class returns ships.

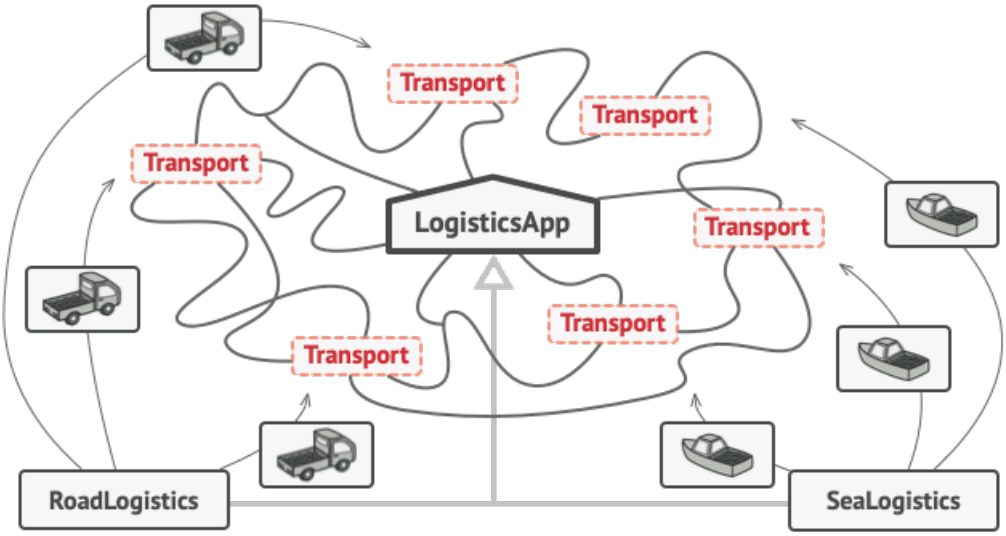


Figure 3.5: As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

The client code, which employs the factory method, views the products returned by different subclasses as identical. The client interacts with all products as an abstract `Transport` entity, and does not distinguish between them. The client understands that all transportation objects must have the `deliver` method, but it does not concern itself with the specifics of how the method operates.

4 Structure

There are 4 main components:

1. The **Product** defines the interface that is shared by all objects that can be created by the creator and its subclasses.
2. **Concrete Products** are distinct implementations of the product interface.
3. The **Creator** class plays a crucial role in the Factory Method pattern, as it declares the factory method that produces new product objects. The return type of this method must align with the product interface. You can enforce the creation of individual versions of the factory method in all subclasses by declaring the method as abstract, or you can have the base factory method

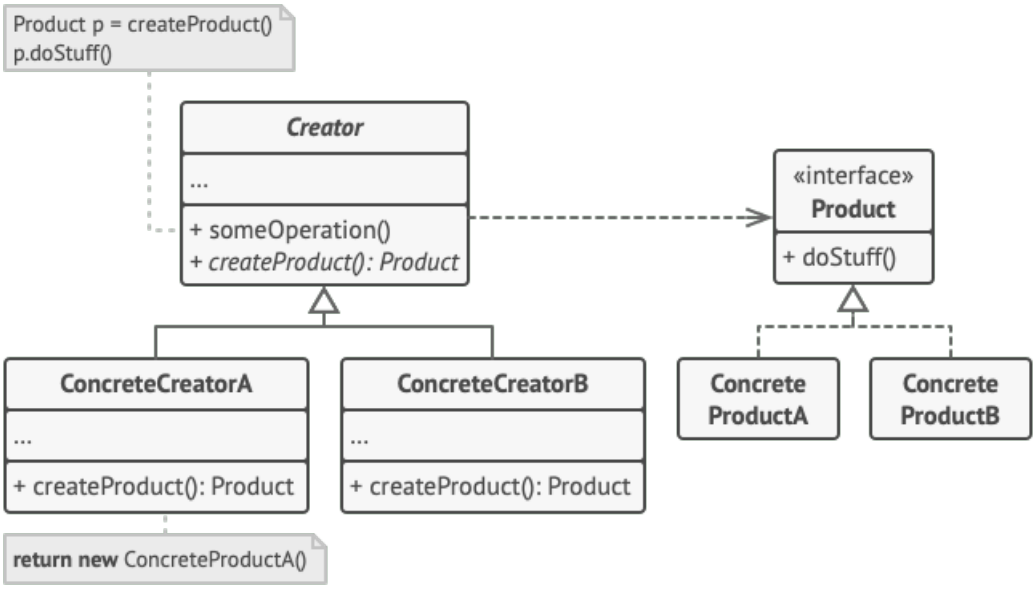


Figure 3.6: Factory Method structure.

return a default product type. But, the Creator class's primary purpose is not generating products. Instead, it usually includes essential business logic related to products. The factory method serves to decouple this logic from the concrete product classes. To illustrate this idea, imagine a large software development company that also has a training department for programmers. However, the main function of the company is still writing code, not training programmers.

4. **Concrete Creators** inherit from the Creator class and provide their own implementation of the factory method that generates a different type of product. It's important to note that the factory method does not have to create new instances every time it's called. Instead, it can also return pre-existing objects from various sources such as a cache, an object pool, or any other source.

5 Implementation

To implement the Factory Method pattern:

1. The first step is to ensure that all products have the **same interface**. This interface should have methods that make sense in every product.

2. Next, add an empty factory method to the **Creator** class. The return type of this method should match the common product interface.

3. In the **Creator**'s code, find all references to product constructors and replace them one by one with calls to the factory method. During this process, extract the product creation code into the factory method. A temporary parameter may be needed to control the type of returned product. The code inside the factory method may look unorganized at this point, with a large `switch` statement that selects the product class to instantiate. However, this will be improved later on.

4. Then, create a set of subclasses for each type of product listed in the factory method. Override the factory method in these subclasses and extract the appropriate construction code from the base method.

5. If there are too many product types to create a subclass for each one, you can reuse the control parameter from the base class in subclasses. For example, if the base `Mail` class has `AirMail` and `GroundMail` subclasses, and the `Transport` classes are `Plane`, `Truck`, and `Train`, the

`GroundMail` subclass can handle both `Truck` and `Train` objects by accepting an argument from the client code that controls which product it wants to receive.

6. If, after all of the extractions, the base factory method is empty, it can be made abstract. Otherwise, it can be made a default behavior of the method.

6 Advantages and disadvantages

- **Advantages:** The Factory Method pattern provides several benefits, including decoupling the creator from concrete products, adhering to the Single Responsibility Principle by consolidating product creation code in one place, and adhering to the Open/Closed Principle by allowing for the introduction of new product types without affecting existing client code. By avoiding tight coupling between the creator and the concrete products, the pattern also promotes flexibility, maintainability, and scalability of the codebase.
- **Disadvantages:** A potential downside of using the Factory Method pattern is that it may result in a larger codebase due to the introduction of many new subclasses. This can make the code more complex and difficult to maintain, especially if you're not using the pattern in an existing hierarchy of creator classes. Additionally, the pattern can add overhead to the application, as there is an extra layer of abstraction between the client and the product creation process. Finally, if not used appropriately, the pattern can lead to over-engineering, where the complexity of the solution outweighs the problem being solved.

7 Example

Suppose we have a program that needs to create different types of pizza objects, such as Cheese Pizza, Pepperoni Pizza, and Veggie Pizza. We can use the factory method pattern to create a `PizzaFactory` class that encapsulates the pizza creation logic.

First, we define an abstract `Pizza` class with a virtual method for preparing the pizza:

```
class Pizza {
public:
    virtual void prepare() = 0;
};
```

Next, we define concrete subclasses of the `Pizza` class for each type of pizza:

```
class CheesePizza : public Pizza {
public:
    void prepare() override {
        std::cout << "Preparing Cheese Pizza...\n";
    }
};

class PepperoniPizza : public Pizza {
public:
    void prepare() override {
        std::cout << "Preparing Pepperoni Pizza...\n";
    }
};

```

```
class VeggiePizza : public Pizza {
public:
    void prepare() override {
        std::cout << "Preparing Veggie Pizza...\n";
    }
};
```

Then we define a `PizzaFactory` class that has a factory method for creating pizza objects based on a given pizza type:

```
class PizzaFactory {
public:
    std::unique_ptr<Pizza> createPizza(const std::string& type) {
        if (type == "cheese") {
            return std::make_unique<CheesePizza>();
        } else if (type == "pepperoni") {
            return std::make_unique<PepperoniPizza>();
        } else if (type == "veggie") {
            return std::make_unique<VeggiePizza>();
        } else {
            throw std::runtime_error("Invalid pizza type.");
        }
    }
};
```

Finally, we can use the PizzaFactory to create different types of pizza objects without having to know the details of how each pizza is created:

```
int main() {
    PizzaFactory factory;
    std::unique_ptr<Pizza> cheesePizza = factory.createPizza("cheese");
    std::unique_ptr<Pizza> pepperoniPizza = ↩
       factory.createPizza("pepperoni");
    std::unique_ptr<Pizza> veggiePizza = factory.createPizza("veggie");

    cheesePizza->prepare();
    pepperoniPizza->prepare();
    veggiePizza->prepare();
}
```

3.1.2 Abstract Factory

1 Definition

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

2 Problem

Suppose you're developing a simulator for a furniture store.

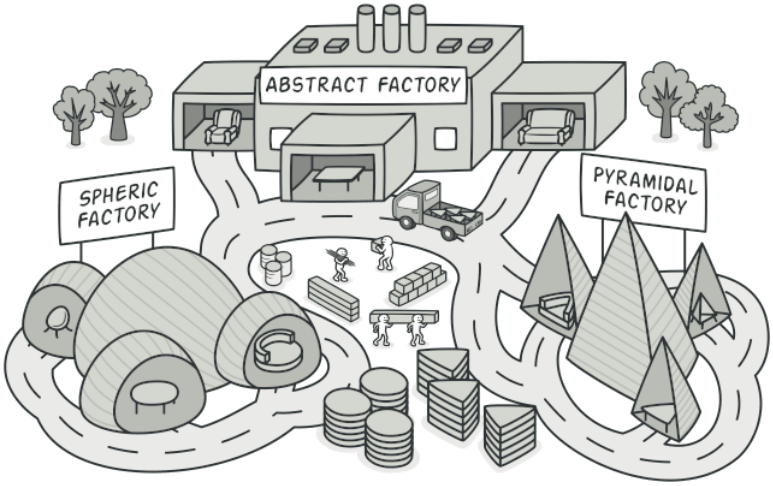


Figure 3.7: Problem

Your programming code comprises of classes that embody:

1. A group of interrelated furniture products, such as `Chair` + `Sofa` + `CoffeeTable`.
2. Multiple versions of this furniture group. For instance, the `Chair` + `Sofa` + `CoffeeTable` products are obtainable in styles like `Modern`, `Victorian`, and `ArtDeco`.

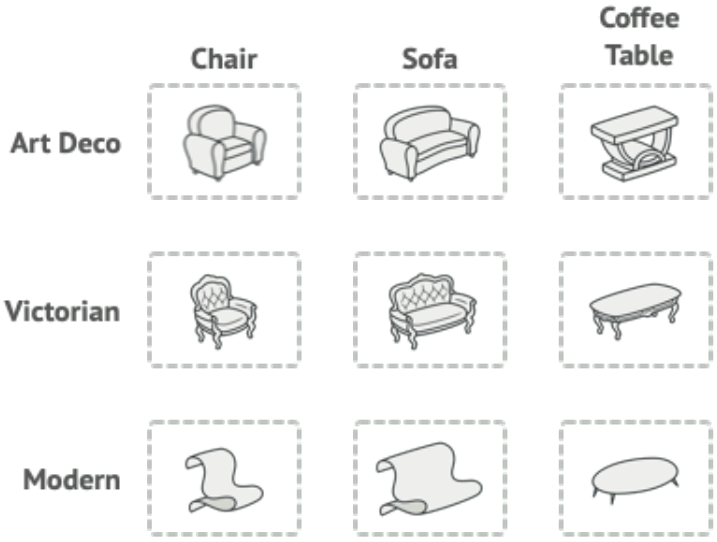


Figure 3.8: Product families and their variants.

To ensure that the furniture objects match others within the same family, you require a method to create individual furniture objects. It is vital to avoid non-matching furniture since customers can become very upset if they receive such products.

3 Solution

The Abstract Factory pattern proposes defining explicit interfaces for each unique product in the product family, such as chair, sofa, or coffee table. By doing so, all variations of the products can adhere to these interfaces. For instance, all chair variants can implement the `Chair` interface, while all coffee table variations can implement the `CoffeeTable` interface, and so forth.

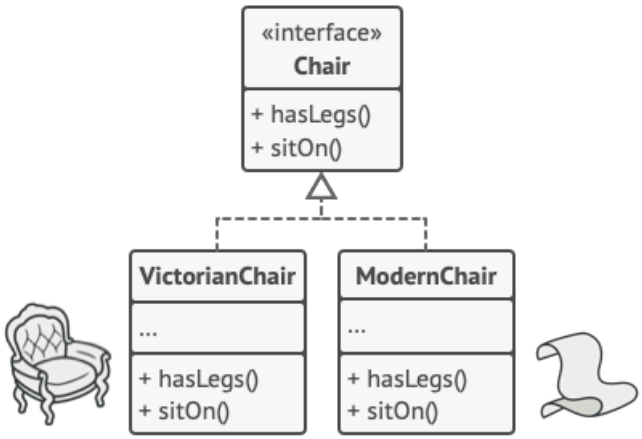


Figure 3.9: All variants of the same object must be moved to a single class hierarchy.

The next step involves defining the **Abstract Factory**. This is an interface that includes a set of methods for creating all the products in a product family, such as `createChair`, `createSofa`, and `createCoffeeTable`. These methods should return abstract product types, which are represented by the interfaces we previously extracted, such as `Chair`, `Sofa`, `CoffeeTable`, etc.

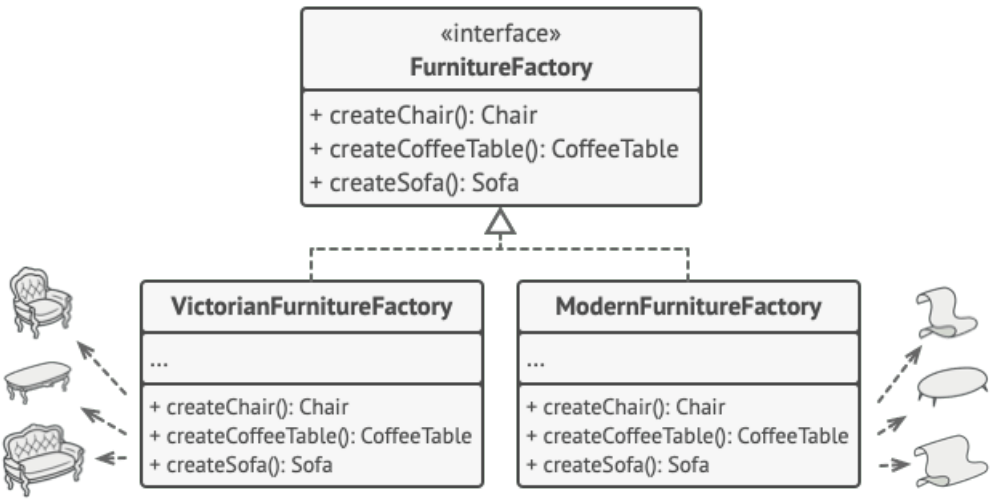


Figure 3.10: Each concrete factory corresponds to a specific product variant.

Now, let's talk about the product variants. For each variant within a product family, we create a separate factory class based on the **AbstractFactory** interface. A factory is a class that returns products of a specific kind. For instance, the `ModernFurnitureFactory` can only create `ModernChair`, `ModernSofa`, and `ModernCoffeeTable` objects.

If a client wants a chair from a factory, they don't need to know the factory's class or the specific type of chair they'll receive. Whether it's a `Modern` or `Victorian-style` chair, the client interacts with all chairs using the abstract `Chair` interface. This way, the client only knows that the chair implements the `sitOn` method. Also, any chair variant returned will match the sofa or coffee table from the same factory.

One more thing: If the client only uses abstract interfaces, what creates the actual factory objects? Usually, during initialization, the application creates a concrete factory object. Before that, it must choose the factory type based on its configuration or environment.

4 Structure

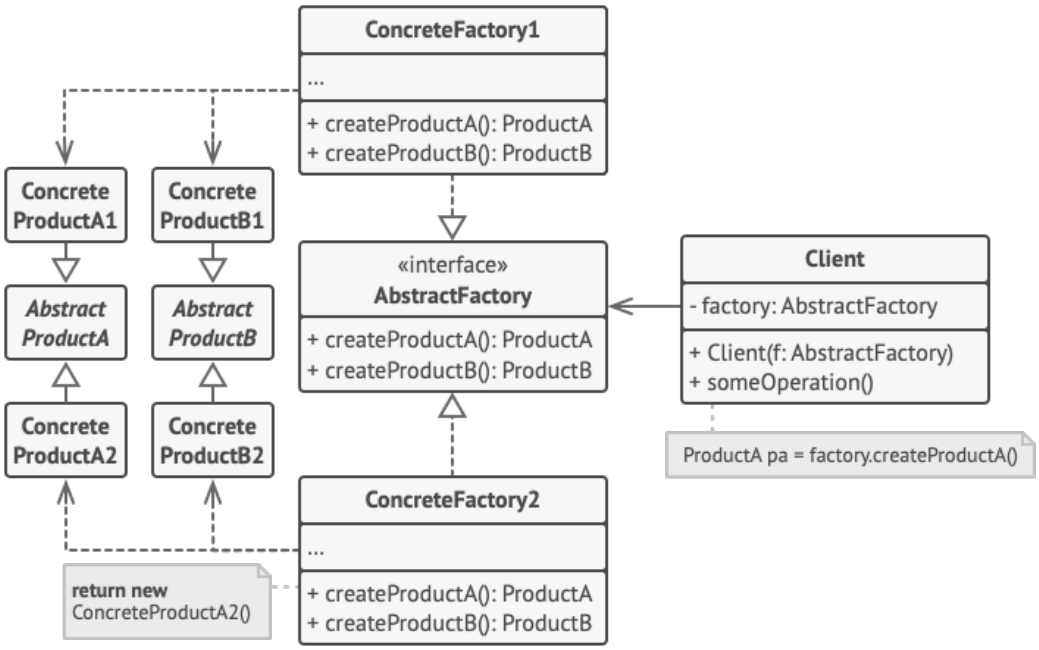


Figure 3.11: Structure of Abstract Factory method.

1. **Abstract Products** define interfaces for a group of different but related products that form a product family.
2. **Concrete Products** are different implementations of abstract products, organized by variant. Each abstract product (e.g., chair/sofa) must have an implementation for every variant (e.g., Victorian/-Modern).
3. The **Abstract Factory** interface defines a set of methods for creating each abstract product.
4. **Concrete Factories** implement the creation methods defined by the abstract factory. Each concrete factory corresponds to a specific product variant and only creates that variant of products.
5. Even though concrete factories create concrete products, their creation methods must return the corresponding abstract products. This way, the client code using a factory isn't tied to the specific product variant it receives from the factory. The client can work with any concrete factory/product variant as long as it interacts with their objects through abstract interfaces.

5 Implementation

In a cross-platform application, UI elements are expected to behave similarly but look slightly different on different operating systems. It's your responsibility to ensure that the UI elements match the style of the current OS. You wouldn't want your program to display macOS controls on Windows.

The **Abstract Factory** interface defines creation methods that the client code can use to produce different types of UI elements. Concrete factories correspond to specific operating systems and create UI elements that match that OS.

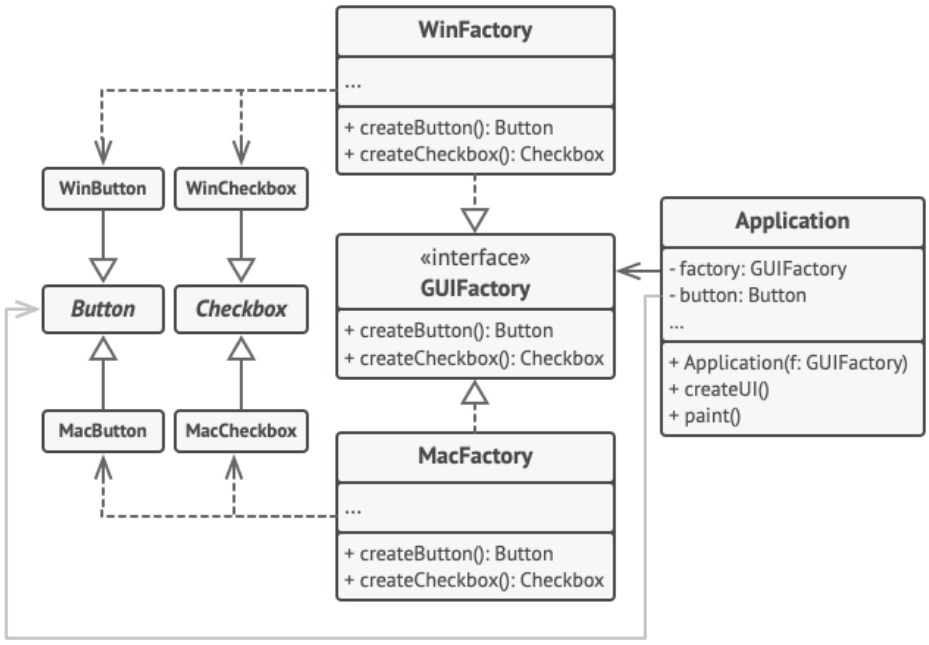


Figure 3.12: The cross-platform UI classes example.

Here's how it works: When an application starts, it checks the current operating system. The app uses this information to create a factory object from a class that matches the OS. The rest of the code uses this factory to create UI elements, preventing the creation of incorrect elements.

With this approach, the client code isn't dependent on concrete factory and UI element classes as long as it interacts with them through their abstract interfaces. This also allows the client code to support other factories or UI elements added in the future.

As a result, you don't need to modify the client code each time you add a new variation of UI elements to your app. You just need to create a new factory class that produces these elements and slightly adjust the app's initialization code to select that class when appropriate.

6 Advantages and disadvantages

- **Advantages:**
  - With a factory, you can be confident that the products you receive are compatible with each other.
  - You also avoid tight coupling between concrete products and client code.
  - The Single Responsibility Principle is followed by extracting product creation code into one place, making it easier to maintain.
  - The Open/Closed Principle is also followed by allowing new product variants to be introduced without breaking existing client code.
- **Disadvantages:** The code can become more complex due to the introduction of many new interfaces and classes with the pattern.

7 Example

Suppose we have a car dealership that sells different types of cars. We want to implement a system that can create different car models with different features, such as engine type, transmission, and color. We can use the Abstract Factory Method pattern to create different factories that can create different types of cars with different features.

First, we define an abstract car class with virtual methods for each feature:

```
class Car {
public:
    virtual void setEngineType() = 0;
    virtual void setTransmission() = 0;
    virtual void setColor() = 0;
};
```

Next, we create concrete classes for different car models, each of which inherit from the abstract car class:

```
class SedanCar : public Car {
public:
    void setEngineType() override {
        // Set engine type for sedan car
    }

    void setTransmission() override {
        // Set transmission for sedan car
    }

    void setColor() override {
        // Set color for sedan car
    }
};

class SportsCar : public Car {
public:
    void setEngineType() override {
        // Set engine type for sports car
    }

    void setTransmission() override {
        // Set transmission for sports car
    }

    void setColor() override {
        // Set color for sports car
    }
};
```

Then, we create an abstract factory class with virtual methods for each type of car:

```
class CarFactory {
public:
```

```
    virtual Car* createSedanCar() = 0;
    virtual Car* createSportsCar() = 0;
};
```

Next, we create concrete factory classes for each type of car, each of which inherit from the abstract factory class:

```
class LuxuryCarFactory : public CarFactory {
public:
    Car* createSedanCar() override {
        return new SedanCar(); // Create a luxury sedan car
    }

    Car* createSportsCar() override {
        return new SportsCar(); // Create a luxury sports car
    }
};

class EconomyCarFactory : public CarFactory {
public:
    Car* createSedanCar() override {
        return new SedanCar(); // Create an economy sedan car
    }

    Car* createSportsCar() override {
        return new SportsCar(); // Create an economy sports car
    }
};
```

Finally, we can use the factories to create different types of cars with different features:

```
int main() {
    // Create a luxury sedan car
    CarFactory* luxuryFactory = new LuxuryCarFactory();
    Car* luxurySedan = luxuryFactory->createSedanCar();

    // Create an economy sports car
    CarFactory* economyFactory = new EconomyCarFactory();
    Car* economySports = economyFactory->createSportsCar();

    return 0;
}
```

3.1.3 Builder

1 Definition

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

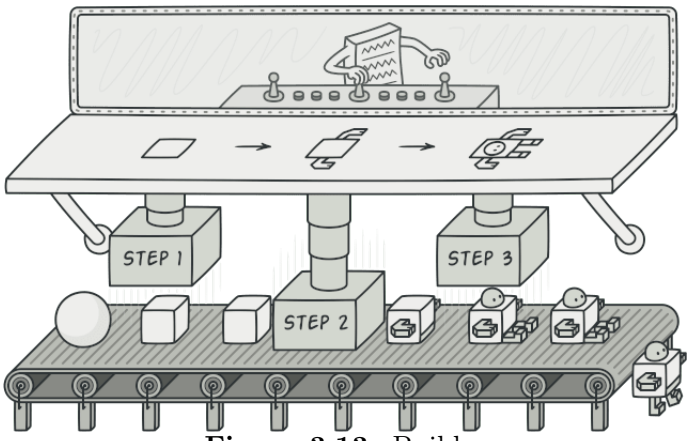


Figure 3.13: Builder

2 Problem

Imagine an intricate entity that necessitates a meticulous, sequential setup of numerous attributes and embedded entities. Typically, this setup code is encapsulated within a massive constructor with numerous parameters or, worse still, dispersed throughout the client code.

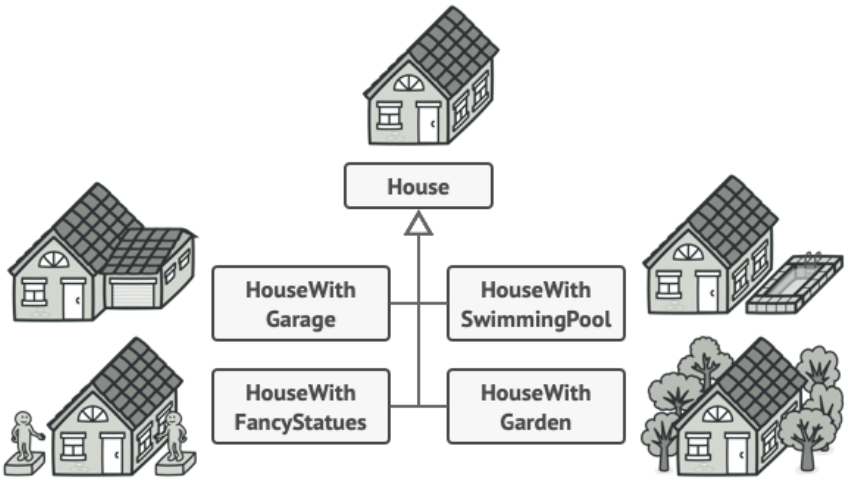


Figure 3.14: You might make the program too complex by creating a subclass for every possible configuration of an object.

Consider the creation of a `House` object, where a basic house requires the construction of four walls and a floor, the installation of a door, the fitting of a pair of windows, and the building of a roof. However, for a more complex house with additional features, such as a heating system, plumbing, electrical wiring, and a backyard, simply extending the base `House` class and creating subclasses to cover all combinations of parameters can become cumbersome, resulting in a considerable number of subclasses. Moreover, any new parameter, such as the porch style, would require an expansion of this hierarchy.

Alternatively, a more efficient approach is to avoid creating subclasses and instead incorporate a comprehensive constructor within the base `House` class, containing all potential parameters to control

the `House` object. However, while this approach removes the need for subclasses, it presents another issue.

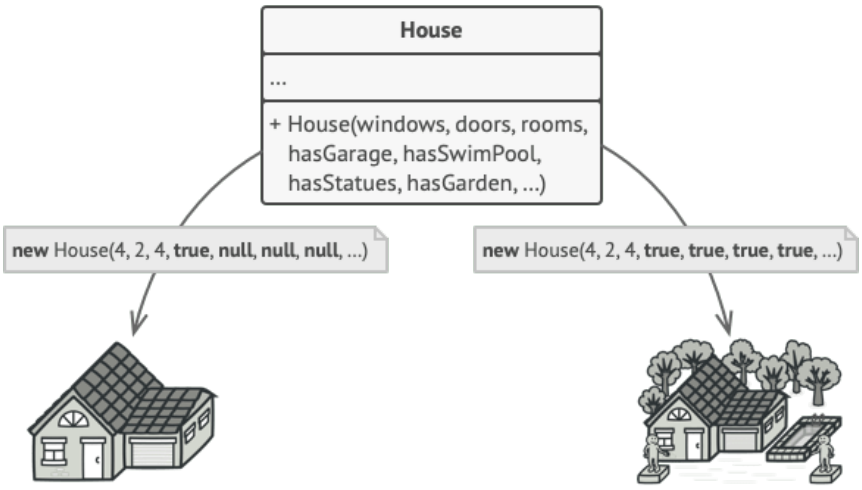


Figure 3.15: The constructor with lots of parameters has its downside: Not all the parameters are needed at all times.

3 Solution

To avoid these issues, the Builder pattern proposes the extraction of the object construction code from its own class and relocation to separate objects known as **builders**.

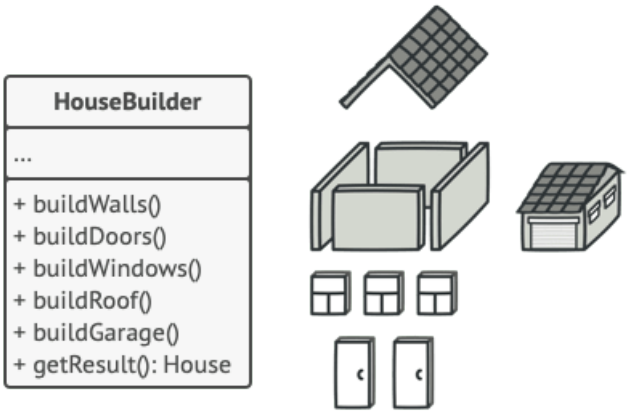


Figure 3.16: The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

The Builder pattern arranges object construction into a sequence of steps (e.g., `buildWalls`, `buildDoor`, etc.). To construct an object, you execute a sequence of these steps on a builder object. The essential aspect is that not all of the steps need to be called, only those that are necessary for producing a specific configuration of an object.

Furthermore, some of the construction steps may require different implementations to build various representations of the product. For instance, the walls of a cabin may be constructed of wood, while the walls of a castle must be built with stone. In such cases, several different builder classes can be created to implement the same set of building steps but in a distinct manner. These builders can then be used in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

Director

You have the option to take it a step further and create a separate class named `director`, which will contain a sequence of calls to the builder steps used to construct a product. The `director` class defines the order in which the building steps will be executed, while the `builder` class provides the implementation for each of those steps.

While having a `director` class in a program is not mandatory, it is still beneficial to have one as it allows for the reuse of construction routines across the program. The client code can call the building steps directly in a particular order, but using a `director` class is a better approach.

Moreover, the `director` class abstracts the product construction details from the client code, simplifying the client's responsibilities to only associate a `builder` with a `director`, execute the construction through the `director`, and retrieve the outcome from the `builder`.

4 Structure

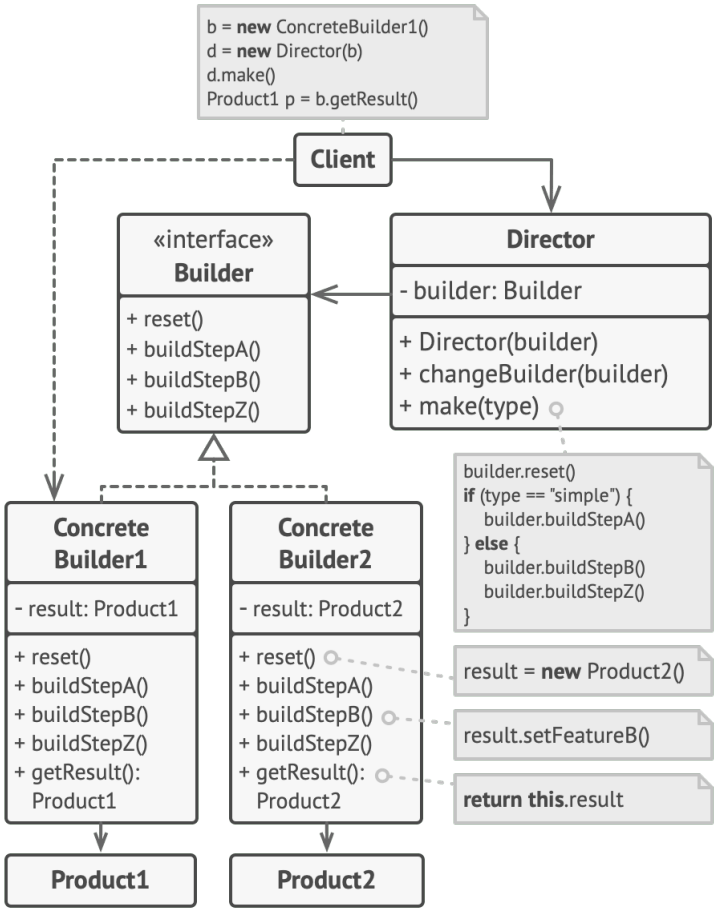


Figure 3.17: Structure of builder method

1. The `Builder` interface declares product construction steps that are common to all types of builders.
2. Concrete builders provide different implementations of the construction steps. They may produce products that do not follow the common interface declared by the `Builder` interface.
3. Products are resulting objects that can be constructed by different builders. They do not necessarily have to belong to the same class hierarchy or interface.

4. The `Director` class defines the order in which to call construction steps, allowing you to create and reuse specific configurations of products.

5. The client must associate one of the `Builder` objects with the `Director`. This is typically done once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach where the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

5 Implementation

A **car** is a intricate object that can be created in numerous ways. Rather than overloading the `Car` class with a lengthy constructor, we separated the car assembly code into a distinct `CarBuilder` class. This class comprises of a collection of methods for setting up different components of a car.

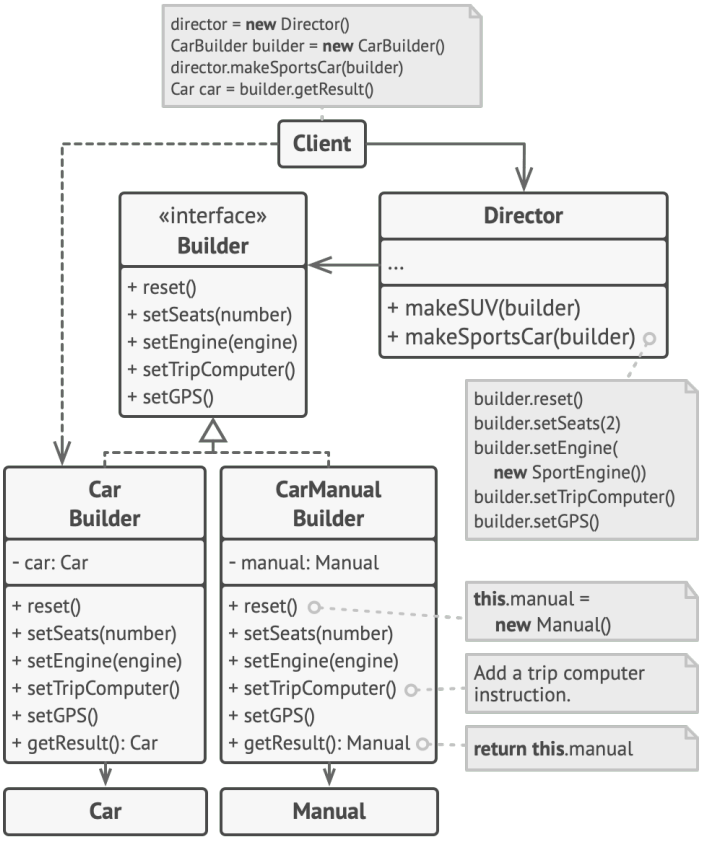


Figure 3.18: The example of step-by-step construction of cars and the user guides that fit those car models.

Constructing a car can be a complex task that can have numerous variations. Instead of creating a massive constructor in the `Car` class, we separated the code responsible for building the car into a separate class called `CarBuilder`. This new class contains a collection of methods that can configure different parts of the car.

To create a customized car, the client code can use the builder directly, whereas for creating popular car models, the client can delegate the assembly process to the `Director` class, which knows how to use the builder.

Each car comes with a manual that describes its features. As the details in the manuals can vary across different car models, it makes sense to reuse the existing construction process for both the cars and their manuals. Therefore, we need another builder class that specializes in creating manuals, implementing the same building methods as the `CarBuilder` class.

To obtain the resulting object, we can't have a method for fetching the result in the director without coupling the director to specific product classes. Hence, the `Builder` class that performed the construction job is responsible for providing the result of the construction, whether it is a metal car or a paper manual.

6 Advantages and disadvantages

These are some benefits of using the Builder pattern in object-oriented programming:

- The ability to construct objects in a step-by-step manner, with the flexibility to defer certain construction steps or run them recursively.
- The ability to reuse the same construction code when building different representations of products.
- The application of the Single Responsibility Principle, which allows for the isolation of complex construction code from the business logic of the product. This separation makes the code easier to maintain and modify over time.

One potential drawback of using the Builder pattern is that it may increase the overall complexity of the codebase. This is because the pattern often requires the creation of multiple new classes, such as the Builder and Director classes, which can make the codebase larger and potentially more difficult to understand.

7 Example

Suppose we are developing a game that allows players to create their own characters. Each character has a variety of attributes, such as name, gender, race, class, and equipment. However, creating a character with all these attributes can be a complex task, especially if we want to allow for different combinations of attributes. Additionally, we may want to reuse some of the construction code for other parts of the game.

We can use the Builder pattern to solve this problem. The idea is to create a separate builder class for constructing characters, which can handle the complexity of creating characters and allow for different combinations of attributes.

First, we need to define the `Product` class that represents the character being constructed. This class should have attributes for each of the character's attributes, such as `name`, `gender`, `race`, `class`, and `equipment`.

```
class Character {
public:
    void setName(std::string name) { mName = name; }
    void setGender(std::string gender) { mGender = gender; }
    void setRace(std::string race) { mRace = race; }
    void setClass(std::string characterClass) { mClass = characterClass; }
    void setEquipment(std::string equipment) { mEquipment = equipment; }
private:
    std::string mName;
    std::string mGender;
    std::string mRace;
    std::string mClass;
    std::string mEquipment;
};
```

Define the abstract `Builder` class that provides methods for constructing the character. These methods should correspond to the attributes of the Product class.

```
class CharacterBuilder {
public:
    virtual ~CharacterBuilder() {}
    virtual void buildName() = 0;
    virtual void buildGender() = 0;
    virtual void buildRace() = 0;
    virtual void buildClass() = 0;
    virtual void buildEquipment() = 0;
    virtual Character* getCharacter() = 0;
};
```

Define one or more concrete `Builder` classes that implement the methods of the `Builder` class. Each concrete builder should provide its own implementation for constructing the character.

```
class HumanCharacterBuilder : public CharacterBuilder {
public:
    virtual void buildName() override { mCharacter->setName("Human"); }
    virtual void buildGender() override { mCharacter->setGender("Male"); }
    virtual void buildRace() override { mCharacter->setRace("Human"); }
    virtual void buildClass() override { mCharacter->setClass("Warrior"); }
    virtual void buildEquipment() override { ↩
       mCharacter->setEquipment("Sword and Shield"); }
    virtual Character* getCharacter() override { return mCharacter; }
private:
    Character* mCharacter = new Character();
};
```

Define the `Director` class that manages the construction process. The `Director` class should have a method for setting the Builder object, and a method for starting the construction process.

```
class CharacterDirector {
public:
    void setBuilder(CharacterBuilder* builder) { mBuilder = builder; }
    void constructCharacter() {
        mBuilder->buildName();
        mBuilder->buildGender();
        mBuilder->buildRace();
        mBuilder->buildClass();
        mBuilder->buildEquipment();
    }
private:
    CharacterBuilder* mBuilder;
};
```

Now everything is set. We can use the `Builder` pattern to create a character.

3.1.4 Prototype

1 Definition

The **Prototype** pattern is a creational design pattern that enables you to create copies of existing objects without the need to know their specific class types. This approach helps to avoid introducing dependencies in your codebase, since you can create new objects by copying existing ones instead of creating them from scratch. The pattern is particularly useful when creating complex objects that are expensive to instantiate or when you need to isolate the creation process from the rest of the codebase.

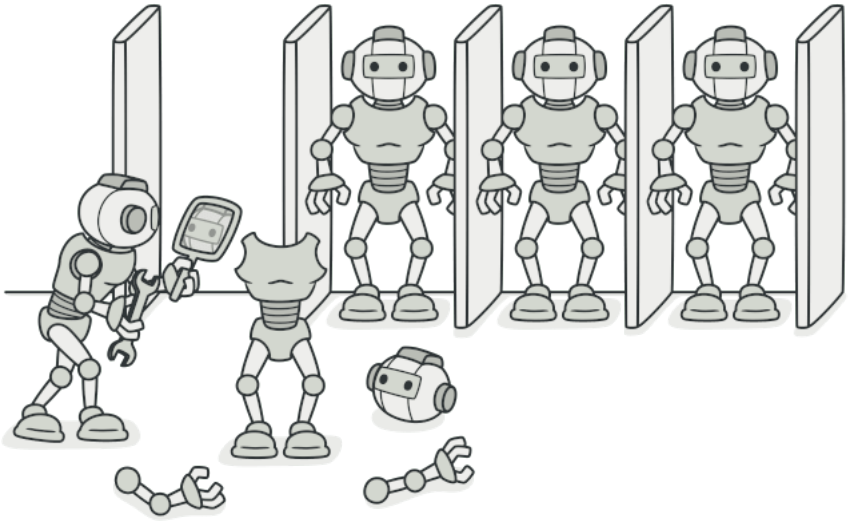


Figure 3.19: Prototype method

2 Problem

Suppose you have an object that you want to replicate exactly. How would you go about it? One approach is to create a new object of the same class and then manually copy the values of all the fields from the original object to the new object. However, this approach can be problematic if some of the object's fields are private and inaccessible from outside the object. In such cases, copying the object directly may not be possible, and an alternative approach is required.

In addition to the aforementioned issue, there is another problem with the direct approach of copying objects. This approach introduces a dependency on the class of the object you wish to copy, which can lead to coupling issues in your codebase. Moreover, in some cases, you may only have access to an object's interface and not its concrete class. For instance, a method may accept any object that adheres to a specific interface. In such scenarios, creating a copy of the object directly is not possible, and you need to resort to alternative techniques for replicating objects.

3 Solution

The Prototype pattern offers a solution to the problem of cloning objects by delegating the cloning process to the objects themselves. The pattern defines a common interface that all cloneable objects should implement, enabling the creation of a clone without coupling the code to the object's class. Typically, this interface contains a single method, clone.

The implementation of the clone method is similar across all classes. The method creates a new object of the current class and copies all of the field values from the original object to the new one. Even private fields can be copied, as most programming languages allow objects to access private fields of other objects belonging to the same class.

An object that supports cloning is referred to as a prototype. When working with objects that have numerous fields and countless possible configurations, cloning can serve as a viable alternative to subclassing.

4 Structure

First, let's investigate basic implementation of Prototype function:

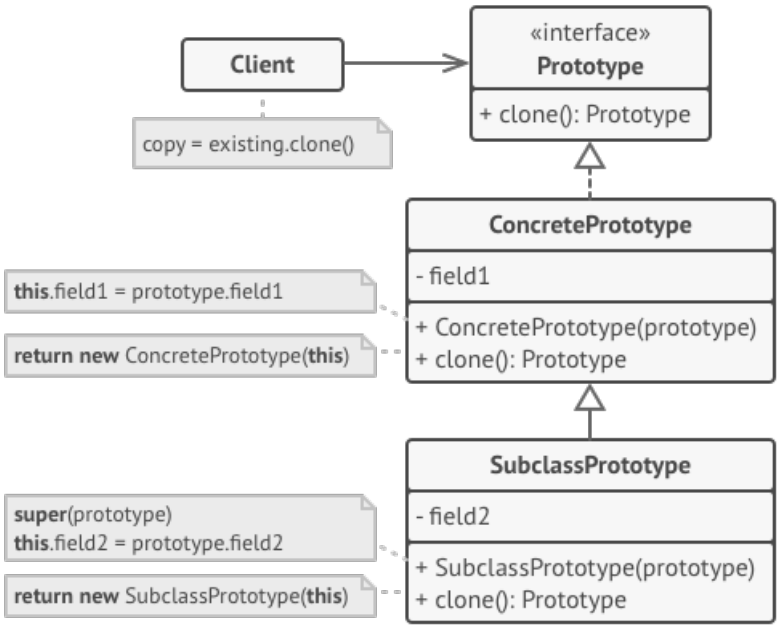


Figure 3.20: Basic implementation

The **Prototype** interface specifies the methods for cloning. Typically, this is just one method called `clone`.

The **Concrete Prototype** class carries out the cloning method. Besides replicating the original object's data to the clone, this method may also address specific edge cases of the cloning process, such as cloning linked objects and resolving recursive dependencies.

A **Client** can create a duplicate of any object that adheres to the prototype interface.

About **Prototype registry** implementation:

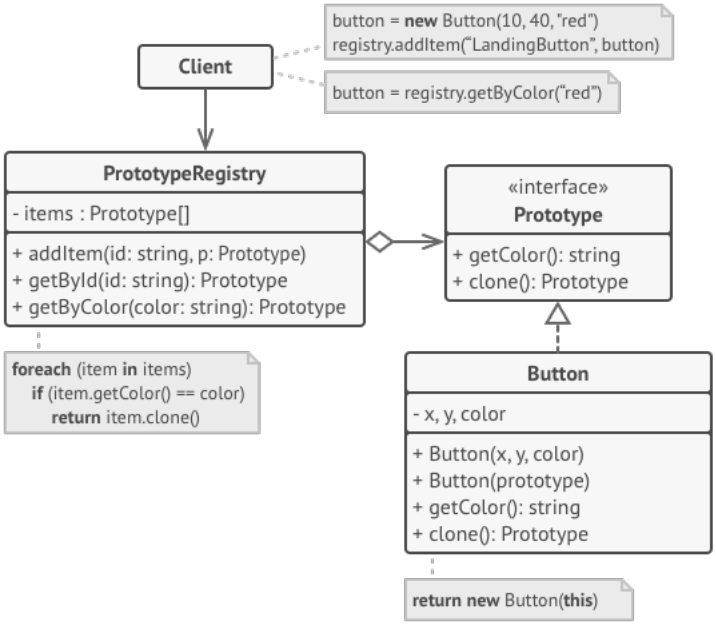


Figure 3.21: Prototype registry

The **Prototype Registry** offers a convenient way to access commonly used prototypes. It contains a set of pre-constructed objects that are ready to be duplicated. The most basic prototype registry is a `name → prototype` hash map. However, if you require more sophisticated search criteria than just a name, you can create a more robust version of the registry.

5 Implementation

1. Establish the prototype interface and include the `clone` method. Alternatively, simply add the method to all classes in an existing class hierarchy if you have one.
2. A prototype class must have an alternate constructor that takes an object of that class as an argument. This constructor must copy the values of all fields defined in the class from the given object into the new instance. If you are modifying a subclass, you must invoke the parent constructor to allow the superclass to handle the cloning of its private fields.
3. If your programming language does not support method overloading, you will not be able to create a separate "prototype" constructor. As a result, copying the object's data into the newly created clone will have to be done within the `clone` method. However, having this code in a regular constructor is safer because the resulting object is returned fully configured immediately after calling the `new` operator.
4. The `clone` method usually consists of just one line: invoking a `new` operator with the prototypical version of the constructor. Note that each class must explicitly override the `clone` method and use its own class name along with the `new` operator. Otherwise, the `clone` method may produce an object of a parent class. Optionally, create a centralized prototype registry to store a catalog of frequently used prototypes.
5. You can implement the registry as a new factory class or include it in the base prototype class with a static method for fetching prototypes. This method should search for a prototype based on search criteria passed to it by client code. The criteria could be either a simple string tag or a complex set of search parameters. After finding the appropriate prototype, the registry should clone it and return the copy to the client.
6. Finally, replace direct calls to subclass constructors with calls to the factory method of the prototype registry.

6 Advantages and disadvantages

- **Advantages:**
  - You have the ability to systematically build objects by following a sequence of steps. It is also possible to delay certain construction steps or execute them recursively.
  - It is feasible to employ the same construction code to build different representations of products, promoting reusability.
  - The Single Responsibility Principle can be applied, allowing you to separate intricate construction code from the core business logic of the product.
- **Disadvantage**:

  The overall complexity of the code rises as the pattern necessitates the creation of multiple new classes.

7 Example

Let's consider a scenario where we have a system that deals with generating various types of documents. Each document type has a specific structure and content. Creating a new document instance every time we need to generate a document can be time-consuming and inefficient. Moreover, if we need to modify the structure or content of a document type, it would require changes in multiple places, making the codebase more complex and error-prone.

Step 1: Define an abstract base class, `Document`, which represents the common interface for all document types.

```
class Document {
    public:
        virtual Document* clone() const = 0;
        virtual void print() const = 0;
        // ...
};
```

Step 2: Implement concrete classes derived from the `Document` base class, representing different document types.

```
class Resume : public Document {
    public:
        Document* clone() const override {
            return new Resume(*this);
        }

        void print() const override {
            // Implementation for printing a resume document
        }
        // ...
};

class Report : public Document {
    public:
        Document* clone() const override {
            return new Report(*this);
```

```
        }

        void print() const override {
            // Implementation for printing a report document
        }
    // ...
};
```

Step 3: Create a `DocumentPrototypeFactory` class that manages a pool of pre-configured document prototypes.

```
class DocumentPrototypeFactory {
    private:
        std::unordered_map<std::string, Document*> prototypes;

    public:
        DocumentPrototypeFactory() {
            // Initialize prototypes with pre-configured instances
            prototypes["resume"] = new Resume();
            prototypes["report"] = new Report();
        }

        Document* createDocument(const std::string& type) const {
            auto it = prototypes.find(type);
            if (it != prototypes.end())
                return it->second->clone();
            return nullptr;
        }
};
```

Step 4: Demonstrate the usage of the Prototype pattern by creating new document instances based on the prototypes.

```
int main() {
    DocumentPrototypeFactory factory;
    Document* resume = factory.createDocument("resume");
    if (resume) {
        resume->print();
        delete resume;
    }

    Document* report = factory.createDocument("report");
    if (report) {
        report->print();
        delete report;
    }

    return 0;
}
```

3.1.5 Singleton

1 Definition

Singleton is a creational design pattern that guarantees the existence of only one instance of a class, while also providing a global access point to this instance.

2 Problem

The **Singleton** pattern addresses two simultaneous concerns, potentially conflicting with the Single Responsibility Principle:

1. Ensuring a class has a sole instance: Controlling the number of instances of a class is often necessary to manage access to shared resources, such as databases or files. The Singleton pattern guarantees that subsequent object creation requests return the same instance rather than creating a new one. This behavior cannot be achieved with a regular constructor, which always generates a new object.

2. Providing a global access point to the instance: Previously, global variables were employed to store crucial objects. However, this approach posed risks as any code could overwrite these variables, leading to application crashes. The Singleton pattern offers a solution by granting access to an object from anywhere in the program while safeguarding it against external overwrites.

Moreover, consolidating the code addressing the first concern within a single class is favorable, as opposed to scattering it throughout the program. This approach enhances code organization, especially when other parts of the program depend on this shared functionality.

It's worth mentioning that the term *singleton* is sometimes used loosely to describe solutions that address only one of the aforementioned problems, showcasing the popularity and broad usage of the Singleton pattern in contemporary development.

3 Solution

Here are the steps commonly shared by all implementations of the Singleton pattern:

1. Make the default constructor private: This step ensures that other objects cannot use the new operator with the Singleton class, preventing direct instantiation.

2. Create a static creation method: This method serves as a substitute for a constructor. Internally, it invokes the private constructor to create a new object and stores it in a static field. Subsequent calls to this method return the cached object, ensuring that the same instance is always returned.

If your code has access to the Singleton class, it can utilize the Singleton's static method. Consequently, whenever this method is invoked, the same object is consistently returned.

4 Structure

The **Singleton** class defines the `getInstance` static method, which allows obtaining the single instance of the class.

To ensure encapsulation, the constructor of the Singleton class is concealed from client code. The `getInstance` method serves as the sole means of acquiring the Singleton object.

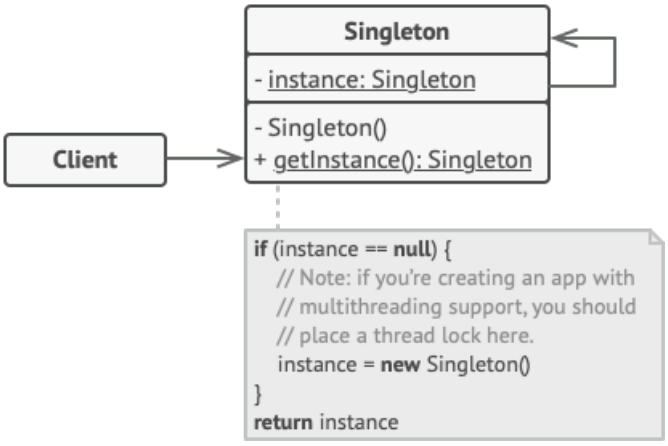


Figure 3.22: Structure of Singleton

5 Implementation

The **Singleton** class includes a private static field for storing the singleton instance. Additionally, a public static creation method is declared to obtain the singleton instance.

Inside the static creation method, a "lazy initialization" approach is implemented. On its first invocation, the method creates a new object and assigns it to the static field. Subsequent calls to the method will always return the same instance from the static field.

To ensure encapsulation, the constructor of the class is made private. As a result, only the static method within the class can access the constructor, while other objects are unable to do so.

When working with the client code, it is recommended to replace all direct calls to the singleton's constructor with calls to its static creation method.

6 Advantages and disadvantages

Advantages:

- **Single Instance**: With the Singleton pattern, you can ensure that a class has only a single instance. This eliminates the possibility of multiple instances being created, which can be useful in scenarios where having multiple instances could lead to conflicts or inconsistent behavior.
- **Global Access Point**: The Singleton pattern provides a global access point to the single instance it manages. This means that the instance can be easily accessed from anywhere in the codebase, facilitating a centralized and convenient way of working with the singleton object.
- **Lazy Initialization**: The singleton object is initialized only when it is requested for the first time. This "lazy initialization" approach helps optimize resource usage by deferring the creation of the object until it is actually needed. This can be particularly beneficial in scenarios where the creation of the object is resource-intensive or time-consuming.

Disadvantages:

1. **Violations of the Single Responsibility Principle:** The Singleton pattern can potentially violate the Single Responsibility Principle since it solves two problems simultaneously. It not only ensures the existence of a single instance but also provides a global access point to that instance. This dual responsibility can make the code more complex and less modular.
2. **Masking Bad Design:** In some cases, the Singleton pattern can be used to mask poor design choices. When components of a program have excessive knowledge about each other, the Singleton pattern may be employed to create a shared instance and facilitate communication. However, this can lead to tight coupling and hinder code maintainability and testability.

3. **Thread Safety Concerns:** The Singleton pattern requires special handling in a multithreaded environment to prevent multiple threads from creating multiple instances. Synchronization mechanisms or other thread-safe techniques need to be implemented to ensure the proper creation and use of the Singleton object in concurrent scenarios.

4. **Challenges in Unit Testing:** Unit testing the client code of the Singleton can be challenging. Many test frameworks rely on inheritance to produce mock objects, but the private constructor and the inability to override static methods in most languages make it difficult to mock the Singleton instance. As a result, alternative strategies or creative approaches may be required to effectively test code that depends on the Singleton pattern. Alternatively, considering different design patterns or design principles may be preferable to avoid such testing difficulties.

7 Example

In a multi-threaded application, you need to ensure that there is only one instance of a database connection object shared across the entire system.

Step 1: Define the Singleton class, let's call it `DatabaseConnection`, with a private constructor and a private static instance member variable.

```
class DatabaseConnection {
    private:
        static DatabaseConnection* instance;
        // Private constructor to prevent direct instantiation
        DatabaseConnection() {
            // Perform connection setup
        }

    public:
    // Public static method to access the singleton instance
        static DatabaseConnection* getInstance() {
            if (!instance) {
                // Perform lazy initialization on first access
                instance = new DatabaseConnection();
            }
            return instance;
        }
};
```

Step 2: Implement lazy initialization inside the `getInstance` method. The first time `getInstance` is called, it creates a new `DatabaseConnection` object and assigns it to the `instance` variable. Subsequent calls to `getInstance` will simply return the already instantiated object.

```
DatabaseConnection* DatabaseConnection::instance = nullptr;
```

Now, you can use the Singleton `DatabaseConnection` class in your code as follows:

```
int main() {
    // Get the singleton instance
    DatabaseConnection* connection = DatabaseConnection::getInstance();

```

```
5       // Use the connection for database operations
6       connection->query("SELECT * FROM users");
7       connection->insert("INSERT INTO products (name, price) VALUES ↩
            ('Product 1', 10.99)");
8
9       return 0;
10  }
```

3.2 Structural patterns

Structural Patterns deal with class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionality. [1]

3.2.1 Adapter

1 Definition

The **Adapter** is a structural design pattern that enables collaboration between objects with incompatible interfaces.

2 Problem

Let's consider a scenario where we are developing a stock market monitoring app. The app downloads stock data from multiple sources in XML format and presents visually appealing charts and diagrams to the user.

As we aim to enhance the app, we decide to integrate a sophisticated third-party analytics library. However, there's a challenge: the analytics library exclusively supports data in JSON format.

One possible solution is to modify the library itself to handle XML data. However, this approach carries the risk of potentially breaking existing code that relies on the library. Moreover, we might not have access to the library's source code, making it impossible to implement this solution.

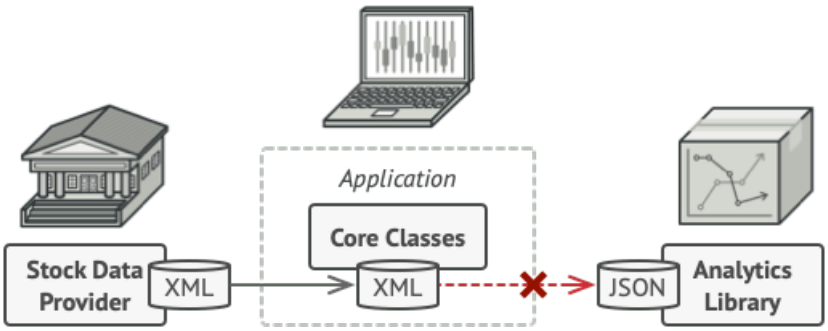


Figure 3.23: You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

3 Solution

You can utilize an **adapter** to address the issue. An adapter is a specialized object that converts the interface of one object into a format understandable by another object.

The adapter acts as a wrapper around one of the objects, shielding the underlying complexity of the conversion process. The wrapped object remains oblivious to the presence of the adapter. For instance, you can wrap an object that operates in meters and kilometers with an adapter that converts all the data into imperial units such as feet and miles.

Adapters not only facilitate data conversion into different formats but also enable objects with disparate interfaces to collaborate. Here's a breakdown of how it functions:

1. The adapter acquires an interface compatible with one of the existing objects.
2. Leveraging this interface, the existing object can securely invoke the adapter's methods.

[1] https://www.gofpatterns.com/design-patterns/module2/behavioral-creational-structural.php

3. Upon receiving a call, the adapter transfers the request to the second object, but in a format and order expected by the second object.

In certain cases, it is even possible to create a *two-way adapter* that can convert the calls in both directions.

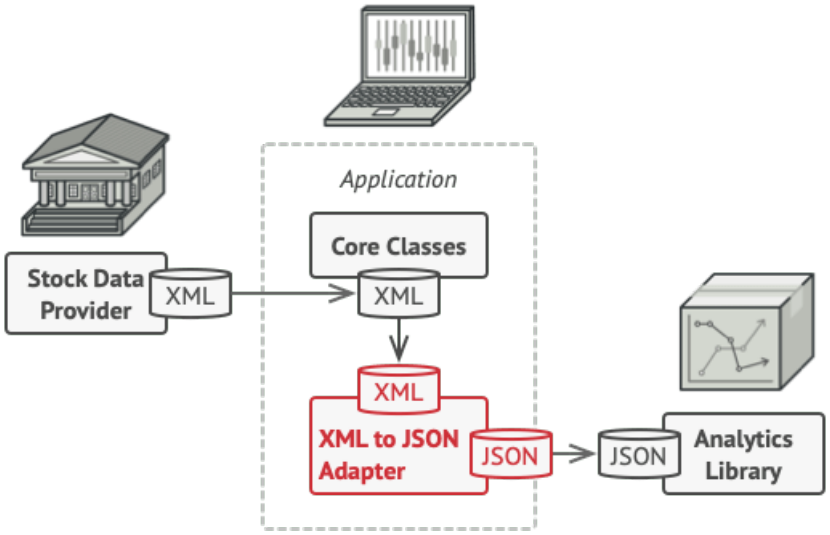


Figure 3.24: Solution using Adapter pattern design

Now, let's revisit our stock market application scenario. To resolve the problem of incompatible formats, you can create *XML-to-JSON adapters* for each class in the analytics library that your code directly interacts with. Subsequently, you adjust your code to communicate solely with the library through these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and forwards the call to the appropriate methods of the wrapped analytics object.

4 Structure

This implementation adheres to the *object composition* principle, where the adapter implements the interface of one object and wraps the other object. This approach can be implemented in various programming languages, making it versatile and applicable in different development environments.

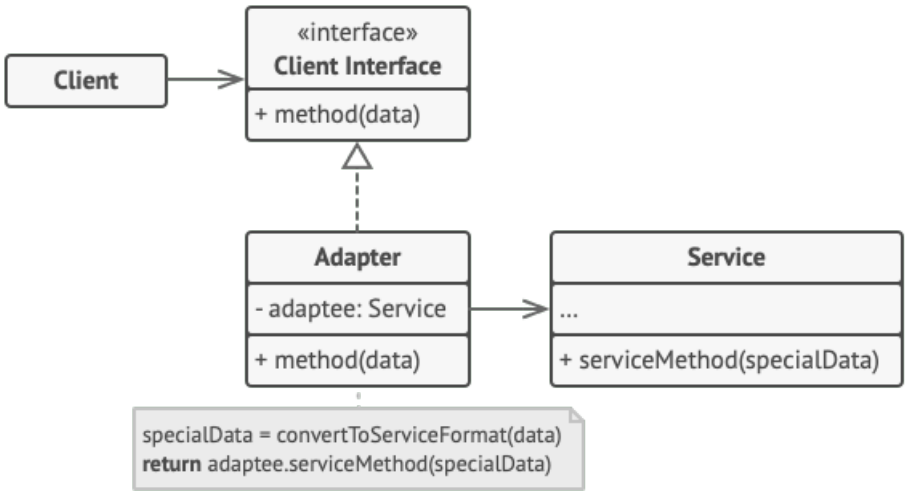


Figure 3.25: Structure of Adapter pattern design

The `Client` class contains the existing business logic of the program. It relies on the **Client Interface**, which defines a protocol that other classes must adhere to in order to collaborate with the client

code effectively.

The `Service` class represents a useful class, often from a third-party or legacy source. However, the client cannot directly utilize this class due to its incompatible interface.

To bridge the gap between the client and the service, the `Adapter` class is introduced. This class is capable of working with both the client and the service. It implements the client interface while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format that the service can understand.

The client code remains decoupled from the specific adapter implementation, as long as it interacts with the adapter through the client interface. This flexibility allows for the introduction of new adapter types into the program without breaking the existing client code. This becomes particularly valuable when the interface of the service class undergoes changes or is replaced. In such cases, a new adapter class can be created without requiring modifications to the client code.

5 Implementation

To demonstrate the Adapter pattern, ensure that you have at least two classes with incompatible interfaces:

1. **Useful service class**, which you are unable to modify due to it being a third-party library, legacy code, or having numerous existing dependencies.
2. One or more **client classes** that would benefit from utilizing the functionality provided by the service class.

To establish communication between the client and the service, follow these steps:

1. Declare the **client interface**, which defines the methods and protocol through which clients interact with the service.
2. Create the **adapter class** and ensure it adheres to the client interface. At this stage, leave all the methods within the adapter class empty.
3. Add a field to the adapter class to store a **reference to the service object**. Typically, this field is initialized through the constructor. However, in certain scenarios, it may be more convenient to pass it to the adapter when calling its methods.
4. Implement each method of the client interface within the adapter class. The adapter should delegate most of the actual work to the service object, focusing primarily on handling interface or data format conversions.
5. Clients should interact with the adapter solely through the **client interface**. By doing so, you can modify or extend the adapters without affecting the client code. This ensures flexibility and enables future adaptations or improvements without requiring changes in the client implementation.

6 Advantages and disadvantages

The Adapter pattern aligns with the **Single Responsibility Principle** by allowing the separation of interface or data conversion code from the primary business logic of the program. This promotes modular design and improves code organization by isolating the adapter's responsibilities to handling interface compatibility or data format conversions.

Moreover, the Adapter pattern adheres to the **Open/Closed Principle** as it enables the introduction of new adapter types into the program without impacting the existing client code. As long as the new adapters conform to the client interface, they can seamlessly integrate with the client code. This extensibility ensures that the system remains open for future adaptation and enhancement without requiring modifications to the existing client implementation.

However, the Adapter pattern introduces additional complexity to the codebase as it requires the introduction of new interfaces and classes. This can lead to increased overall complexity, especially when multiple adapters are involved.

7 Example

You have a legacy library in C++ that provides functionality for reading and writing data in a specific file format. However, you need to integrate this library into your modern C++ codebase that expects data to be in a different format. Directly using the legacy library would require significant modifications to your existing code.

Solution:

Step 1: Create the `LegacyLibrary` class that represents the legacy library and its functionalities.

```
class LegacyLibrary {
    public:
        void readDataFromLegacyFormat(const std::string& fileName);
        void writeDataToLegacyFormat(const std::string& fileName);
};
```

Step 2: Define the `ModernInterface` that represents the interface expected by your modern codebase.

```
class ModernInterface {
    public:
        virtual void readData(const std::string& fileName) = 0;
        virtual void writeData(const std::string& fileName) = 0;
        virtual ~ModernInterface() = default;
};
```

Step 3: Create the `LegacyAdapter` class that acts as a bridge between the `LegacyLibrary` and the `ModernInterface`. This adapter implements the `ModernInterface` and internally uses the `LegacyLibrary` to perform the necessary operations.

```
class LegacyAdapter : public ModernInterface {
    private:
        LegacyLibrary legacyLibrary;
    public:
        void readData(const std::string& fileName) override {
            legacyLibrary.readDataFromLegacyFormat(fileName);
        // Perform necessary conversions to modern format
        }

    void writeData(const std::string& fileName) override {
        // Perform necessary conversions from modern format
        legacyLibrary.writeDataToLegacyFormat(fileName);
    }
};
```

Step 4: Update your modern code to use the `ModernInterface` instead of directly interacting with the `LegacyLibrary`. This allows for seamless integration and compatibility between the legacy library and your modern codebase, without requiring modifications to the existing code.

```
void someFunction(ModernInterface& adapter) {
    adapter.readData("data.txt");
    // Use the data in the modern format
    adapter.writeData("output.txt");
}

int main() {
    LegacyAdapter adapter;
    someFunction(adapter);
    return 0;
}
```

3.2.2 Bridge

1 Definition

The Bridge pattern is a structural design pattern that enables the division of a large class or a group of closely related classes into two distinct hierarchies: abstraction and implementation. These two hierarchies can be developed independently of each other, allowing for greater flexibility and scalability in the system.

2 Problem

Let's explore a straightforward example to understand them better.

- Imagine you have a class called Shape that represents geometric shapes, with two subclasses: `Circle` and `Square`.
- Now, you want to introduce colors into this class hierarchy. To achieve that, you plan to create subclasses for Red and Blue shapes.
- However, with the existing subclasses, you would need to create four combinations of classes, such as `BlueCircle` and `RedSquare`, to incorporate both shape types and colors.

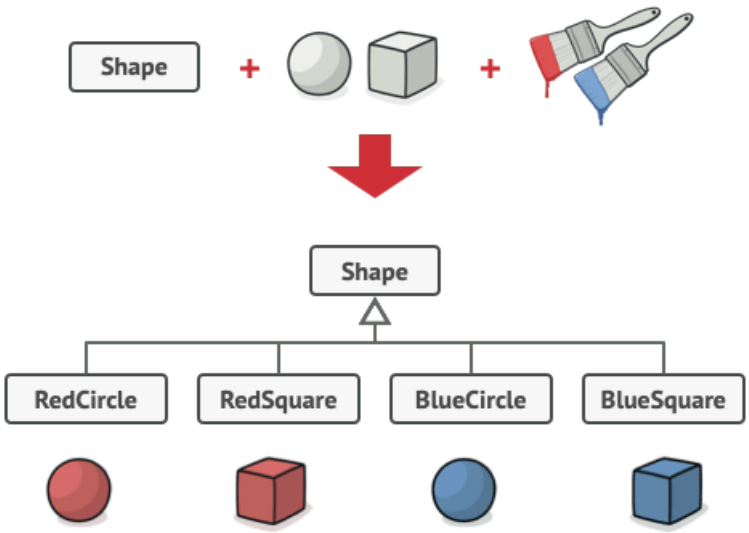


Figure 3.26: Number of class combinations grows in geometric progression.

Adding additional shape types and colors to the hierarchy leads to exponential growth. For instance, to include a triangle shape, two subclasses would be required, one for each color. Furthermore, introducing a new color would necessitate the creation of three subclasses, each representing a different shape type. As we continue this pattern, the situation progressively worsens.

3 Solution

The problem arises when we attempt to expand the shape classes in two separate dimensions: by form and by color. This is a common challenge encountered with class inheritance.

The Bridge pattern addresses this problem by shifting from inheritance to object composition. This involves extracting one of the dimensions into a distinct class hierarchy. As a result, the original classes will reference an object from the new hierarchy instead of containing all the state and behaviors within a single class.

By employing this approach, we can separate the code related to colors into a dedicated class consisting of two subclasses: `Red` and `Blue`. The `Shape` class is then equipped with a reference field that points to

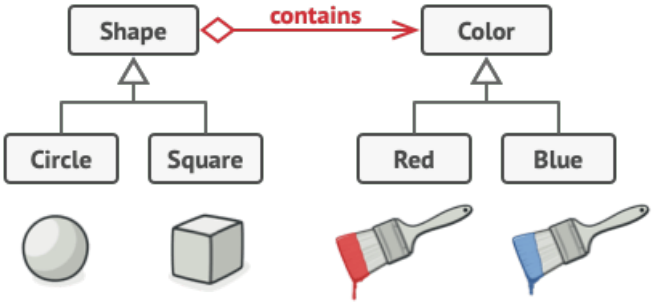


Figure 3.27: You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

one of these color objects. Consequently, the shape can delegate any color-related tasks to the associated color object. This reference acts as a bridge connecting the `Shape` and `Color` classes. Importantly, this design allows for the addition of new colors without altering the shape hierarchy, and vice versa.

4 Structure

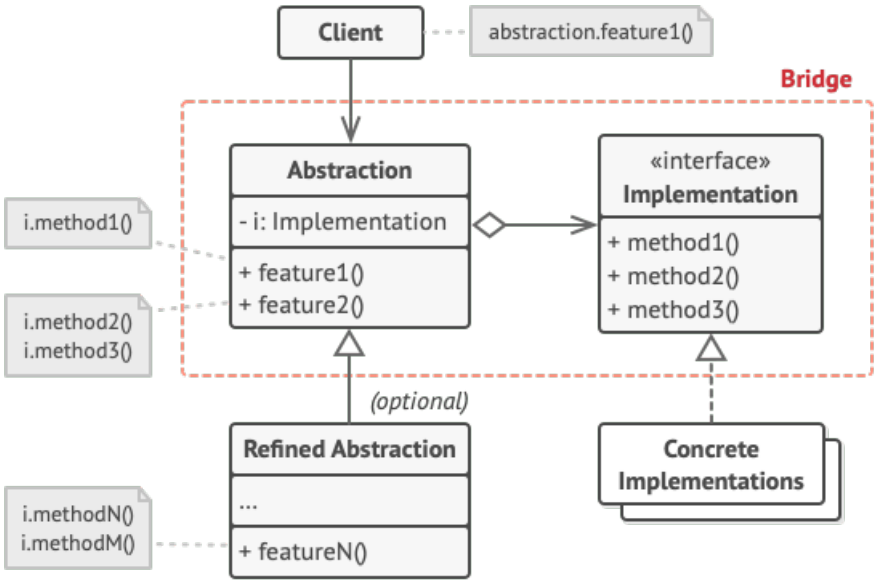


Figure 3.28: Structure of Bridge pattern

The **Abstraction** offers a higher-level control logic, while delegating the execution of low-level tasks to the implementation object.

The **Implementation** defines the common interface for all concrete implementations. The **Abstraction** can only interact with the implementation object through the methods declared in this interface.

While the abstraction may have some of the same methods as the implementation, it typically declares complex behaviors that depend on a range of primitive operations declared by the implementation.

Concrete Implementations contain platform-specific code.

Refined Abstractions offer variations of control logic. Similar to their parent, they interact with different implementations through the general implementation interface.

Typically, the **Client** is primarily concerned with working with the abstraction. However, it is the responsibility of the client to establish the connection between the abstraction object and one of the implementation objects.

5 Implementation

Here are the steps to follow:

1. Identify the orthogonal dimensions in your classes. These independent concepts could be: abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation.
2. Determine the operations required by the client and define them in the base abstraction class.
3. Identify the operations available on all platforms. Declare the necessary operations in the general implementation interface.
4. Create concrete implementation classes for all platforms in your domain, ensuring that they adhere to the implementation interface.
5. Inside the abstraction class, add a reference field for the implementation type. The abstraction delegates most of the work to the implementation object referenced in that field.
6. If you have multiple variants of high-level logic, create refined abstractions for each variant by extending the base abstraction class.
7. The client code should provide an implementation object to the abstraction's constructor to establish the association between the two. Afterward, the client can work exclusively with the abstraction object, disregarding the implementation details.

6 Advantages and disadvantages

- Platform independence: The pattern allows the creation of platform-independent classes and applications.
- Abstraction-focused client code: Clients work with high-level abstractions and are shielded from platform-specific details.
- Open/Closed Principle: The pattern enables the introduction of new abstractions and implementations independently, promoting extensibility.
- Single Responsibility Principle: It allows the separation of high-level logic in the abstraction and platform details in the implementation, promoting better code organization.

Disadvantages:

- Increased code complexity: Applying the pattern to a highly cohesive class may introduce additional complexity to the codebase.

7 Example

Suppose we have a drawing application that can draw different shapes such as circles, rectangles, and squares. We also want to support different rendering systems like raster and vector rendering. However, we want to keep the rendering implementation separate from the shape classes to ensure flexibility and extensibility.

Solution:
Step 1: Define the shape classes hierarchy.

```
// Step 1: Define the shape classes hierarchy
class Shape {
public:
```

```
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() override {
        // Draw circle implementation
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        // Draw rectangle implementation
    }
};

// Add more shape classes if needed
```

Step 2: Define the rendering interface and its implementations.

```
// Step 2: Define the rendering interface and its implementations
class Renderer {
public:
    virtual void render() = 0;
};

class RasterRenderer : public Renderer {
public:
    void render() override {
        // Raster rendering implementation
    }
};

class VectorRenderer : public Renderer {
public:
    void render() override {
        // Vector rendering implementation
    }
};
```

Step 3: Implement the bridge between the shape classes and the rendering implementations.

```
// Step 3: Implement the bridge between the shape classes and the ↩
   rendering implementations
class ShapeRenderer {
protected:
    Renderer* renderer;

public:
    ShapeRenderer(Renderer* renderer) : renderer(renderer) {}

    virtual void draw() = 0;
};

```

```
class CircleRenderer : public ShapeRenderer {
public:
    CircleRenderer(Renderer* renderer) : ShapeRenderer(renderer) {}

    void draw() override {
        // Additional circle-specific logic if needed

        renderer->render();
    }
};

class RectangleRenderer : public ShapeRenderer {
public:
    RectangleRenderer(Renderer* renderer) : ShapeRenderer(renderer) {}

    void draw() override {
        // Additional rectangle-specific logic if needed

        renderer->render();
    }
};

// Add more shape renderer classes if needed
```

Step 4: Usage example.

```
int main() {
    // Create rendering systems
    Renderer* rasterRenderer = new RasterRenderer();
    Renderer* vectorRenderer = new VectorRenderer();

    // Create shapes with different rendering systems
    ShapeRenderer* circleRenderer = new CircleRenderer(rasterRenderer);
    ShapeRenderer* rectangleRenderer = new ↩
       RectangleRenderer(vectorRenderer);

    // Draw shapes
    circleRenderer->draw();
    rectangleRenderer->draw();

    // Clean up
    delete rasterRenderer;
    delete vectorRenderer;
    delete circleRenderer;
    delete rectangleRenderer;

    return 0;
}
```

3.2.3 Composite

1 Definition

Composite is a structural design pattern that enables the composition of objects into tree structures and facilitates working with these structures as if they were individual objects.

2 Problem

Using the **Composite** pattern is meaningful only when the fundamental model of your application can be represented as a tree structure.

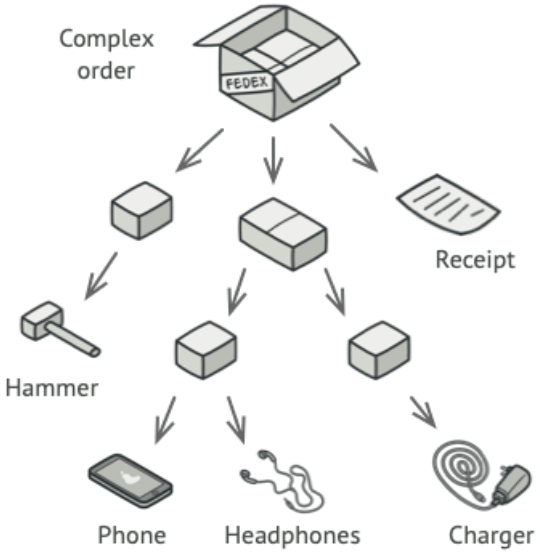


Figure 3.29: An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.

For instance, consider a scenario where you have two types of objects: `Products` and `Boxes`. A `Box` can contain multiple `Products` as well as a number of smaller `Boxes`. These smaller `Boxes` can also hold various `Products` or even smaller `Boxes`, and so on.

Now, let's say you decide to develop an ordering system that utilizes these classes. Orders may include simple products without any wrapping, as well as boxes filled with products, and even other boxes. In such a scenario, how would you determine the total price of an order?

You might consider the direct approach of unwrapping all the boxes, iterating over the products, and calculating the total price. Although this might be feasible in the real world, it's not as straightforward in a program. You would need to have prior knowledge of the classes involved (e.g., `Products` and `Boxes`), the nesting level of the boxes, and other intricate details. All of these factors make the direct approach cumbersome or even unfeasible in practice.

3 Solution

The `Composite` pattern suggests working with `Products` and `Boxes` through a common interface that defines a method for calculating the total price.

How does this method function? For a `Product`, it simply returns the price of the product. For a `Box`, the method iterates over each item within the box, retrieves its price, and computes a total for the box. If any of these items happen to be a smaller box, that box would also begin traversing its contents in a similar manner, continuing this process recursively until the prices of all inner components are calculated. Additionally, a box could include an extra cost, such as packaging expenses, to the final price calculation.

4 Structure

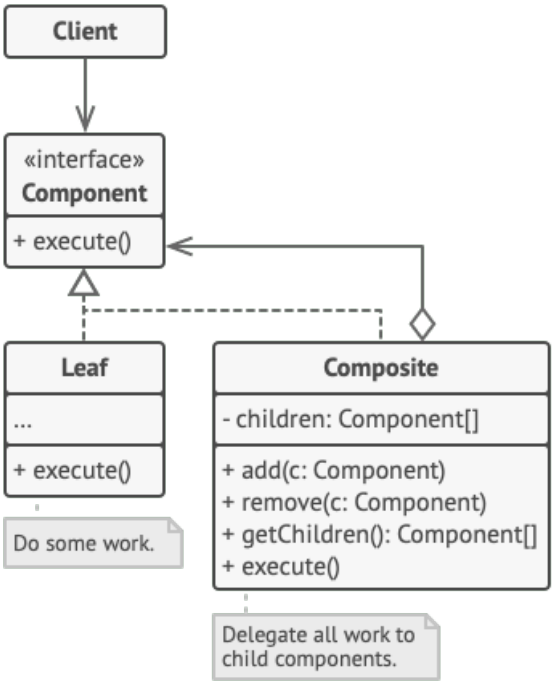


Figure 3.30: Structure of Composite pattern

The `Component` interface defines operations that are common to both simple and complex elements within the tree structure.

A `Leaf` represents a basic element in the tree that does not have any sub-elements.

In general, `Leaf` components often perform most of the actual work since they do not delegate tasks to other elements.

A `Container` (also known as a composite) is an element that contains sub-elements, which can be either `Leaf` elements or other `Container` elements. A `Container` does not have knowledge of the specific classes of its children; it interacts with all sub-elements solely through the `Component` interface.

When a request is received, a `Container` delegates the task to its sub-elements, processes intermediate results, and eventually returns the final outcome to the client.

The `Client` interacts with all elements of the tree structure through the `Component` interface. As a result, the client can seamlessly work with both simple and complex elements of the tree in a consistent manner.

5 Implementation

To implement the Composite pattern in your application, follow these steps:

1. Ensure that the core model of your application can be represented as a tree structure. Identify and break it down into simple elements (leaves) and containers (complex elements).
2. Declare the `Component` interface, listing the methods that make sense for both simple and complex components.
3. Create a `Leaf` class to represent simple elements. Your program may have multiple different leaf classes depending on the specific requirements.
4. Implement a `Container` class to represent complex elements. This class should include an array field to store references to sub-elements. Ensure that the array is declared with the type of the `Component` interface to accommodate both leaves and containers.

5. Implement the methods of the `Component` interface in the `Container` class. Remember that a container delegates most of the work to its sub-elements.
6. Define methods for adding and removing child elements in the `Container` class.

Note: These operations can be declared in the `Component` interface. It is important to note that this approach violates the Interface Segregation Principle, as the methods will be empty in the `Leaf` class. However, treating all elements equally, including when composing the tree, allows the client to work with them uniformly.

6 Advantages and disadvantages

- **Advantages**:

  - **Convenient handling of complex tree structures**: Utilize the benefits of polymorphism and recursion to work with the elements more conveniently.

  - **Open/Closed Principle**: Introduce new element types into the application without causing disruptions to the existing code that operates on the object tree.

- **Disadvantages**:

  Difficulty in providing a common interface: When classes have significantly different functionalities, it can be challenging to define a unified component interface. In such cases, there may be a need to overly generalize the component interface, which can make it harder to understand and maintain.

7 Example

Suppose we have a company that sells products. Each product can be either a single item or a composite product that consists of multiple sub-products. We need to calculate the total price of a product, including composite products, in a flexible and scalable manner.

Solution:
Step 1: Define the common interface for both single products and composite products.

```
// Step 1: Define the common interface for both single products and composite products
class Product {
public:
    virtual double getPrice() const = 0;
};
```

Step 2: Implement the single product class.

```
// Step 2: Implement the single product class
class SingleProduct : public Product {
private:
    double price;

public:
    SingleProduct(double price) : price(price) {}

    double getPrice() const override {
        return price;
```

```
    }
};
```

Step 3: Implement the composite product class.

```
// Step 3: Implement the composite product class
class CompositeProduct : public Product {
private:
    std::vector<Product*> products;

public:
    void addProduct(Product* product) {
        products.push_back(product);
    }

    void removeProduct(Product* product) {
        // Remove product from the composite
        // ...
    }

    double getPrice() const override {
        double totalPrice = 0.0;

        for (const auto& product : products) {
            totalPrice += product->getPrice();
        }

        return totalPrice;
    }
};
```

Step 4: Usage example

```
int main() {
    // Create single products
    Product* laptop = new SingleProduct(1500.0);
    Product* phone = new SingleProduct(800.0);

    // Create a composite product
    CompositeProduct* bundle = new CompositeProduct();
    bundle->addProduct(laptop);
    bundle->addProduct(phone);

    // Calculate the total price
    double totalPrice = bundle->getPrice();
    std::cout << "Total Price: $" << totalPrice << std::endl;

    // Clean up
    delete laptop;
    delete phone;
    delete bundle;

    return 0;
}
```

3.2.4 Decorator

1 Definition

The **Decorator** is a structural design pattern that enables the attachment of additional behaviors to objects. It accomplishes this by encapsulating the objects within special wrapper objects that contain the desired behaviors.

2 Problem

Imagine you are working on a notification library that allows other programs to notify their users about important events.

The initial version of the library consists of the `Notifier` class, which has a few fields, a constructor, and a single `send` method. This method accepts a message argument from the client and sends the message to a list of emails provided to the notifier through its constructor. The intended usage is for a third-party app, acting as a client, to create and configure the notifier object once and then use it whenever an important event occurs.

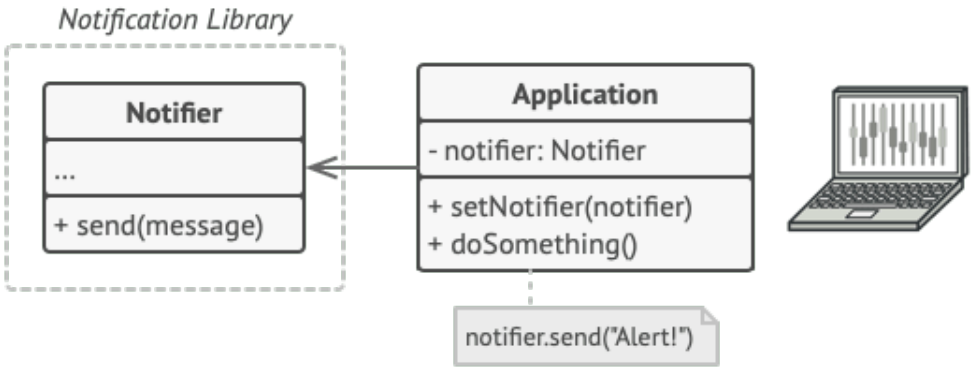


Figure 3.31: A program could use the notifier class to send notifications about important events to a predefined set of emails.

Initially, extending the `Notifier` class and adding new subclasses for additional notification methods seemed like a viable solution. The client would instantiate the desired notification class and use it for all future notifications.

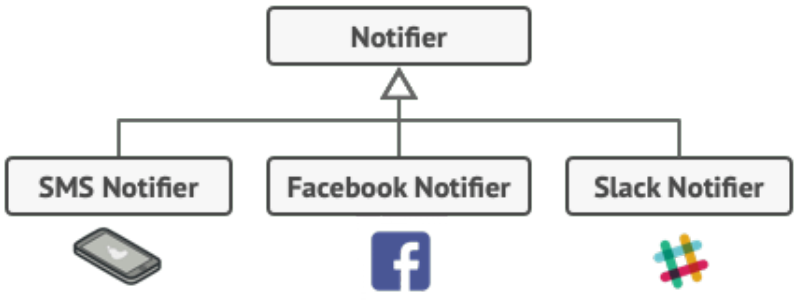


Figure 3.32: Each notification type is implemented as a notifier's subclass.

However, a reasonable question arose: "Why can't we use multiple notification types simultaneously? For instance, if your house is on fire, you'd want to be notified through every available channel."

To address this requirement, you attempted to create special subclasses that combined multiple notification methods within a single class. Unfortunately, this approach quickly proved to be impractical, as it would lead to excessively bloated code, both in the library and the client.

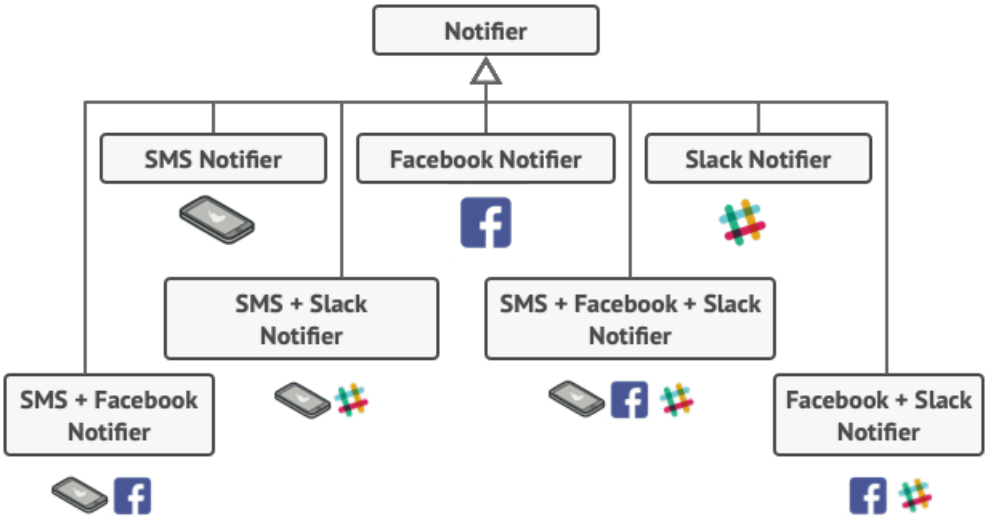


Figure 3.33: Combinatorial explosion of subclasses.

You are now faced with the challenge of finding an alternative way to structure the notification classes that avoids an excessive proliferation of classes. The goal is to maintain flexibility while keeping the codebase manageable, without unnecessarily increasing the number of notification classes.

3 Solution

Extending a class is often the initial approach when you need to modify an object's behavior. However, inheritance has certain limitations that need to be considered:

- **Inheritance is static:** The behavior of an existing object cannot be altered at runtime. Only the entire object can be replaced with another one created from a different subclass.
- **Subclasses can have only one parent class:** In most programming languages, inheritance does not allow a class to inherit behaviors from multiple classes simultaneously.

To overcome these limitations, Aggregation or Composition can be used as alternatives to inheritance. Both approaches involve one object having a reference to another object and delegating some work to it. In contrast to inheritance, where the object itself performs the work inherited from its superclass, aggregation/composition allows for easy substitution of the referenced "helper" object, thereby changing the container object's behavior at runtime. By having references to multiple objects and delegating different work to each of them, an object can incorporate behaviors from various classes. Aggregation/composition is a fundamental principle behind many design patterns, including the Decorator pattern, which we will now revisit.

Figure 3.34: Inheritance vs. Aggregation

The Decorator pattern is also known as the "Wrapper" pattern, as this name accurately reflects its main idea. A wrapper is an object that can be associated with a target object. The wrapper implements

the same set of methods as the target object and delegates all requests it receives to the target. However, the wrapper can modify the result by performing additional actions before or after passing the request to the target.

When does a simple wrapper become a true decorator? As mentioned earlier, the wrapper implements the same interface as the wrapped object, making them indistinguishable from the client's perspective. To enable wrapping an object with multiple decorators and combining their behaviors, the reference field of the wrapper should accept any object that follows the common interface.

In our notifications example, we can keep the basic email notification behavior within the base `Notifier` class, while converting all other notification methods into decorators.

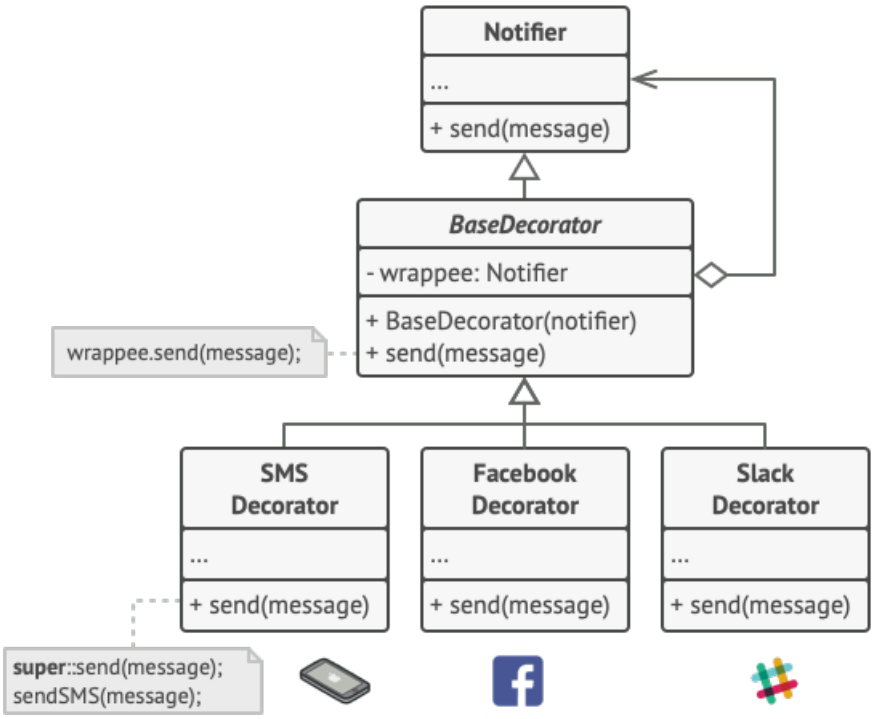


Figure 3.35: Various notification methods become decorators.

The client code would then wrap a basic notifier object with a set of decorators that align with the client's preferences.

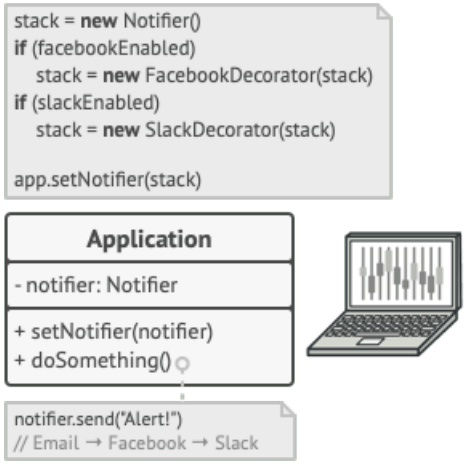


Figure 3.36: Apps might configure complex stacks of notification decorators.

The resulting objects would be organized in a stack-like structure.

4 Structure

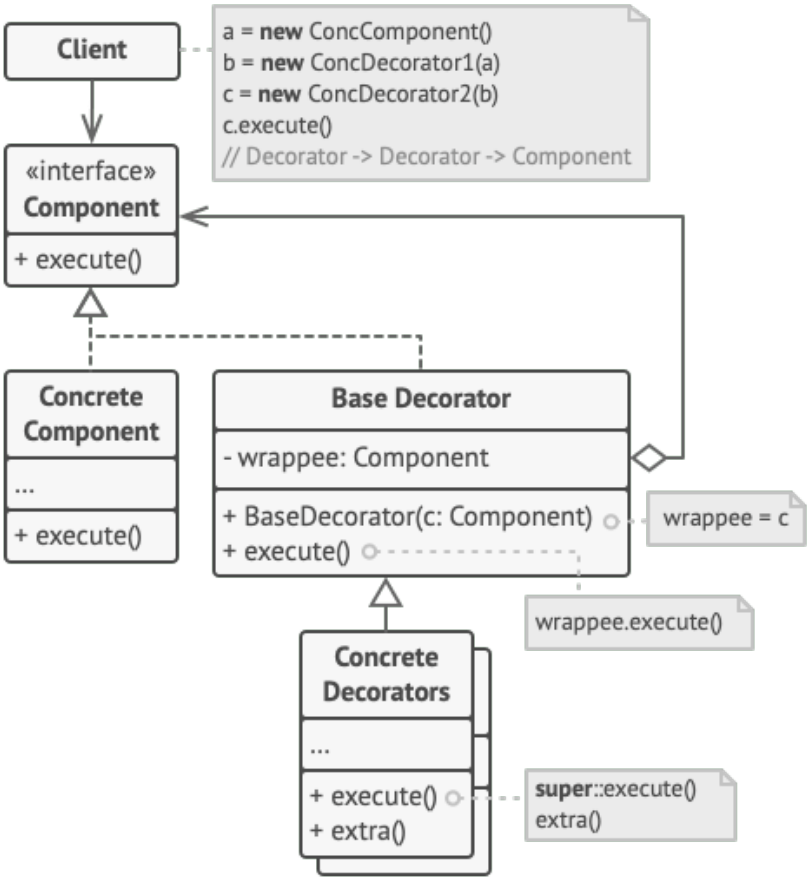


Figure 3.37: Structure of Decorator

The `Component` declares the common interface for both wrappers and wrapped objects.

`Concrete` Component is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

The `Base Decorator` class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

`Concrete Decorators` define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

The `Client` can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

5 Implementation

To implement the Decorator pattern, follow these steps:

1. Identify the primary component in your business domain that will serve as the base for decoration.
2. Determine the methods that are common to both the primary component and the optional layers of decoration. These methods should be declared in a component interface.
3. Create a concrete component class that implements the component interface. This class represents the base behavior of the component.

4. Create a base decorator class that serves as the foundation for all decorators. This class should have a field to store a reference to the wrapped object, which can be either a concrete component or another decorator. The field should be declared with the component interface type to allow flexibility in linking to different objects.

5. Ensure that all classes, including concrete components and decorators, implement the component interface. This allows them to be treated uniformly by the client code.

6. Create concrete decorator classes by extending the base decorator class. Each concrete decorator adds specific behavior before or after delegating the call to the wrapped object's method.

7. The client code is responsible for creating decorators and composing them in the desired way. The client can wrap the primary component with multiple layers of decorators to add or modify functionality as needed.

6 Advantages and disadvantages

Advantages:

- **Extension of behavior:** The pattern allows you to extend an object's behavior without the need to create a new subclass. This promotes flexibility and avoids the potential class explosion that can occur with traditional inheritance.
- **Runtime behavior modification:** Decorators provide the ability to add or remove responsibilities from an object at runtime. This enables dynamic customization of an object's behavior based on changing requirements or conditions.
- **Combination of behaviors:** By wrapping an object with multiple decorators, you can combine several behaviors together. This allows for the creation of complex and customizable combinations of functionality.
- Single **Responsibility Principle:** The Decorator pattern helps in adhering to the Single Responsibility Principle by allowing you to divide a monolithic class that implements many possible variants of behavior into smaller and more specialized classes. Each decorator class has a specific responsibility, making the codebase more modular and maintainable.

Disadvantages:

- **Removal of specific decorators:** It can be challenging to remove a specific decorator from the decorators stack, especially if the order of decorators is important. Removing a decorator may require restructuring the entire stack of decorators, which can introduce complexity and potential errors.
- **Dependency on decorator order:** The behavior of a decorator may depend on its position in the decorators stack. If the order of decorators is not carefully managed, it may result in unexpected or undesired behavior.
- **Initial configuration code complexity:** Setting up the initial configuration of layers, especially when dealing with multiple decorators, can sometimes lead to complex and potentially verbose code. Managing the composition of decorators and their order of application may require careful planning and implementation.

7 Example

Imagine we have a simple coffee shop application that offers different types of coffee. Each coffee can have various additives like milk, sugar, or caramel. We want to dynamically add or remove these additives to our coffee objects without modifying their individual classes.

Solution:

Step 1: Define the base component interface.

```
// Step 1: Define the base component interface
class Coffee {
public:
    virtual std::string getDescription() const = 0;
    virtual double getCost() const = 0;
};
```

Step 2: Implement the concrete component class.

```
// Step 2: Implement the concrete component class
class SimpleCoffee : public Coffee {
public:
    std::string getDescription() const override {
        return "Simple Coffee";
    }

    double getCost() const override {
        return 1.0;
    }
};
```

Step 3: Define the decorator class hierarchy.

```
// Step 3: Define the decorator class hierarchy
class CoffeeDecorator : public Coffee {
protected:
    Coffee* decoratedCoffee;

public:
    CoffeeDecorator(Coffee* coffee) : decoratedCoffee(coffee) {}

    std::string getDescription() const override {
        return decoratedCoffee->getDescription();
    }

    double getCost() const override {
        return decoratedCoffee->getCost();
    }
};

class MilkDecorator : public CoffeeDecorator {
public:
    MilkDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}

```

```
    std::string getDescription() const override {
        return CoffeeDecorator::getDescription() + ", Milk";
    }

    double getCost() const override {
        return CoffeeDecorator::getCost() + 0.5;
    }
};

class SugarDecorator : public CoffeeDecorator {
public:
    SugarDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}

    std::string getDescription() const override {
        return CoffeeDecorator::getDescription() + ", Sugar";
    }

    double getCost() const override {
        return CoffeeDecorator::getCost() + 0.2;
    }
};

// Add more decorators if needed
```

Step 4: Usage example.

```
int main() {
    // Create a simple coffee
    Coffee* simpleCoffee = new SimpleCoffee();
    std::cout << "Description: " << simpleCoffee->getDescription() << ↩
        std::endl;
    std::cout << "Cost: $" << simpleCoffee->getCost() << std::endl;

    // Add decorators to the coffee
    Coffee* coffeeWithMilk = new MilkDecorator(simpleCoffee);
    std::cout << "Description: " << coffeeWithMilk->getDescription() << ↩
        std::endl;
    std::cout << "Cost: $" << coffeeWithMilk->getCost() << std::endl;

    Coffee* coffeeWithMilkAndSugar = new SugarDecorator(coffeeWithMilk);
    std::cout << "Description: " << ↩
        coffeeWithMilkAndSugar->getDescription() << std::endl;
    std::cout << "Cost: $" << coffeeWithMilkAndSugar->getCost() << ↩
        std::endl;

    // Clean up
    delete simpleCoffee;
    delete coffeeWithMilk;
    delete coffeeWithMilkAndSugar;

    return 0;
}
```

3.2.5 Facade

1 Definition

Facade is a structural design pattern that offers a simplified interface to a library, framework, or any other intricate set of classes.

2 Problem

Suppose you are required to integrate your code with a comprehensive library or framework consisting of numerous objects. Typically, this would involve initializing the objects, managing dependencies, ensuring correct method execution order, and more.

Consequently, the business logic of your classes would become tightly coupled with the internal implementation details of the third-party classes. This coupling would make the codebase complex, challenging to understand, and difficult to maintain.

3 Solution

A **facade** is a class that offers a simplified interface to a complex subsystem, which consists of numerous interconnected components. While the facade may provide limited functionality compared to directly working with the subsystem, it exposes only the features that are essential to clients.

Having a facade proves beneficial when integrating an application with an intricate library that offers a multitude of features, while only a small portion of its functionality is required.

For example, consider an application that uploads short funny videos featuring cats to social media platforms. Although it could potentially utilize a professional video conversion library with numerous capabilities, all it truly needs is a class containing a single method, `encode(filename, format)`. By creating such a class and establishing the necessary connections with the video conversion library, you create your first facade, allowing for simplified interaction with the underlying complex subsystem.

4 Structure

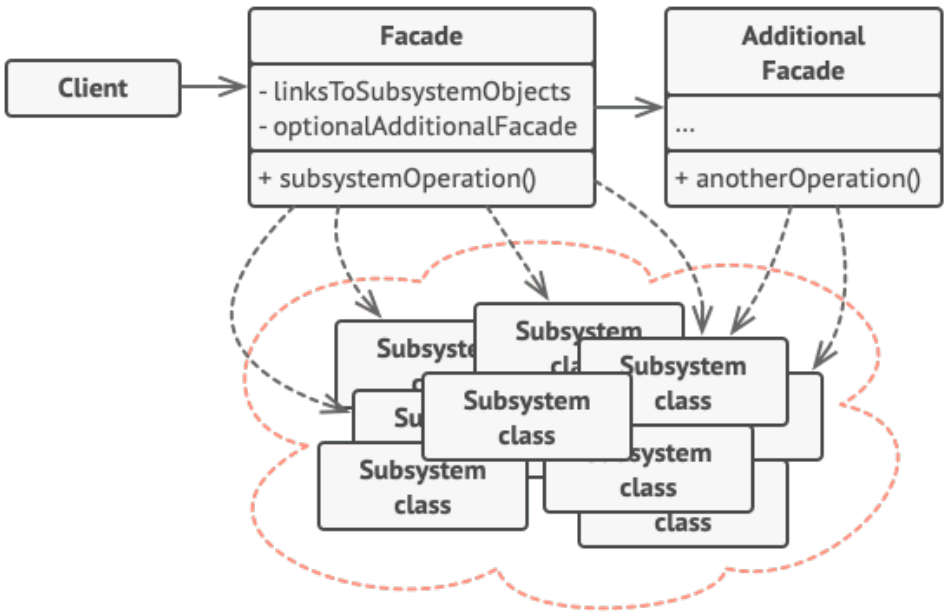


Figure 3.38: Structure of Facade

The **Facade** class provides a convenient and simplified access point to a specific part of the subsystem's functionality. It possesses knowledge of where to direct client requests and how to orchestrate the interactions among the various components within the subsystem.

In certain cases, it may be beneficial to create **Additional Facade** classes to avoid cluttering a single facade with unrelated features. These additional facades can be utilized by both clients and other facades, providing a modular and cohesive approach.

The **Complex Subsystem** encompasses numerous objects, requiring intricate handling to achieve meaningful functionality. To accomplish this, it becomes necessary to delve into the internal implementation details of the subsystem. This involves tasks such as initializing objects in the correct order and providing them with the appropriate data formats.

Importantly, the subsystem classes remain unaware of the existence of the facade. They operate autonomously within the system, interacting directly with each other to fulfill their respective roles.

5 Implementation

To determine if it is possible to provide a simpler interface than what the existing subsystem offers, assess whether the new interface can make the client code independent from many of the subsystem's classes. If this is achievable, proceed with the following steps:

1. Declare and implement the new interface in a dedicated facade class. The facade class should serve as an intermediary, redirecting client calls to the appropriate objects within the subsystem. It is responsible for initializing the subsystem and managing its lifecycle, unless the client code already handles these tasks.

2. To fully leverage the benefits of the pattern, ensure that all client code communicates solely through the facade. This shields the client code from any future changes in the subsystem. For instance, when the subsystem undergoes an upgrade to a new version, modifications will only be required within the facade class.

3. If the facade class becomes overly large or complex, consider extracting specific behaviors into a refined facade class. This promotes better organization and maintains a modular structure within the facade implementation.

6 Advantages and disadvantages

Using a facade allows you to isolate your code from the inherent complexity of a subsystem. By providing a simplified interface, the facade shields the client code from the intricacies and internal workings of the subsystem, promoting a more straightforward and intuitive interaction.

However, it is important to note that there is a potential drawback with facades. In some cases, a facade can become a "god object" that becomes tightly coupled to all classes within an application. This can lead to issues such as reduced maintainability, increased dependencies, and decreased flexibility. It is crucial to strike a balance and ensure that the facade remains focused on providing a streamlined interface without accumulating excessive responsibilities.

7 Example

Suppose we have a complex subsystem consisting of multiple classes with intricate dependencies. Accessing and using these classes directly can be cumbersome for clients, especially when they only need to perform high-level operations. We want to simplify the interface and provide a unified entry point to the subsystem to improve usability and maintainability.

Solution

Step 1: Identify the classes and their dependencies in the subsystem.

```
class SubsystemClass1 {
public:
    void operation1() {
        // Implementation of operation 1
    }
};

class SubsystemClass2 {
    public:
        void operation2() {
        // Implementation of operation 2
    }
};

class SubsystemClass3 {
public:
    void operation3() {
        // Implementation of operation 3
    }
};
```

Step 2: Create a Facade class that provides a simplified interface to the subsystem.

```
class Facade {
private:
    SubsystemClass1 subsystem1;
    SubsystemClass2 subsystem2;
    SubsystemClass3 subsystem3;

public:
    void performOperation() {
        subsystem1.operation1();
        subsystem2.operation2();
        subsystem3.operation3();
        // Additional high-level operations if needed
    }
};
```

Step 3: Clients can now use the Facade class to interact with the subsystem without worrying about its internal complexity.

```
int main() {
    Facade facade;
    facade.performOperation();
    return 0;
}
```

3.2.6 Flyweight

1 Definition

The **Flyweight** pattern is a structural design pattern that optimizes memory usage by allowing multiple objects to share common parts of their state instead of duplicating the data in each object.

2 Problem

To add an exciting element to your video game, you decided to incorporate a realistic particle system. This system would create a visually stunning experience with a multitude of bullets, missiles, and shrapnel flying across the map, enhancing the overall gameplay.

After completing the development, you eagerly shared the game with your friend for testing. However, your friend encountered a recurring issue on their computer. The game consistently crashed after just a few minutes of gameplay. Troubleshooting the problem, you meticulously analyzed the debug logs and identified the root cause—an insufficient amount of RAM on your friend's less powerful rig.

The culprit behind the crashes was your particle system. Each individual particle, such as a bullet, missile, or shrapnel, was represented by a distinct object containing a significant amount of data. As the intensity of the on-screen action escalated, the number of newly created particles exceeded the available RAM capacity. Consequently, the program crashed due to memory overload.

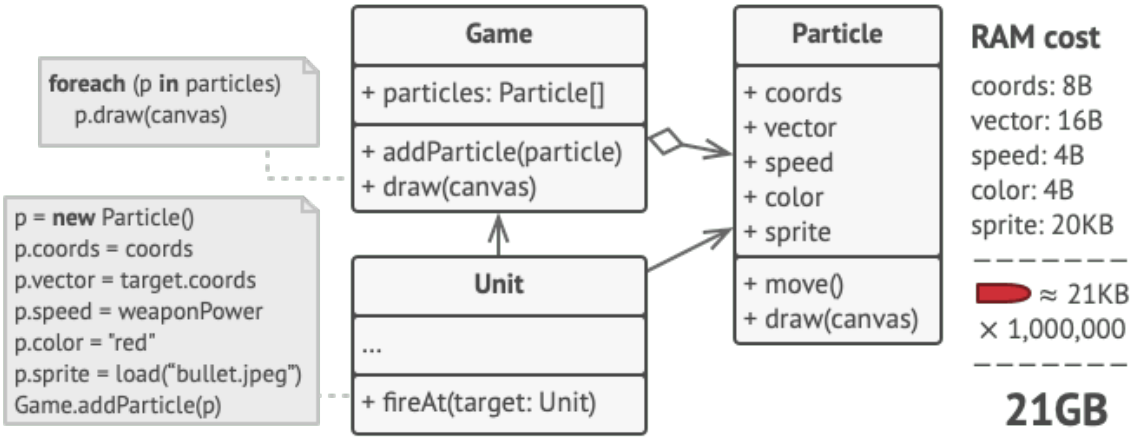
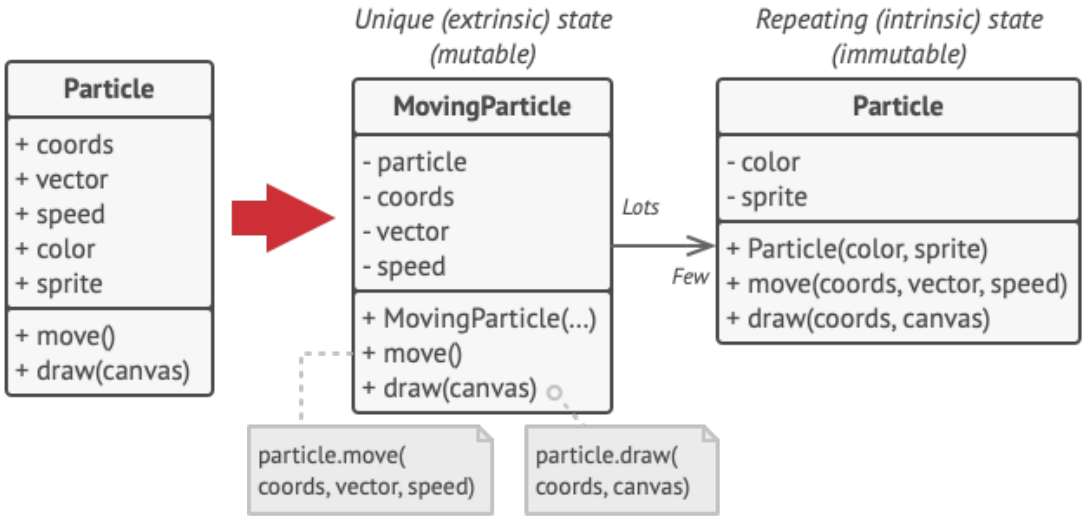


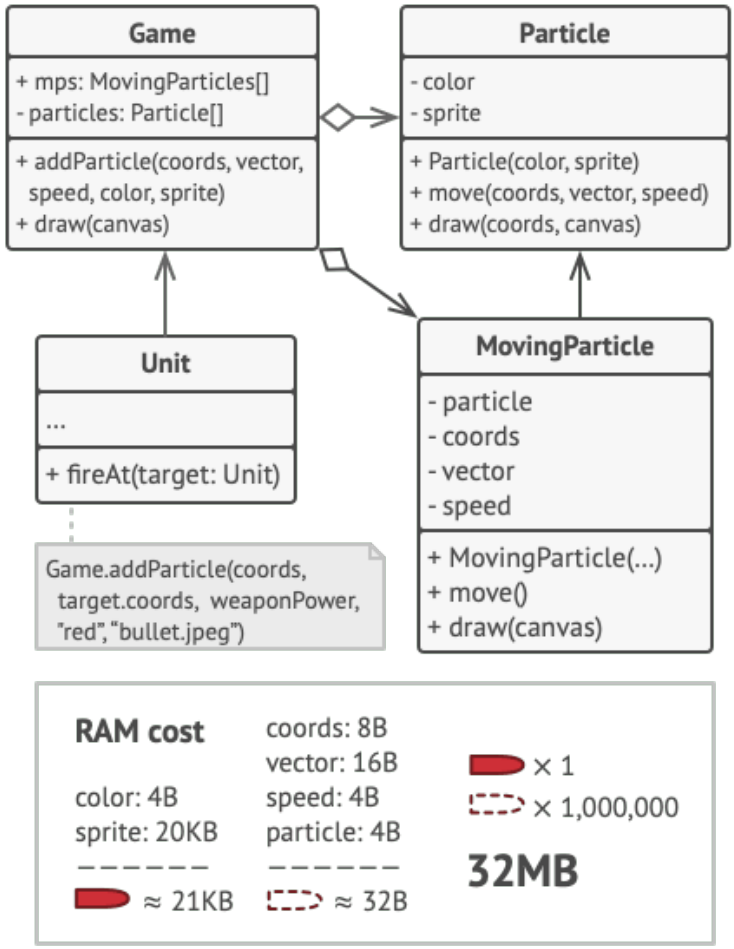
Figure 3.39: Problem

3 Solution

Upon closer examination of the `Particle` class in your game, you discovered that the color and sprite fields consume a significant amount of memory compared to other fields. What's more, these fields store nearly identical data across all particles. For instance, all bullets share the same color and sprite.



While certain aspects of a particle's state, such as coordinates, movement vector, and speed, are unique to each particle, the color and sprite remain constant. These constant attributes are referred to as the intrinsic state and reside within the object. On the other hand, the remaining aspects of the particle's state, subject to change from external sources, are known as the extrinsic state.



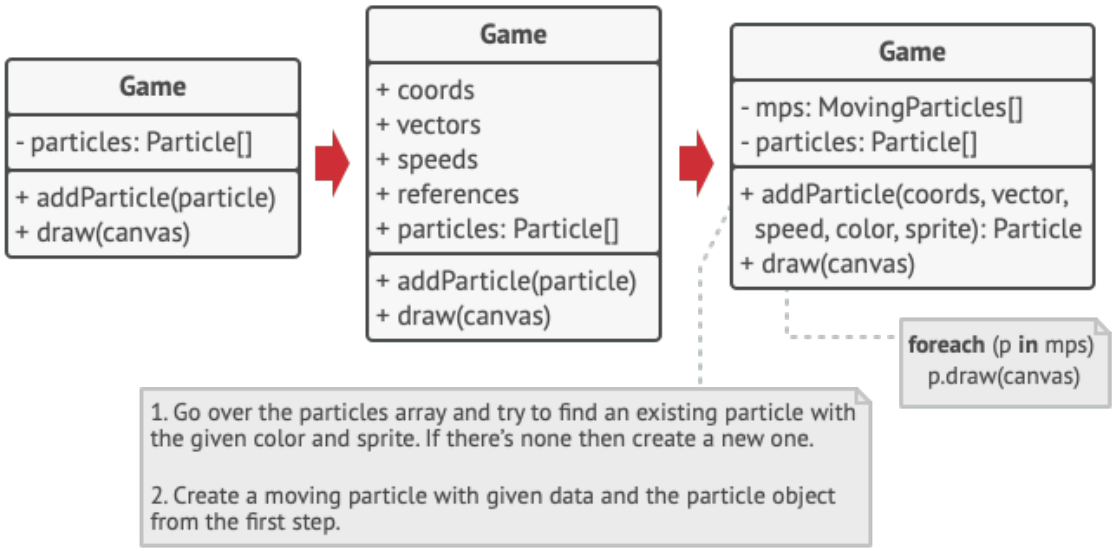
To address this issue, the Flyweight pattern suggests separating the extrinsic state from the object and passing it to specific methods as needed. By doing so, you can reuse the intrinsic state in different contexts, reducing the need for multiple objects. Objects that solely store the intrinsic state are called

flyweights.

In the context of your game, only three distinct objects, namely a bullet, a missile, and a piece of shrapnel, would be required to represent all particles. By extracting the extrinsic state, you can significantly reduce the number of objects, as the variations lie primarily in the intrinsic state.

The extrinsic state is typically stored in the container object that aggregates the objects before applying the pattern. In your case, this would be the main Game object, which stores all particles in the particles field. To accommodate the extrinsic state, you can create arrays within the container object to store coordinates, vectors, and speed for each particle. Additionally, another array can store references to the specific flyweight object representing a particle. These arrays must be synchronized to access all the necessary data using the same index.

An alternative approach involves creating a separate context class that stores the extrinsic state along with a reference to the flyweight object. This would require only a single array in the container class, resulting in a more elegant solution.



It's important to ensure that the state of a flyweight object remains immutable since it can be shared across different contexts. The flyweight should initialize its state only once through constructor parameters and should not expose any setters or public fields.

To facilitate easy access to various flyweights, you can create a flyweight factory method that manages a pool of existing flyweight objects. This method takes the intrinsic state of the desired flyweight from a client, searches for an existing flyweight object with a matching state, and returns it if found. If no match is found, a new flyweight object is created and added to the pool.

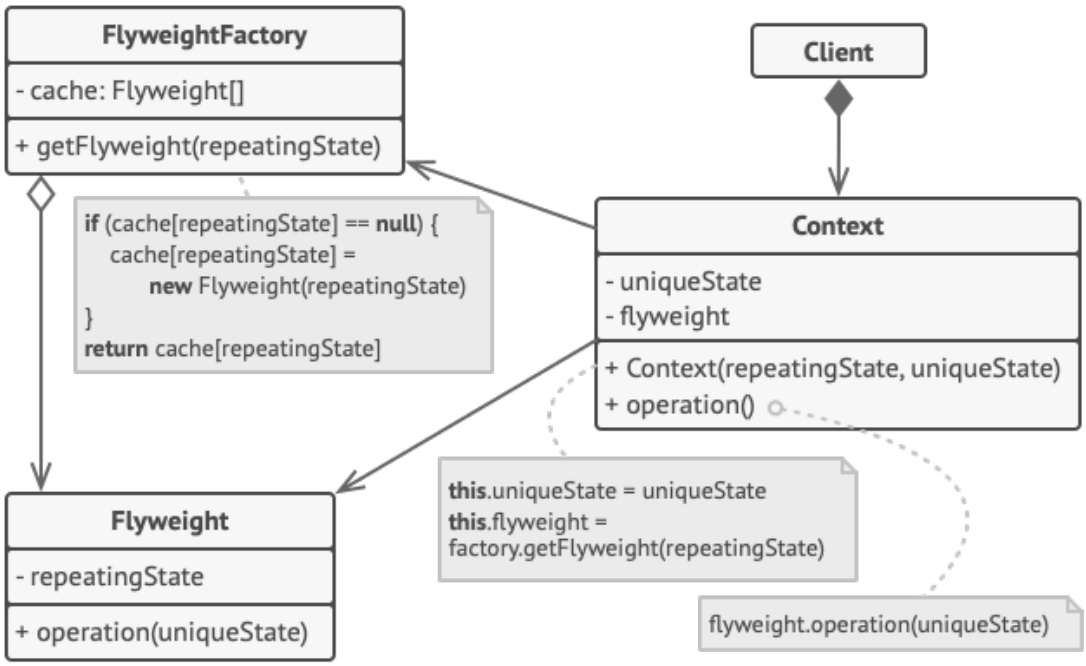
The placement of this factory method can vary. It can be incorporated into a flyweight container, a dedicated factory class, or as a static method within the flyweight class itself.

4 Structure

The `Flyweight` pattern should only be applied if your program indeed faces a RAM consumption problem due to a large number of similar objects in memory simultaneously. Before proceeding with the pattern, ensure that there is no other viable solution to address the issue.

In the Flyweight pattern, the `Flyweight` class contains the portion of the original object's state that can be shared among multiple objects. This flyweight object can be utilized in various contexts, and the state it holds is known as intrinsic state. The extrinsic state, which is unique across all original objects, is stored in the Context class. When a context is associated with a flyweight object, it represents the complete state of the original object.

Typically, the behavior of the original object remains within the flyweight class. When calling a



flyweight's method, the caller must provide the appropriate extrinsic state as parameters to the method. Alternatively, the behavior can be moved to the context class, where the flyweight is used primarily as a data object.

The `Client` is responsible for calculating or storing the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object that can be configured at runtime by passing contextual data into the method parameters.

The `Flyweight Factory` manages a pool of existing flyweights. Clients interact with the factory instead of creating flyweights directly. By providing bits of the intrinsic state of the desired flyweight, the factory searches for a matching existing flyweight and returns it if found. If no match is found, the factory creates a new flyweight and adds it to the pool.

5 Implementation

1. To transform a class into a flyweight, you need to divide its fields into two parts: the intrinsic state and the extrinsic state. The intrinsic state represents the unchanging data that is duplicated across many objects, while the extrinsic state holds the contextual data unique to each object.

2. Keep the fields representing the intrinsic state in the class and ensure that they are immutable. These fields should be assigned their initial values only within the constructor.

3. Next, review the methods that utilize the fields of the extrinsic state. For each of these fields, introduce a new parameter in the method and use it instead of accessing the field directly.

4. Optionally, you can create a factory class to manage the pool of flyweights. The factory should check if an existing flyweight with the desired intrinsic state already exists before creating a new one. Once the factory is in place, clients should request flyweights through it by providing the intrinsic state as a description of the desired flyweight.

5. Finally, the client needs to store or calculate the values of the extrinsic state (context) to be able to invoke methods on the flyweight objects. For convenience, you can create a separate context class that combines the extrinsic state and the reference to the flyweight object.

6 Advantages and disadvantages

- **Advantages:**

  Significant reduction in RAM usage when dealing with numerous similar objects.

- **Disadvantages:**

  - Increased CPU cycles may be required when recalculating context data for flyweight methods.

  - Code complexity is heightened, potentially causing confusion for new team members regarding the separation of entity state.

7 Example

Suppose we have a text editor application that needs to handle a large number of characters with different fonts. Creating individual objects for each character and font combination can lead to excessive memory usage.

Step 1: Define the flyweight class hierarchy.

```
class Character {
    public:
        virtual void draw(const std::string& font) const = 0;
    };

    class ConcreteCharacter : public Character {
        private:
            char symbol;

        public:
            ConcreteCharacter(char symbol) : symbol(symbol) {}

            void draw(const std::string& font) const override {
                std::cout << "Character: " << symbol << ", Font: " << font << std::endl;
            }
    }
};
// Add more flyweight classes if needed
```

Step 2: Implement a flyweight factory to manage the flyweight objects.

```
class CharacterFactory {
    private:
        std::map<char, Character*> characters;

    public:
        Character* getCharacter(char symbol) {
            if (characters.find(symbol) == characters.end()) {
                characters[symbol] = new ConcreteCharacter(symbol);
            }
            return characters[symbol];
        }
}
```

Step 3: Usage example.

```
int main() {
    CharacterFactory characterFactory;
    // Get flyweight objects for different characters
    Character* characterA = characterFactory.getCharacter('A');
    Character* characterB = characterFactory.getCharacter('B');
    Character* characterA2 = characterFactory.getCharacter('A');

    // Use flyweight objects with different fonts
    characterA->draw("Arial");
    characterB->draw("Times New Roman");
    characterA2->draw("Arial");

    // Clean up

    return 0;
}
```

3.2.7 Proxy

1 Definition

Proxy is a structural design pattern that enables the provision of a substitute or placeholder for another object. The proxy acts as a intermediary, controlling access to the original object. This control allows you to perform additional actions either before or after the request is forwarded to the original object.

2 Problem

Controlling access to an object can be beneficial in various scenarios. For example, consider a situation where you have a massive object that consumes a significant amount of system resources. Although you need this object occasionally, it is not required at all times.

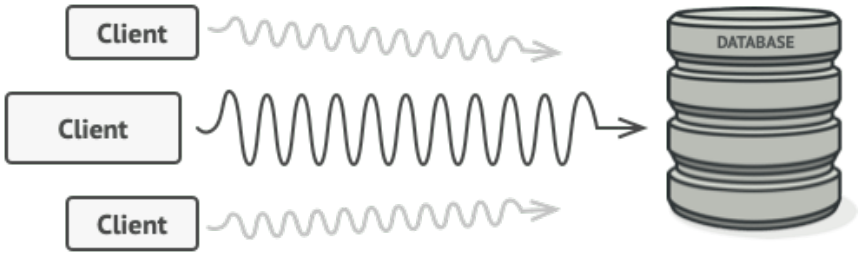


Figure 3.40: Database queries can be really slow.

One approach to address this is lazy initialization, where the object is created only when it is actually needed. However, implementing lazy initialization would require each client of the object to execute deferred initialization code, resulting in code duplication throughout the client codebase.

Ideally, we would like to incorporate this initialization code directly into the object's class. However, this may not always be possible, especially if the class is part of a closed 3rd-party library or inaccessible for modification. In such cases, utilizing the `Proxy` pattern allows us to control access to the object, enabling us to perform the deferred initialization code before or after forwarding the request to the original object. This way, we can achieve the desired lazy initialization behavior without modifying the original object's class.

3 Solution

The `Proxy` pattern suggests the creation of a new proxy class that has the same interface as the original service object. In your application, you replace the original object with the proxy object and pass the proxy to all the clients that previously used the original object. When a client sends a request, the proxy creates an instance of the real service object and delegates the actual work to it.

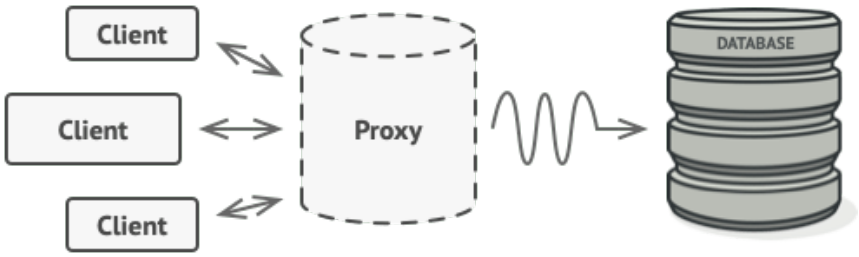


Figure 3.41: The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

Now, you may wonder about the benefits of using the proxy pattern. One key advantage is that it allows you to execute additional actions either before or after the primary logic of the class, without

modifying the original class itself. By implementing the same interface as the original class, the proxy can seamlessly substitute the real service object in any client code that expects it. This provides flexibility and transparency to the clients, as they interact with the proxy in the same way they would with the original object, unaware of the underlying proxy mechanics.

4 Structure

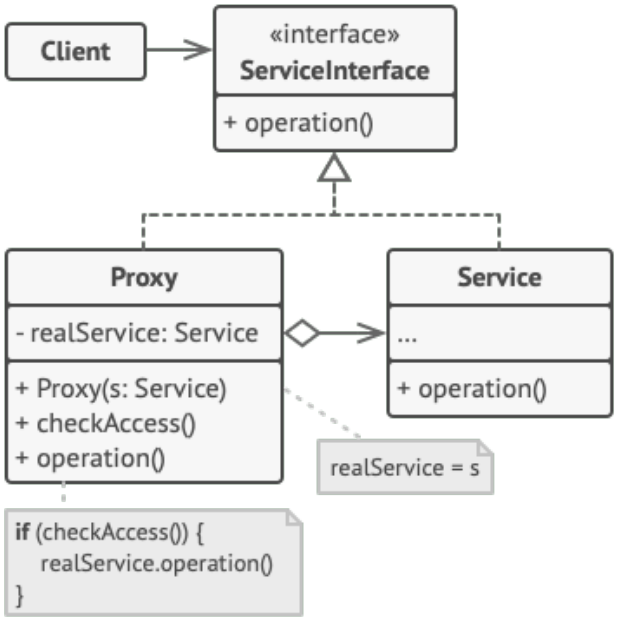


Figure 3.42: Structure of Proxy

The `Service Interface` declares the interface that the Service adheres to. In order to successfully masquerade as a service object, the proxy must conform to this interface.

The `Service` class is responsible for providing valuable business logic.

On the other hand, the `Proxy` class contains a reference field that points to a service object. Once the proxy completes its processing, such as lazy initialization, logging, access control, caching, etc., it forwards the request to the service object.

Typically, proxies are responsible for managing the entire lifecycle of their service objects.

The `Client` is designed to work with both services and proxies through the same interface. This approach enables the passing of a proxy object into any code segment that expects a service object, ensuring compatibility and seamless integration.

5 Implementation

1. If there is no pre-existing service interface, it is recommended to create one to ensure interchangeability between proxy and service objects. Creating an interface allows the proxy and service objects to be used interchangeably without requiring changes to the existing clients of the service. In cases where extracting the interface directly from the service class is not feasible due to the need to modify all the clients, an alternative approach is to make the proxy class a subclass of the service class. This way, the proxy inherits the interface of the service.

2. Once the interface is established, create the proxy class. The proxy should have a field for storing a reference to the service object. Typically, proxies are responsible for managing the entire lifecycle of their service objects. In some cases, the service object may be passed to the proxy via a constructor by the client.

3. Implement the proxy methods based on their intended purposes. In most scenarios, the proxy performs certain operations and then delegates the actual work to the underlying service object.
4. Consider introducing a creation method that determines whether the client receives a proxy or a real service object. This can be a simple static method in the proxy class or a more complex factory method.
5. Additionally, consider implementing lazy initialization for the service object. This means that the service object is only created when it is actually needed, allowing for more efficient resource utilization.

6 Advantages and disadvantages

- **Advantages:**

  - **Control over the service object**: The proxy pattern allows you to have control over the service object without the clients being aware of it. You can add additional functionality, such as access control, logging, caching, or lazy initialization, without modifying the service or impacting the clients.

  - **Lifecycle management**: Proxies enable you to manage the lifecycle of the service object when clients are not concerned with it. The proxy can handle the creation, initialization, and destruction of the service object, ensuring efficient resource utilization.

  - **Fault tolerance**: The proxy can handle scenarios where the service object is not yet ready or temporarily unavailable. It can provide default values, return cached results, or handle errors gracefully, ensuring uninterrupted operation of the system.

  - **Open/Closed Principle**: The proxy pattern allows for the introduction of new proxies without modifying the service or clients. This promotes the principle of open-closed, where you can extend the functionality by adding new proxies without altering the existing codebase.

- **Disadvantages**:

  - **Increased complexity**: Introducing the proxy pattern may add complexity to the codebase as it involves the creation of additional classes and the coordination of interactions between the proxy and the service object. This complexity needs to be managed effectively to ensure maintainability and understandability of the code.

  - **Delayed response**: In certain cases, the use of proxies may introduce a delay in the response from the service object. This can occur when additional operations or checks are performed by the proxy before forwarding the request to the service. It is important to consider the impact on performance and ensure that the delay is acceptable in the context of the application's requirements.

7 Example

Suppose we have a resource-intensive object that takes a significant amount of time to create. We want to delay the object's creation until it is actually needed, and we also want to control access to the object by performing additional checks or operations before allowing the client to use it.

Solution:
Step 1: Define the interface for the resource and its proxy.

```
// Step 1: Define the interface for the resource and its proxy
class Resource {
    public:
        virtual void performOperation() = 0;
```

```
};

class ResourceProxy : public Resource {
    private:
        Resource* resource;

    public:
        void performOperation() override {
            if (resource == nullptr) {
                resource = new Resource(); // Delayed creation of the resource
            }

            // Additional checks or operations before accessing the resource
            // ...
            resource->performOperation(); // Delegate the operation to the real resource
        }
};
```

Step 2: Use the proxy to control access to the resource.

```
int main() {
    // Use the resource proxy
    ResourceProxy proxy;
    proxy.performOperation(); // Resource creation and operation execution
    return 0;
}
```

3.3 Behavioural patterns

Behavioral Patterns are specifically concerned with communication between objects.[1]

3.3.1 Interpreter

1 Definition

The Interpreter pattern is a behavioral design pattern that provides a way to evaluate or interpret a language or expression. It is commonly used to create domain-specific languages or to implement complex grammars.

2 Problem

Let's consider a problem where we need to evaluate and interpret a set of mathematical expressions. Each expression consists of arithmetic operations (addition, subtraction, multiplication, division) and operands (numbers).

Without the Interpreter pattern, implementing the evaluation logic without the Interpreter pattern would require handling each operator and operand separately. We would need to manually parse the expression, identify the operators and operands, and perform the operations in the correct order, following the operator precedence rules. This approach can quickly become complex and error-prone, especially for more complex expressions.

3 Solution

By applying the **Interpreter** pattern, we can define a set of classes that represent different elements of an expression, such as numbers, addition, subtraction, multiplication, and division. Each class possesses the knowledge to interpret or evaluate itself.

We can establish an abstract base class named `Expression` which defines a common method, such as `interpret()`, that all concrete expression classes implement. The `interpret()` method evaluates the expression and returns the result.

For instance, we can have classes like `NumberExpression`, `AdditionExpression`, `SubtractionExpression`, `MultiplicationExpression`, and `DivisionExpression`. Each class receives the necessary operands and performs the corresponding operation in their `interpret()` method.

Utilizing the Interpreter pattern, we can construct a syntax tree from the expression and then interpret the tree to accurately evaluate the expression. The interpretation process follows the hierarchical structure of the syntax tree, simplifying the evaluation procedure.

This approach facilitates the addition of new operations or language extensions. We can introduce new expression classes for additional operations, and the interpreter handles their interpretation based on the syntax tree.

The Interpreter pattern offers a modular and maintainable solution for evaluating mathematical expressions, enabling the separation of expression interpretation logic from the client code. This makes it easier to comprehend and modify the language semantics.

4 Structure

- **AbstractionExpression**: Declares an interface for performing an operation.
- **TerminalExpression**: Implements an interpreting operation associated with terminal symbols. It represents an expression that is required for all terminal symbols in a sentence.

[1] https://www.gofpatterns.com/design-patterns/module2/behavioral-creational-structural.php

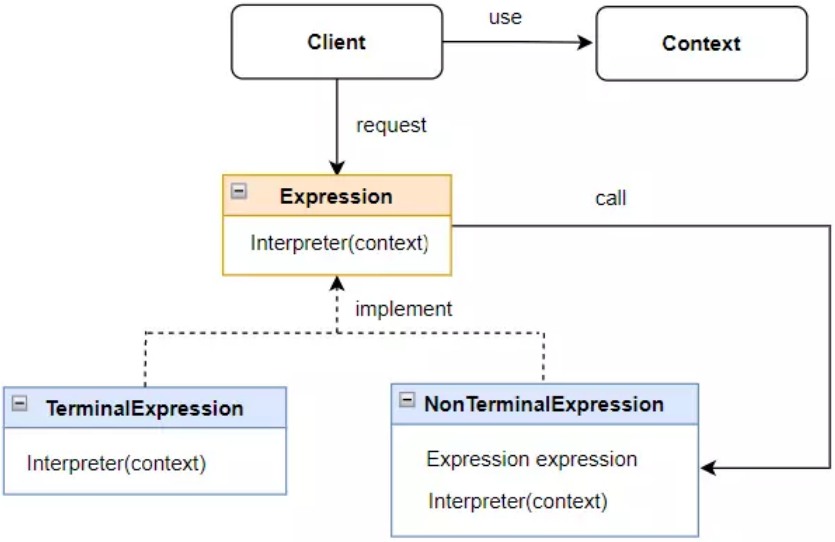


Figure 3.43: Structure of Interpreter

- **NonterminalExpression**: Can internally contain `TerminalExpression` and may also contain another `NonterminalExpression`. It serves as the "grammar" of the specification language.
- **Context**: An information object used to perform the interpretation. This object is global to the interpreting process and is shared between nodes.

5 Implementation

1. **Define the expression classes:** Create a set of expression classes that represent the different elements of the mathematical expressions. This includes classes for numbers, addition, subtraction, multiplication, division, and any other operations you want to support. Each class should implement a common interface or inherit from a common base class that includes an `interpret()` method.
2. **Build the syntax tree:** Implement a parser or a combination of lexical analysis and parsing techniques to convert the mathematical expression into a syntax tree. The syntax tree represents the structure of the expression and its hierarchical relationships. Each node in the tree corresponds to an expression class from step 1.
3. **Implement interpretation logic:** Traverse the syntax tree and interpret each node by calling its `interpret()` method. The interpretation logic will vary based on the type of the expression. For example, in the `NumberExpression` class, the `interpret()` method would simply return the stored number value. In the `AdditionExpression` class, the `interpret()` method would evaluate the left and right expressions and return their sum.
4. **Evaluate the expression:** Once the interpretation of the syntax tree is complete, the final result can be obtained by calling the `interpret()` method on the root node of the tree. This will recursively evaluate the entire expression and return the final result.
5. **Test the implementation:** Create test cases with different mathematical expressions and verify that the Interpreter pattern correctly evaluates them. Ensure that the results are accurate and match the expected outcomes.

6 Advantages and disadvantages

- **Advantage**

- Reduce the dependency between abstraction and implementation (loose coupling).
- Reduce the number of unnecessary subclasses.
- The code will be cleaner and the application size will be smaller.
- Easier to maintain.
- Easy to expand later.
- Allows hiding implementation details from the client.

- **Disadvantages**
  - The constructed specification language requires a simple grammatical structure.
  - Performance is not guaranteed

7 Example

Problem: Suppose we have a simple language that consists of expressions in the form "number + number". We want to parse these expressions and evaluate the sum of the two numbers. Example expressions: "10 + 5", "7 + 3", "15 + 8", etc.

Solution

Step 1: Define the abstract syntax tree (AST) classes representing the expressions.

```
class Expression {
    public:
        virtual int evaluate() const = 0;
};

class NumberExpression : public Expression {
    private:
        int number;

    public:
        NumberExpression(int num) : number(num) {}

        int evaluate() const override {
            return number;
        }
};

class AdditionExpression : public Expression {
    private:
        Expression* left;
        Expression* right;

    public:
        AdditionExpression(Expression* lhs, Expression* rhs)
        : left(lhs), right(rhs) {}

        int evaluate() const override {
            return left->evaluate() + right->evaluate();
        }
};
```

Step 2: Implement a parser that converts the input string into an AST.

```
class Parser {
    public:
        Expression* parse(const std::string& input) {
            // Parse the input string and construct the AST
            // ...

            // For simplicity, let's assume the input format is always "number + number"
            int firstNumber = ...;  // Extract the first number from the input
            int secondNumber = ...; // Extract the second number from the input

            Expression* left = new NumberExpression(firstNumber);
            Expression* right = new NumberExpression(secondNumber);
            return new AdditionExpression(left, right);
        }
};
```

Step 3: Use the Interpreter pattern to evaluate the expressions.

```
int main() {
    Parser parser;

    std::string expression1 = "10 + 5";
    Expression* ast1 = parser.parse(expression1);
    int result1 = ast1->evaluate();
    std::cout << expression1 << " = " << result1 << std::endl;

    std::string expression2 = "7 + 3";
    Expression* ast2 = parser.parse(expression2);
    int result2 = ast2->evaluate();
    std::cout << expression2 << " = " << result2 << std::endl;

    // Clean up
    delete ast1;
    delete ast2;

    return 0;
}
```

3.3.2 Template Method

1 Definition

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but allows subclasses to override specific steps of the algorithm without altering its structure.

2 Problem

Imagine that you're creating a data mining application for analyzing corporate documents. Users input documents in various formats such as PDF, DOC, and CSV, and the application aims to extract meaningful data from these documents in a consistent format.

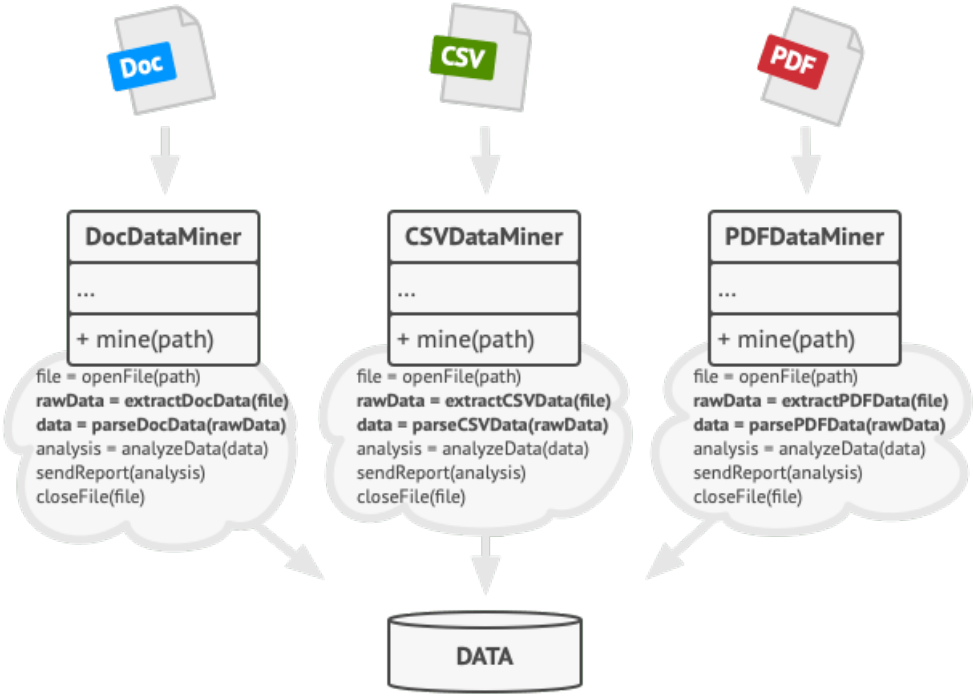


Figure 3.44: Data mining classes contained a lot of duplicate code.

Initially, the first version of the application could only handle DOC files. In the subsequent version, it gained the capability to support CSV files. After a month, it was further enhanced to extract data from PDF files.

During development, you observed that all three classes contained a significant amount of similar code. Although the code for handling different data formats varied completely across the classes, the code for data processing and analysis remained almost identical. It would be beneficial to eliminate this code duplication while preserving the algorithm structure.

Another challenge arose concerning the client code that utilized these classes. It contained numerous conditionals to determine the appropriate course of action based on the class of the processing object. If all three processing classes shared a common interface or inherited from a base class, the conditionals in the client code could be eliminated. Polymorphism could then be employed to invoke methods on a processing object.

3 Solution

The **Template Method** pattern suggests breaking down an algorithm into a series of steps, converting these steps into methods, and encapsulating them within a single template method. These steps can either be abstract or have default implementations. To utilize the algorithm, the client needs to provide its own

subclass, implement all the abstract steps, and optionally override some of the steps (excluding the template method itself) if necessary.

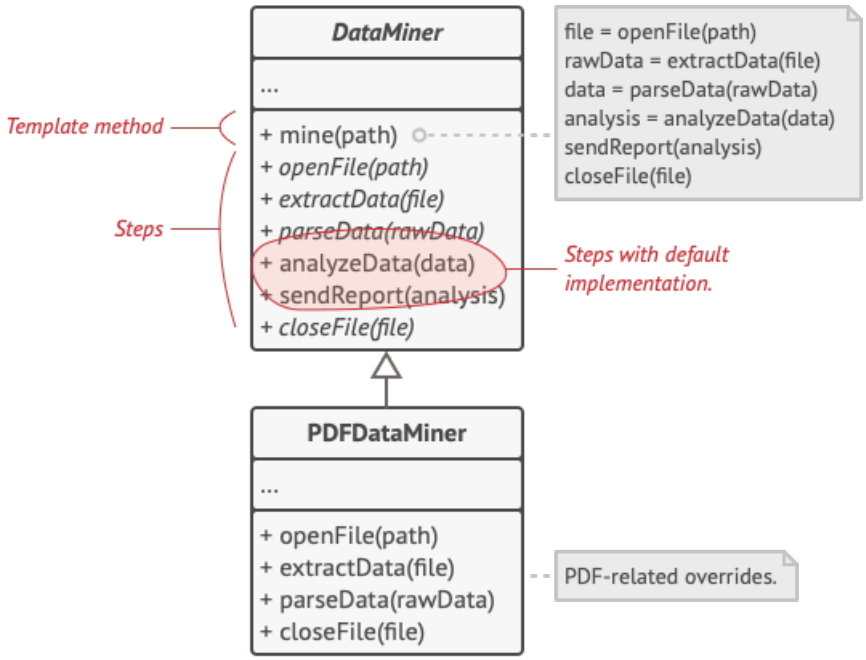


Figure 3.45: Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

Let's apply this pattern to our data mining application. We can create a base class for all three parsing algorithms. This class will define a template method that consists of a sequence of calls to various document processing steps.

Initially, we can declare all the steps as abstract, compelling the subclasses to provide their own implementations for these methods. In our case, the subclasses already have the necessary implementations, so we may only need to adjust the method signatures to match those of the superclass.

Now, let's focus on eliminating the duplicated code. It appears that the code responsible for opening/closing files and extracting/parsing data differs for each data format, so there is no need to modify those methods. However, the implementation of other steps, such as analyzing raw data and composing reports, is very similar. Hence, we can elevate this common code to the base class, enabling the subclasses to share it.

As you can observe, we have two types of steps:

- Abstract steps, which must be implemented by every subclass.
- Optional steps, which already have default implementations but can be overridden if needed.

Additionally, there is another type of step called hooks. A hook is an optional step with an empty body. The template method functions correctly even if a hook is not overridden. Hooks are often placed before and after crucial steps in the algorithm, providing subclasses with additional extension points.

4 Structure

The **Abstract Class** declares methods that serve as the individual steps of an algorithm, as well as the template method that orchestrates the execution of these steps in a specific order. These steps can either be declared as abstract methods or have default implementations.

The **Concrete Classes** that inherit from the abstract class have the ability to override all of the individual steps, but they cannot modify the template method itself. This ensures that the overall algorithm

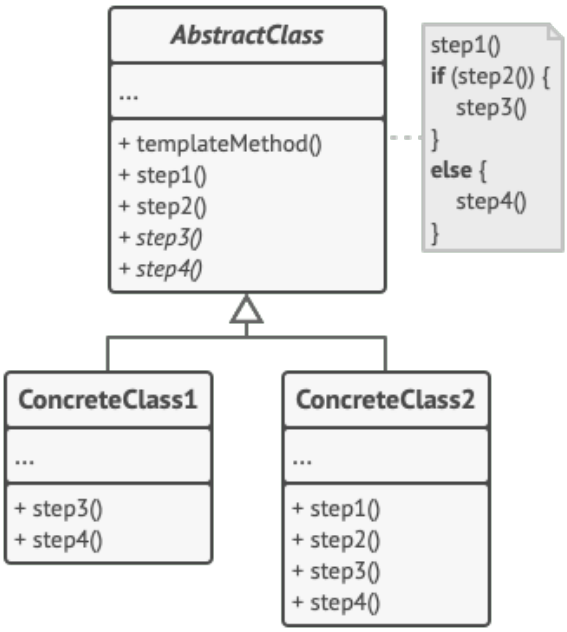


Figure 3.46: Structure of Template Method

structure remains intact while allowing the concrete classes to provide their own implementations for the specific steps as needed.

5 Implementation

1. To apply the Template Method pattern effectively, it is essential to analyze the target algorithm and determine its distinct steps. Identify the steps that are common to all subclasses and those that will always have unique implementations.

2. Next, create an abstract base class that declares the template method and a set of abstract methods representing the individual steps of the algorithm. Within the template method, outline the structure of the algorithm by sequentially executing the corresponding steps. Consider making the template method final to prevent subclasses from overriding it.

3. While it is acceptable for all steps to be abstract, some steps might benefit from having a default implementation. Subclasses are not obligated to implement these methods if they have suitable default behavior.

4. Additionally, consider incorporating hooks between crucial steps of the algorithm. Hooks provide optional extension points for subclasses to add their own logic.

5. For each variation of the algorithm, create a new concrete subclass. Each subclass must implement all of the abstract steps defined in the base class. Optionally, the concrete subclass can override some of the optional steps to customize its behavior.

6. By following these steps, you can effectively utilize the Template Method pattern to structure algorithms, facilitate code reuse, and allow for customization and extension through subclasses.

6 Advantages and disadvantages

Advantages: You have the option to allow clients to override specific sections of a complex algorithm, reducing the impact of changes made to other parts of the algorithm. By extracting duplicate code into a superclass, you can promote code reusability.

Disadvantages: Certain clients may be constrained by the predefined structure of the algorithm. It is important to note that suppressing a default step implementation through a subclass may lead to a violation of the Liskov Substitution Principle. Additionally, as the number of steps increases, template methods become more challenging to maintain.

7 Example

Suppose we have a mobile game that includes different levels. Each level has a similar structure, including initialization, gameplay, and finalization steps. However, the specific implementation of these steps may vary for each level. We want to provide a common framework that defines the overall structure of the levels while allowing individual levels to implement their own logic for each step.

Solution

Step 1: Define an abstract base class that represents the template for the level structure. This class will contain template methods representing the common steps of the level.

```
class Level {
    public:
        // Template method defining the level structure
        void play() {
            initialize();
            gameplay();
            finalize();
        }

        // Abstract methods to be implemented by concrete levels
        virtual void initialize() = 0;
        virtual void gameplay() = 0;
        virtual void finalize() = 0;
};
```

Step 2: Implement concrete classes that inherit from the base class and provide their own implementations for the abstract methods.

```
class EasyLevel : public Level {
    public:
        void initialize() override {
            // Initialize the easy level
        }

        void gameplay() override {
            // Implement gameplay logic for the easy level
        }

        void finalize() override {
            // Finalize the easy level
        }
};

class MediumLevel : public Level {
    public:
        void initialize() override {
```

```
            // Initialize the medium level
        }

        void gameplay() override {
            // Implement gameplay logic for the medium level
        }

        void finalize() override {
            // Finalize the medium level
        }
};

// Add more concrete level classes if needed
```

Step 3: Usage example.

```
int main() {
    // Create instances of the concrete level classes
    Level* easyLevel = new EasyLevel();
    Level* mediumLevel = new MediumLevel();

    // Play the levels
    easyLevel->play();
    mediumLevel->play();

    // Clean up
    delete easyLevel;
    delete mediumLevel;

    return 0;
}
```

3.3.3 Chain of Responsibility

1 Definition

The **Chain of Responsibility** is a design pattern with a focus on behavior. It enables you to pass requests through a series of handlers organized in a chain. When a request is received, each handler makes a decision to either handle the request or pass it along to the next handler in the chain.

2 Problem

Imagine you are working on an online ordering system that requires access restrictions. Only authenticated users should be able to create orders, while users with administrative permissions should have unrestricted access to all orders.

After careful planning, you realized that these checks need to be performed sequentially. The system should first attempt to authenticate a user based on the provided credentials. If authentication fails, there is no need to proceed with any further checks.

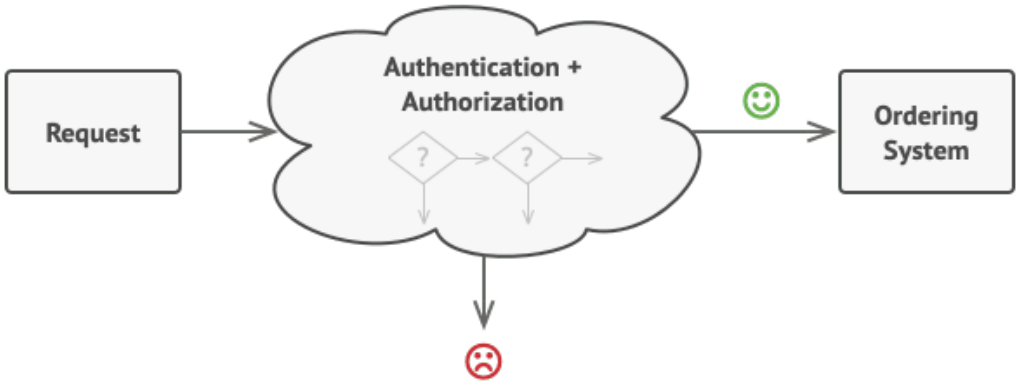


Figure 3.47: The request must pass a series of checks before the ordering system itself can handle it.

Over the course of a few months, you implemented additional sequential checks to enhance the system.

A colleague raised concerns about the security of passing raw data to the ordering system. To address this, you introduced an extra validation step to sanitize the request data.

Later, it was discovered that the system was susceptible to brute force password cracking. To mitigate this risk, you swiftly added a check to filter repeated failed requests originating from the same IP address.

Another suggestion was made to improve system performance by returning cached results for repeated requests with identical data. Consequently, you implemented another check to allow the request to proceed only if there is no suitable cached response.

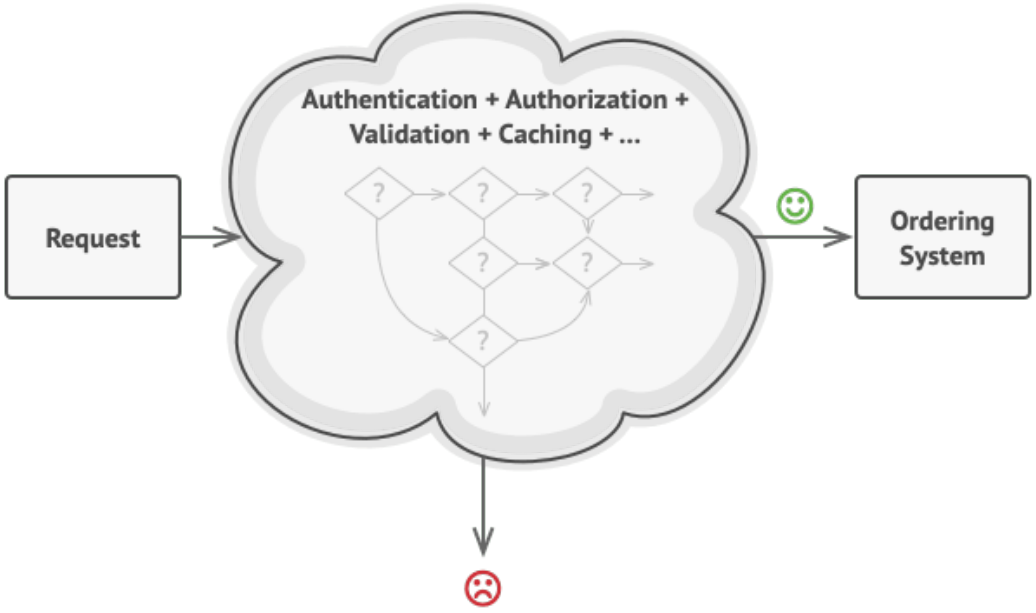


Figure 3.48: The bigger the code grew, the messier it became.

As more features were added, the code for these checks became increasingly convoluted. Modifying one check often affected others, and code duplication occurred when attempting to reuse the checks for other system components that required only some, but not all, of the checks.

The system became difficult to comprehend and costly to maintain. After struggling with the code for some time, you made the decision to refactor the entire system.

3 Solution

Similar to other behavioral design patterns, the Chain of Responsibility pattern leverages the concept of transforming specific behaviors into separate objects known as handlers. In our case, each check should be encapsulated within its own class, featuring a single method responsible for performing the check. The request, along with its data, is passed as an argument to this method.

The pattern suggests establishing a chain by linking these handlers together. Each handler within the chain maintains a reference to the next handler. In addition to processing the request, handlers pass it along the chain, allowing multiple handlers to have the opportunity to process it.

The fascinating aspect is that a handler can decide to halt the further propagation of the request down the chain, effectively stopping any additional processing.

In our example of an ordering system, a handler performs its processing and then determines whether to pass the request down the chain. Assuming the request contains the appropriate data, all the handlers can execute their respective main behaviors, such as authentication checks or caching.

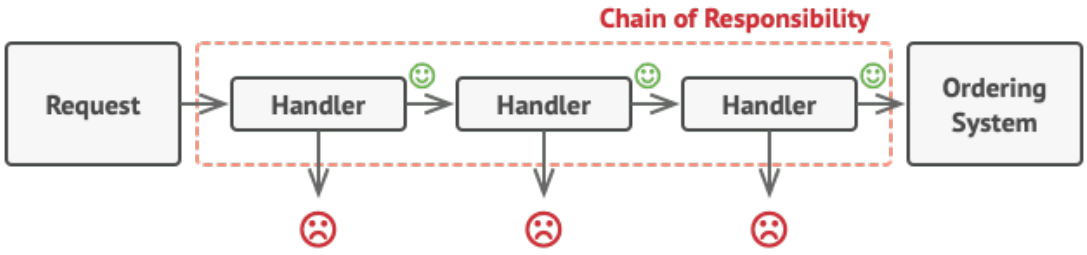


Figure 3.49: Handlers are lined up one by one, forming a chain.

However, there is a slightly different approach, which is more commonly used and considered canonical.

In this approach, upon receiving a request, a handler determines whether it can process the request. If it can, it handles the request and does not pass it further down the chain. Therefore, either only one handler processes the request or none at all. This approach is frequently employed when dealing with event handling in stacks of elements within a graphical user interface.

For example, when a user clicks a button, the event propagates through a chain of GUI elements, starting with the button, traversing its containers (e.g., forms or panels), and finally reaching the main application window. The event is processed by the first element in the chain capable of handling it. This example is noteworthy as it demonstrates how a chain can be derived from an object tree.

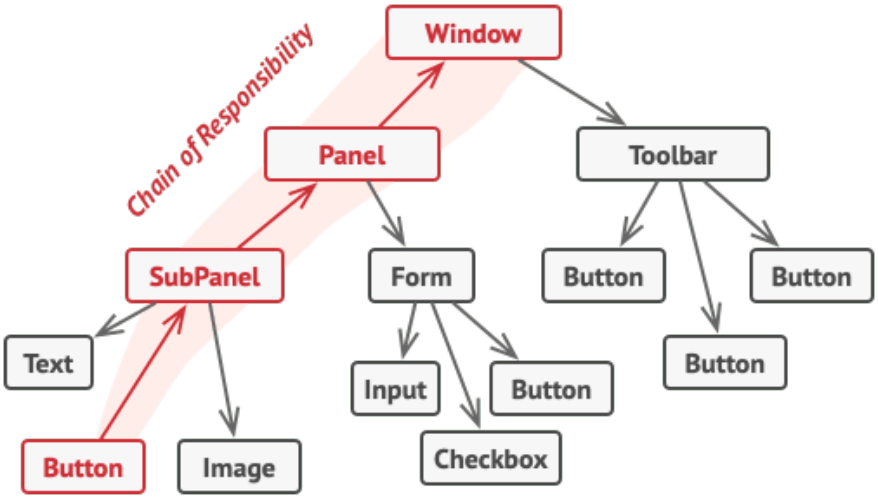


Figure 3.50: A chain can be formed from a branch of an object tree.

It is crucial that all handler classes adhere to the same interface. Each concrete handler should only be concerned with the presence of the execute method in the following handler. This way, you can dynamically compose chains at runtime, utilizing various handlers without coupling your code to their specific classes.

4 Structure

The `Handler` interface defines the common interface for all concrete handlers. Typically, it includes a single method for handling requests, although it may occasionally have an additional method for setting the next handler in the chain.

The `Base Handler` is an optional class where you can place the boilerplate code shared among all handler classes. This class often defines a field to store a reference to the next handler. `Clients` can construct a chain by passing a handler to the constructor or setter of the previous handler. Additionally, the class may implement the default handling behavior by passing the execution to the next handler after checking its existence.

`Concrete Handlers` contain the actual implementation code for processing requests. When a handler receives a request, it must determine whether to process it and whether to pass it along the chain.

`Handlers` are typically self-contained and immutable, accepting all necessary data through the constructor.

The `Client` has the ability to compose chains either once or dynamically, depending on the application's logic. It's important to note that a request can be sent to any handler within the chain and is not limited to the first handler.

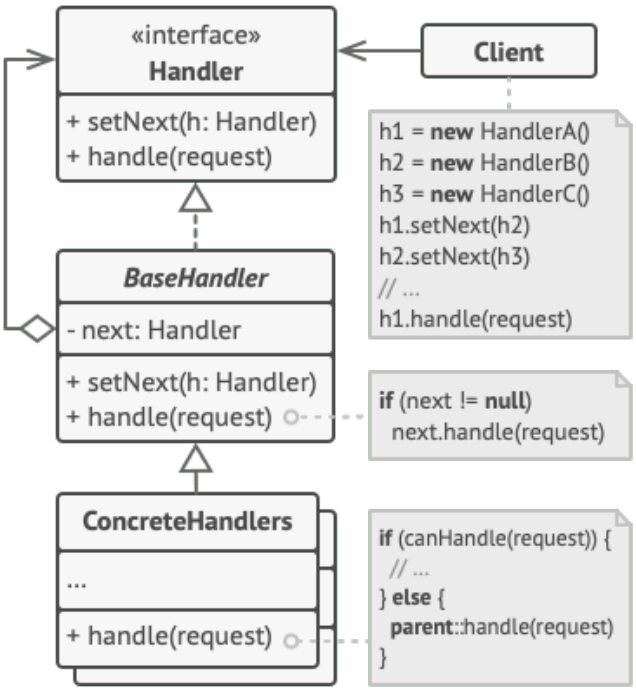


Figure 3.51: Structure of Chain of Responsibility

5 Implementation

1. Declare the handler interface and describe the signature of a method for handling requests.
2. Decide how the client will pass the request data into the method. The most flexible way is to convert the request into an object and pass it to the handling method as an argument.
3. To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.
4. This class should have a field for storing a reference to the next handler in the chain. Consider making the class immutable. However, if you plan to modify chains at runtime, you need to define a setter for altering the value of the reference field.
5. You can also implement the convenient default behavior for the handling method, which is to forward the request to the next object unless there's none left. Concrete handlers will be able to use this behavior by calling the parent method.
6. One by one, create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:
   - Whether it'll process the request.
   - Whether it'll pass the request along the chain.
7. The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.
8. The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.
9. Due to the dynamic nature of the chain, the client should be ready to handle the following scenarios:

- The chain may consist of a single link.
- Some requests may not reach the end of the chain.
- Others may reach the end of the chain unhandled.

6 Advantages and disadvantages

Advantages:

- You can control the order of request handling.
- Single Responsibility Principle: You can decouple classes that invoke operations from classes that perform operations.
- Open/Closed Principle: You can introduce new handlers into the app without breaking the existing client code.

Disadvantages: Some requests may end up unhandled.

7 Example

Suppose we have a system that processes different types of customer support requests. Each request belongs to a specific category such as technical support, billing inquiries, or general inquiries. We want to design a system where each request is handled by the appropriate support team based on its category. However, we want to avoid coupling the client code with specific support teams and allow the requests to be dynamically routed based on their category.

Solution:
Step 1: Define the abstract handler class and its concrete implementations.

```
// Step 1: Define the abstract handler class and its concrete implementations
class SupportHandler {
    protected:
        SupportHandler* nextHandler;

    public:
        SupportHandler() : nextHandler(nullptr) {}
        void setNextHandler(SupportHandler* handler) {
            nextHandler = handler;
        }

        virtual void handleRequest(const std::string& request) = 0;
};

class TechnicalSupportHandler : public SupportHandler {
    public:
        void handleRequest(const std::string& request) override {
            if (request == "technical") {
                // Handle technical support request
            } else if (nextHandler != nullptr) {
                nextHandler->handleRequest(request);
            }
        }
};

```

```
class BillingHandler : public SupportHandler {
    public:
        void handleRequest(const std::string& request) override {
            if (request == "billing") {
                // Handle billing inquiry
            }

            else if (nextHandler != nullptr) {
                nextHandler->handleRequest(request);
            }
        }
};

class GeneralInquiryHandler : public SupportHandler {
    public:
        void handleRequest(const std::string& request) override {
            if (request == "general") {
                // Handle general inquiry
            }

            else if (nextHandler != nullptr) {
                nextHandler->handleRequest(request);
            }
        }
};
```

Step 2: Configure the chain of responsibility.

```
// Step 2: Configure the chain of responsibility
int main() {
    SupportHandler* technicalSupportHandler = new TechnicalSupportHandler();
    SupportHandler* billingHandler = new BillingHandler();
    SupportHandler* generalInquiryHandler = new GeneralInquiryHandler();

    technicalSupportHandler->setNextHandler(billingHandler);
    billingHandler->setNextHandler(generalInquiryHandler);

    // Usage example
    technicalSupportHandler->handleRequest("technical");
    technicalSupportHandler->handleRequest("billing");
    technicalSupportHandler->handleRequest("general");

    // Clean up
    delete technicalSupportHandler;
    delete billingHandler;
    delete generalInquiryHandler;

    return 0;
}
```

3.3.4 Command

1 Definition

The Command pattern is a behavioral design pattern that encapsulates a request as a separate object, containing all the necessary information about the request. This transformation allows requests to be passed as method arguments, enables the delayed or queued execution of requests, and provides support for undoable operations.

2 Problem

Imagine that you're working on a new text-editor app, and your task is to create a toolbar with multiple buttons for different operations of the editor. Initially, you create a Button class that can be used for buttons on the toolbar and in various dialogs, as they all have a similar appearance.

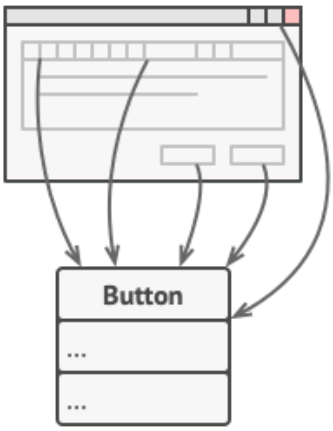


Figure 3.52: All buttons of the app are derived from the same class.

However, each of these buttons is expected to perform a different action when clicked. The question arises: where should you place the code for the various click handlers of these buttons? The simplest approach would be to create numerous subclasses for each usage of the button, with each subclass containing the code to be executed upon button click.

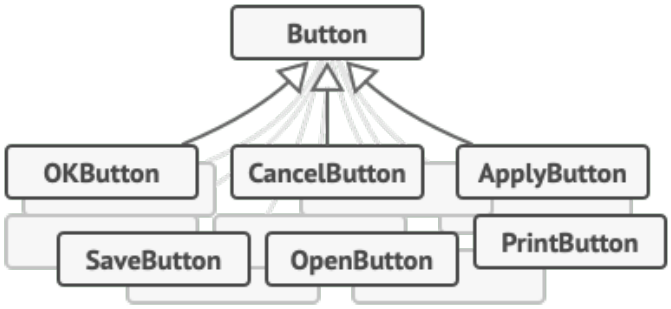


Figure 3.53: Lots of button subclasses. What can go wrong?

Unfortunately, this approach quickly proves to be flawed. Firstly, it leads to a proliferation of subclasses, which would be manageable if modifying the base Button class didn't risk breaking the code in these subclasses. In essence, your GUI code becomes tightly coupled with the volatile business logic code.

Moreover, certain operations, such as copying and pasting text, need to be invoked from multiple sources. For instance, a user might click a "Copy" button on the toolbar, use the context menu, or press Ctrl+C on the keyboard.

Initially, when the app only had the toolbar, it might have seemed acceptable to place the implementation of various operations within the respective button subclasses. In other words, having the code for

Figure 3.54: Several classes implement the same functionality.

copying text inside the CopyButton subclass appeared reasonable. However, as you start implementing context menus, keyboard shortcuts, and other functionalities, you are faced with the dilemma of either duplicating the operation's code in multiple classes or making menus dependent on buttons, which is an even worse solution.

3 Solution

Good software design often follows the principle of separation of concerns, which involves dividing an application into layers. A common example is separating the graphical user interface (GUI) layer from the business logic layer. The GUI layer is responsible for displaying an aesthetically pleasing interface, capturing user input, and presenting the results of user and application actions. However, for important tasks like calculating complex trajectories or generating annual reports, the GUI layer delegates the work to the underlying business logic layer.

In code, this separation can be achieved by having a GUI object call a method of a business logic object, passing relevant arguments. This process is typically described as one object sending a request to another.

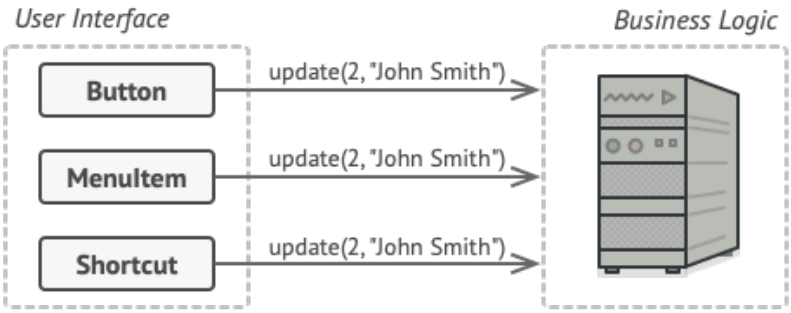


Figure 3.55: The GUI objects may access the business logic objects directly.

The Command pattern suggests that GUI objects should not directly send these requests. Instead, the request details, such as the object being called, the method name, and the list of arguments, should be extracted into a separate command class. This command class should have a single method that triggers the request.

Command objects act as connectors between GUI and business logic objects. Now, the GUI object does not need to know which business logic object will receive the request or how it will be processed. The GUI object simply triggers the command, and the command handles all the necessary details.

The next step is to ensure that all commands implement the same interface. This interface typically includes a single execution method that takes no parameters. By using this common interface, you can use various commands with the same request sender without tightly coupling it to specific command classes. Additionally, it allows you to switch command objects dynamically, effectively changing the behavior of the sender at runtime.

One missing piece of the puzzle is how to pass request parameters. While the command execution method does not have any parameters, the command should be either pre-configured with the necessary data or capable of obtaining it on its own.

Let's return to the example of a text editor. After applying the Command pattern, there is no need for

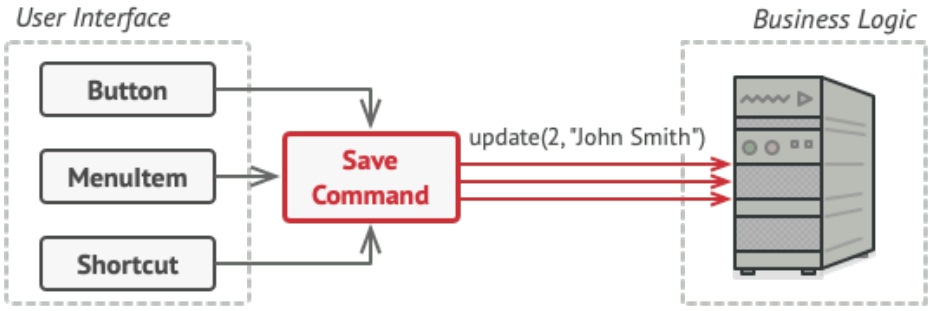


Figure 3.56: Accessing the business logic layer via a command.

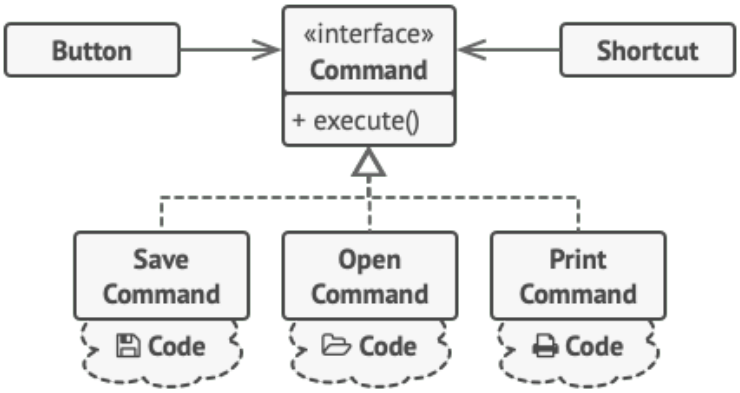


Figure 3.57: The GUI objects delegate the work to commands.

numerous subclasses of buttons to implement different click behaviors. Instead, a single field is added to the base Button class, which stores a reference to a command object. The button executes the command on a click event.

You will implement multiple command classes, each representing a specific operation, and link them with corresponding buttons based on the desired behavior of the buttons.

Other GUI elements, such as menus, shortcuts, or entire dialogs, can be implemented in the same manner. They will be linked to commands that are executed when users interact with the GUI elements. As a result, elements related to the same operations will be linked to the same commands, eliminating code duplication.

In conclusion, the Command pattern acts as a convenient intermediary layer that reduces coupling between the GUI and business logic layers. And these benefits are just a fraction of what the Command pattern can offer!

4 Structure

1. **Sender (Invoker):** The Sender class is responsible for initiating requests. It contains a field to store a reference to a command object. Instead of directly sending the request to the receiver, the sender triggers the command object. The sender does not create the command object itself but receives it from the client, typically through the constructor.

2. **Command:** The Command interface declares a single method for executing the command. This method represents the action that needs to be performed. Concrete command classes will implement this interface and provide the specific implementation for executing different kinds of requests. The command objects are responsible for passing the call to the appropriate business logic objects.

3. **Concrete Commands:** Concrete command classes implement specific requests by extending the Command interface. They define the execution method and may contain additional fields for the

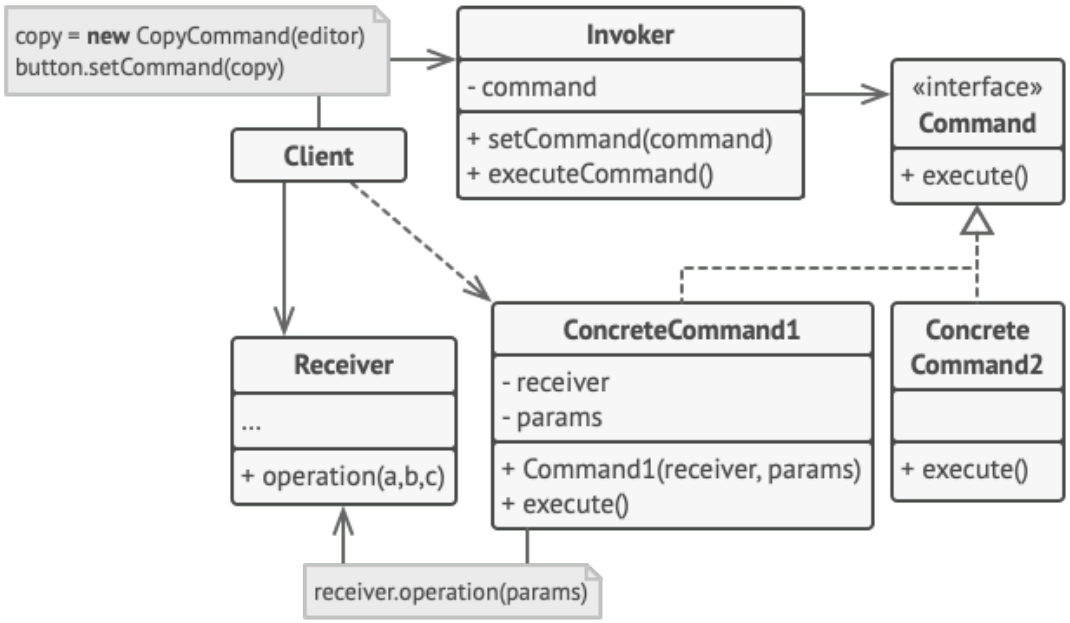


Figure 3.58: Structure of Command.

parameters required to execute the method on the receiver. These command objects can be made immutable by initializing their fields through the constructor.

4. **Receiver:** The Receiver class contains the actual business logic and performs the work requested by the command. Any object can act as a receiver, and most commands handle the details of how the request is passed to the receiver. The receiver itself executes the desired operations.

5. **Client:** The Client creates and configures concrete command objects. The client is responsible for providing all the necessary request parameters, including a receiver instance, through the command's constructor. After creating the command, the client can associate it with one or multiple senders to trigger the execution of the command when needed.

5 Implementation

1. Declare the command interface with a single execution method.

2. Start extracting requests into concrete command classes that implement the command interface. Each class must have a set of fields for storing the request arguments along with a reference to the actual receiver object. All these values must be initialized via the command's constructor.

3. Identify classes that will act as senders. Add the fields for storing commands into these classes. Senders should communicate with their commands only via the command interface. Senders usually don't create command objects on their own, but rather get them from the client code.

4. Change the senders so they execute the command instead of sending a request to the receiver directly.

5. The client should initialize objects in the following order:

   - Create receivers.
   - Create commands and associate them with receivers if needed.
   - Create senders and associate them with specific commands.

6 Advantages and disadvantages

Advantages:

- Single Responsibility Principle: The Command pattern allows for decoupling of classes that invoke operations from classes that perform these operations.
- Open/Closed Principle: New commands can be introduced into the application without breaking existing client code.
- Undo/Redo: The Command pattern facilitates the implementation of undo and redo operations.
- Deferred Execution: Operations can be executed at a later time, allowing for the implementation of deferred execution of commands.
- Complex Commands: Simple commands can be assembled into complex ones, enabling the creation of more sophisticated operations.

Disadvantages:

- Increased Complexity: The Command pattern introduces an additional layer between senders and receivers, which can make the code more complicated.

7 Example

We have a text editor application that needs to support various operations such as opening a file, saving a file, copying text, and pasting text. We want to implement an undo feature that can revert the executed operations in a specific order.

Solution
Step 1: Define the Command interface and its implementations.

```
// Step 1: Define the Command interface and its implementations
class Command {
    public:
        virtual ~Command() {}
        virtual void execute() = 0;
        virtual void undo() = 0;
};

class OpenFileCommand : public Command {
    public:
        OpenFileCommand(/* parameters if needed */) {}
        void execute() override {
            // Open file implementation
        }

        void undo() override {
            // Undo open file implementation
        }
};

class SaveFileCommand : public Command {
    public:
        SaveFileCommand(/* parameters if needed */) {}
        void execute() override {
            // Save file implementation
```

```
        }

        void undo() override {
            // Undo save file implementation
        }
};

class CopyTextCommand : public Command {
    public:
        CopyTextCommand(/* parameters if needed */) {}
        void execute() override {
            // Copy text implementation
        }

        void undo() override {
            // Undo copy text implementation
        }
};

class PasteTextCommand : public Command {
    public:
        PasteTextCommand(/* parameters if needed */) {}
        void execute() override {
            // Paste text implementation
        }

        void undo() override {
            // Undo paste text implementation
        }
};

// Add more command classes if needed
```

Step 2: Implement the Invoker class and the Client code.

```
// Step 2: Implement the Invoker class and the Client code
class Invoker {
    private:
        std::vector<Command*> commands;

    public:
        void executeCommand(Command* command) {
            command->execute();
            commands.push_back(command);
        }

        void undoLastCommand() {
            if (!commands.empty()) {
                Command* lastCommand = commands.back();
                lastCommand->undo();
                commands.pop_back();
            }
    }
};

int main() {
```

```
22      Invoker invoker;
23
24      // Create commands
25      Command* openFileCommand = new OpenFileCommand(/* parameters if ↩
            needed */);
26      Command* saveFileCommand = new SaveFileCommand(/* parameters if ↩
            needed */);
27      Command* copyTextCommand = new CopyTextCommand(/* parameters if ↩
            needed */);
28      Command* pasteTextCommand = new PasteTextCommand(/* parameters if ↩
            needed */);
29
30      // Execute commands
31      invoker.executeCommand(openFileCommand);
32      invoker.executeCommand(copyTextCommand);
33      invoker.executeCommand(saveFileCommand);
34      invoker.executeCommand(pasteTextCommand);
35
36      // Undo the last command
37      invoker.undoLastCommand();
38
39      // Clean up
40      delete openFileCommand;
41      delete saveFileCommand;
42      delete copyTextCommand;
43      delete pasteTextCommand;
44
45      return 0;
46  }
```

3.3.5 Iterator

1 Definition

The Iterator pattern is a behavioral design pattern that provides a way to iterate over elements of a collection without exposing its internal structure, such as a list, stack, tree, or other data structures. It separates the process of traversing the collection from the specific implementation of the collection, making it easier to change or extend the iteration logic without affecting the client code. By encapsulating the iteration logic within an iterator object, the pattern promotes a more flexible and modular design.

2 Problem

Collections are widely used in programming to store and manage groups of objects. However, collections come in various forms, such as lists, stacks, trees, and graphs, each with their own internal structure.

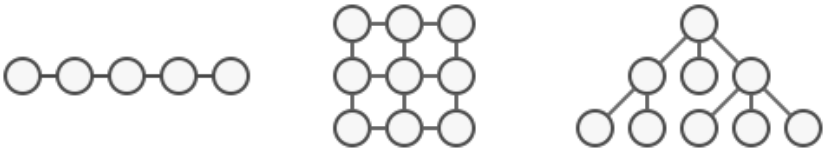


Figure 3.59: Various types of collections.

Regardless of the specific implementation, a collection must provide a way to access its elements so that other code can utilize them. The ability to iterate through the elements of a collection without repetitive access is crucial.

Iterating over elements in a simple list-based collection is relatively straightforward, as you can loop through the elements sequentially. However, when dealing with complex data structures like trees, traversing the elements becomes more challenging. Different traversal algorithms, such as depth-first or breadth-first, may be required depending on the situation. Moreover, there might be a need for random access to the elements.

Adding multiple traversal algorithms to the collection itself can lead to a loss of clarity in its primary responsibility, which is efficient data storage. Additionally, specific algorithms may be tailored for particular applications, making their inclusion in a generic collection class inappropriate.

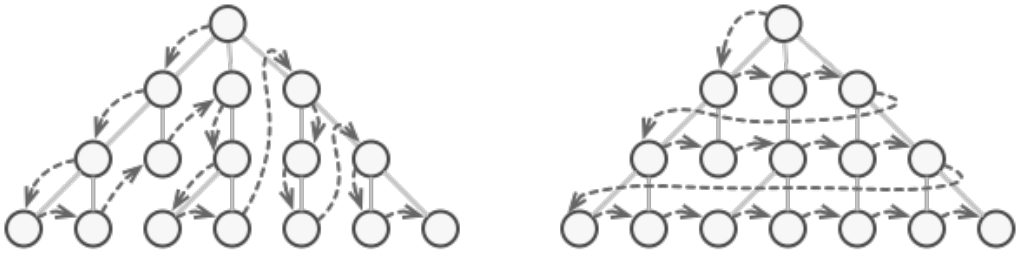


Figure 3.60: Various types of collections.

On the other hand, client code that interacts with various collections may not be concerned with the specific storage mechanism used by each collection. Unfortunately, due to the different ways collections provide access to their elements, the client code often ends up tightly coupled to the specific collection classes.

3 Solution

The Iterator pattern revolves around the concept of separating the traversal behavior of a collection into a distinct object known as an iterator.

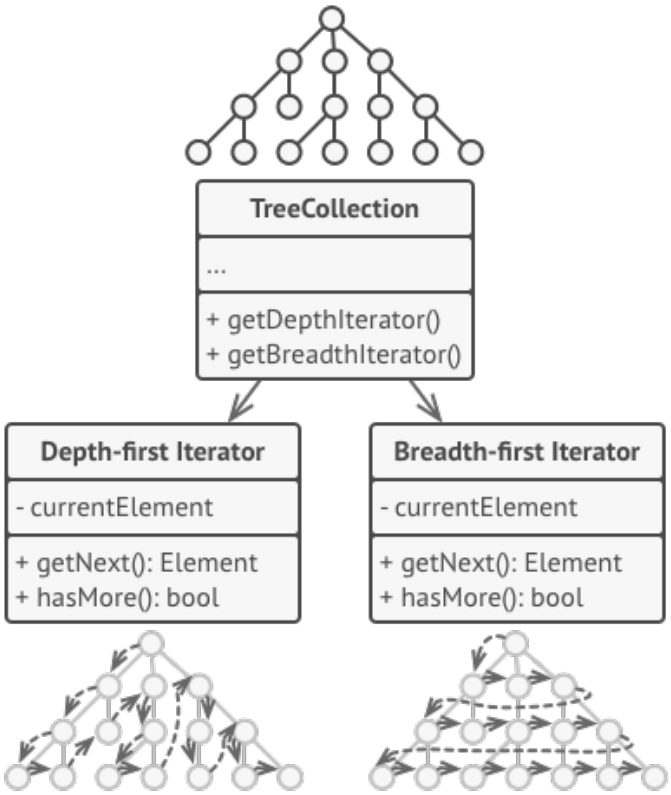


Figure 3.61: Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

The iterator object not only implements the traversal algorithm but also encapsulates the specific details of traversal, such as the current position and the number of remaining elements. As a result, multiple iterators can concurrently traverse the same collection independently.

Typically, iterators provide a single primary method for retrieving elements from the collection. The client code can repeatedly invoke this method until it returns nothing, indicating that the iterator has reached the end of the collection.

All iterators adhere to the same interface, ensuring compatibility with any collection type or traversal algorithm, as long as a suitable iterator is available. Introducing a new way to traverse a collection is as simple as creating a new iterator class, without requiring modifications to the collection or the client code.

4 Structure

- The **Iterator** interface declares the operations required for traversing a collection, such as fetching the next element, retrieving the current position, and restarting iteration.
- **Concrete Iterators** implement specific algorithms for traversing a collection. Each iterator object tracks its own traversal progress, allowing multiple iterators to traverse the same collection independently.
- The **Collection** interface declares one or more methods for obtaining iterators compatible with the collection. These methods should return the iterator interface as the return type, enabling concrete collections to provide different types of iterators.
- **Concrete Collections** return new instances of a specific concrete iterator class whenever a client requests an iterator. The collection class also contains the remaining implementation details, although they are not essential to the pattern and are omitted here.

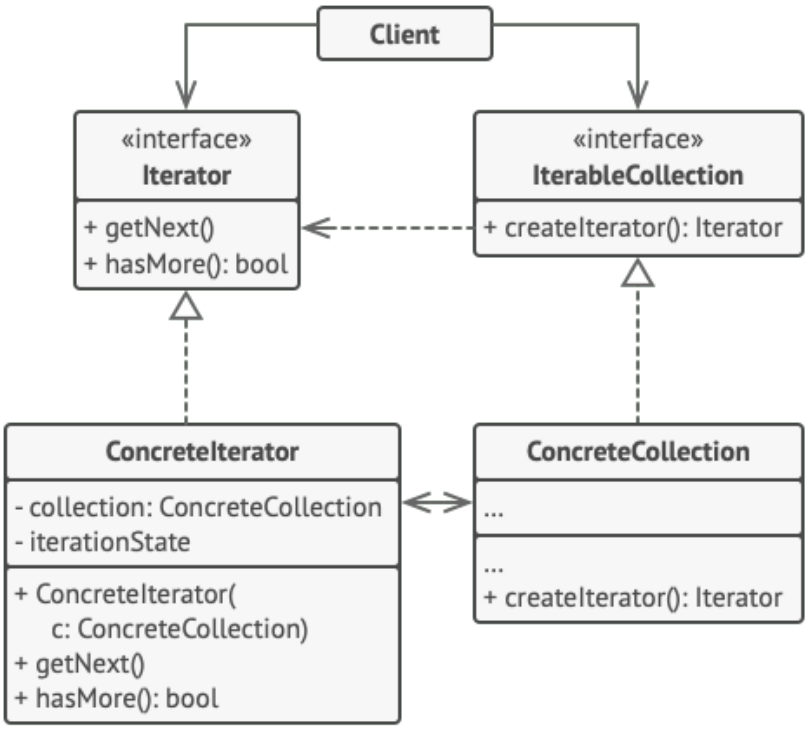


Figure 3.62: Structure of Iterator

- The **Client** interacts with both collections and iterators through their interfaces. This decouples the client code from specific classes, allowing it to work with various collections and iterators interchangeably.
- Typically, clients don't create iterators directly but instead obtain them from collections. However, in certain cases, the client can create iterators directly, such as when defining a specialized iterator.

5 Implementation

1. Declare the **iterator interface**. This interface should include a method for fetching the next element from a collection. Additional methods can be added for convenience, such as fetching the previous element, tracking the current position, and checking the end of the iteration.
2. Declare the **collection interface** and define a method for fetching iterators. The return type of this method should match the iterator interface. You can declare similar methods if you need distinct groups of iterators for different purposes.
3. Implement **concrete iterator classes** for the collections that you want to make traversable with iterators. Each iterator object should be associated with a specific collection instance, typically through its constructor.
4. Implement the **collection interface** in your collection classes. The goal is to provide a convenient way for the client to create iterators that are tailored to the specific collection class. The collection object should pass itself to the iterator's constructor to establish the necessary link between them.
5. Review the client code and replace any existing collection traversal code with the use of iterators. Instead of directly accessing collection elements, the client should fetch a new iterator object whenever it needs to iterate over the elements of the collection.

6 Advantages and disadvantages

Advantages:

- Single Responsibility Principle: The Iterator pattern allows for cleaner code by separating traversal algorithms from the client code and the collections. It promotes a more modular and maintainable design.
- Open/Closed Principle: You can introduce new types of collections and iterators without modifying the existing client code. This makes the code more flexible and extensible.
- Independent Iteration: Each iterator object maintains its own iteration state, allowing multiple iterators to traverse the same collection independently. This enables parallel iteration and provides flexibility in managing iteration progress.
- Delayed Iteration: Iteration can be paused and resumed at any point, allowing for more control and dynamic iteration based on application logic.

Disadvantages:

- Overkill for Simple Collections: Implementing the Iterator pattern may be unnecessary for simple collections that can be easily traversed without the need for a separate iterator object. Using the pattern in such cases can introduce unnecessary complexity.
- Potential Performance Impact: Using an iterator may introduce some overhead compared to directly accessing elements of specialized collections. In performance-critical scenarios, direct access may be more efficient.

7 Example

Suppose we have a collection of books, and we want to iterate over the books to perform certain operations. However, we want to decouple the iteration logic from the actual collection implementation to ensure flexibility and extensibility.

Solution Step 1: Define the Iterator interface and its implementation.

```
// Step 1: Define the Iterator interface and its implementation
class Iterator {
    public:
        virtual bool hasNext() const = 0;
        virtual std::string next() = 0;
};

class BookIterator : public Iterator {
    private:
        std::vectorstd::string books;
        int current = 0;

    public:
        BookIterator(const std::vectorstd::string& books) : books(books) {}

        bool hasNext() const override {
            return current < books.size();
        }

        std::string next() override {
            if (!hasNext()) {
```

```
                throw std::out_of_range("No more books in the collection.");
            }
            return books[current++];
        }
};
```

Step 2: Define the collection class and its methods.

```
// Step 2: Define the collection class and its methods
class BookCollection {
    private:
        std::vectorstd::string books;

    public:
        void addBook(const std::string& book) {
            books.push_back(book);
        }

        Iterator* createIterator() const {
            return new BookIterator(books);
        }
};
```

Step 3: Usage example.

```
int main() {
    BookCollection collection;
    collection.addBook("Book 1");
    collection.addBook("Book 2");
    collection.addBook("Book 3");
    Iterator* iterator = collection.createIterator();

    while (iterator->hasNext()) {
        std::string book = iterator->next();
        std::cout << "Book: " << book << std::endl;
    }

    delete iterator;

    return 0;
}
```

3.3.6 Mediator

1 Definition

The **Mediator** is a behavioral design pattern that aims to minimize complex dependencies among objects. By employing this pattern, direct communication between objects is restricted, and instead, they are compelled to cooperate solely through a mediator object.

2 Problem

In the scenario of a dialog for creating and editing customer profiles, various form controls like text fields, checkboxes, and buttons are utilized. Certain form elements may have interactions with others, such as the selection of the "I have a dog" checkbox revealing a hidden text field for entering the dog's name. Similarly, the submit button is responsible for validating the values of all fields before saving the data.

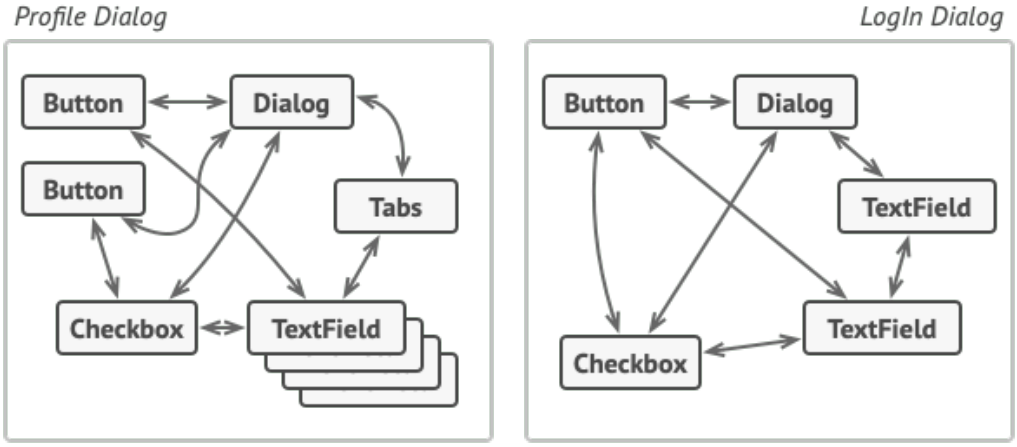


Figure 3.63: Relations between elements of the user interface can become chaotic as the application evolves.

Implementing this logic directly within the code of the form elements increases the complexity of reusing these elements' classes in other forms within the application. For instance, the checkbox class coupled with the dog's text field cannot be easily utilized in another form. The classes involved in rendering the profile form must be used together or not at all.

Figure 3.64: Elements can have lots of relations with other elements. Hence, changes to some elements may affect the others.

3 Solution

The Mediator pattern proposes the cessation of direct communication between components that need to be independent of each other. Instead, these components collaborate indirectly by invoking a dedicated mediator object, which redirects the calls to the appropriate components. As a result, the components rely on a single mediator class rather than being tightly coupled to numerous colleagues.

In the context of our profile editing form example, the dialog class can serve as the mediator. Since the dialog class is likely already aware of its sub-elements, introducing new dependencies into this class may not be necessary.

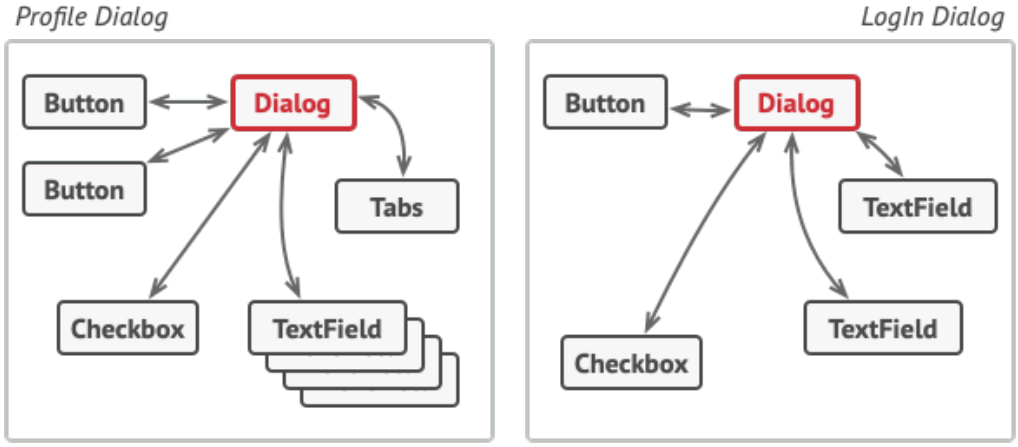


Figure 3.65: UI elements should communicate indirectly, via the mediator object.

The most significant transformation occurs in the actual form elements. Let's consider the submit button. Previously, whenever a user clicked the button, it had to validate the values of all individual form elements. Now, its sole responsibility is to notify the dialog about the click event. Upon receiving this notification, the dialog itself performs the validations or delegates the task to the individual elements. Consequently, the button only depends on the dialog class, rather than being tied to multiple form elements.

To further loosen the dependency, it is possible to extract a common interface for all types of dialogs. This interface would declare a notification method that all form elements can utilize to inform the dialog about events occurring within those elements. Hence, our submit button should be able to function with any dialog implementing this interface.

By employing the Mediator pattern, a complex network of relationships between various objects can be encapsulated within a single mediator object. The fewer dependencies a class has, the easier it becomes to modify, extend, or reuse that class.

4 Structure

Components refer to various classes that encapsulate business logic. Each component holds a reference to a mediator, declared with the mediator interface type. The component remains unaware of the concrete mediator class, enabling its reuse in different programs by connecting it to a different mediator.

The Mediator interface defines communication methods with components, typically including a single notification method. Components can pass any relevant context as arguments to this method, including their own objects. However, it is crucial to ensure that no coupling occurs between the receiving component and the sender's class.

Components should not have knowledge of other components. If a significant event occurs within or to a component, it should only notify the mediator. Upon receiving the notification, the mediator can easily identify the sender, which is often sufficient to determine which component should be triggered in response.

From the perspective of a component, the process appears as a complete black box. The sender remains unaware of who will handle its request, and the receiver is unaware of the sender's identity.

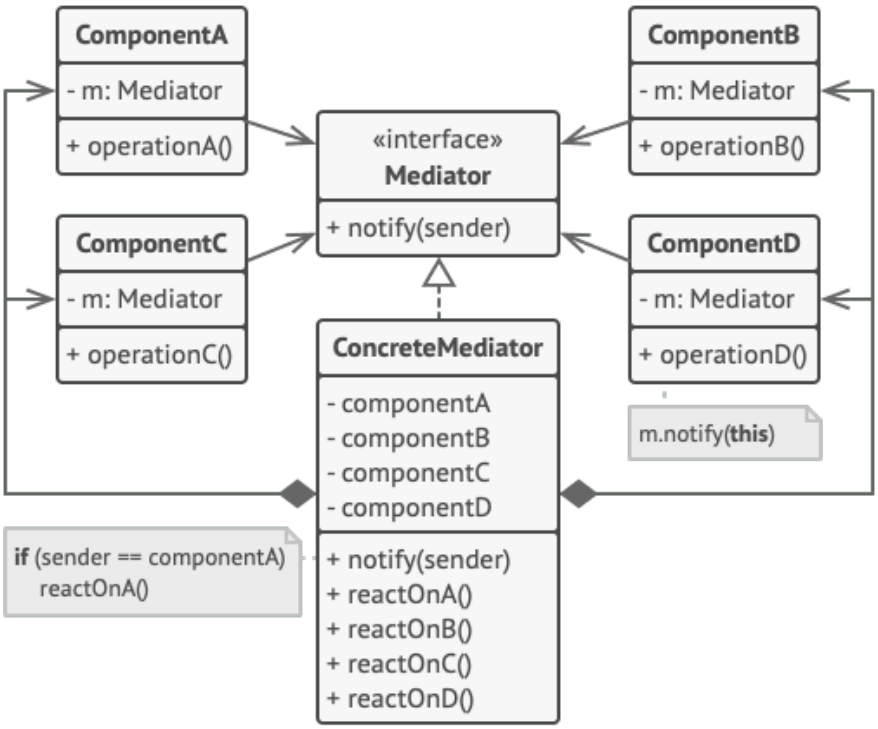


Figure 3.66: Structure of Mediator

5 Implementation

1. Identify a group of tightly coupled classes which would benefit from being more independent (e.g., for easier maintenance or simpler reuse of these classes).

2. Declare the mediator interface and describe the desired communication protocol between mediators and various components. In most cases, a single method for receiving notifications from components is sufficient.

3. This interface is crucial when you want to reuse component classes in different contexts. As long as the component works with its mediator via the generic interface, you can link the component with a different implementation of the mediator.

4. Implement the concrete mediator class. Consider storing references to all components inside the mediator. This way, you could call any component from the mediator's methods.

5. You can go even further and make the mediator responsible for the creation and destruction of component objects. After this, the mediator may resemble a factory or a facade.

6. Components should store a reference to the mediator object. The connection is usually established in the component's constructor, where a mediator object is passed as an argument.

7. Change the components' code so that they call the mediator's notification method instead of methods on other components. Extract the code that involves calling other components into the mediator class. Execute this code whenever the mediator receives notifications from that component.

6 Advantages and disadvantages

Advantages:

- **Single Responsibility Principle:** By consolidating the communications between components in one location, it becomes easier to understand and maintain.

- **Open/Closed Principle:** The introduction of new mediators does not require modifications to the existing components.
- **Reduced Coupling:** The Mediator pattern helps to minimize the coupling between various components in a program.
- **Improved Reusability:** Individual components can be more easily reused in different contexts.

Disadvantages:

- **Potential God Object:** Over time, a mediator can grow and become overly complex, taking on the characteristics of a God Object that handles too many responsibilities.

7 Example

Imagine a chat application where multiple users can send messages to each other. However, each user should be able to send messages to specific users only, rather than broadcasting to all users. We want to implement a solution that allows users to send messages to specific recipients without directly knowing about each other.

Solution:
Step 1: Define the Mediator and Colleague classes

```cpp
#include <iostream>
#include <string>
#include <vector>

class Colleague;

// Step 1: Define the Mediator class
class Mediator {
public:
    virtual void sendMessage(const std::string& message, Colleague* sender, Colleague* recipient) = 0;
};

// Step 1: Define the Colleague class
class Colleague {
protected:
    Mediator* mediator;
    std::string name;

public:
    Colleague(const std::string& name, Mediator* mediator) : name(name), mediator(mediator) {}

    virtual void send(const std::string& message, Colleague* recipient) = 0;
    virtual void receive(const std::string& message) = 0;

    const std::string& getName() const {
        return name;
    }
};
```

Step 2: Implement the Concrete Mediator class

```cpp
// Step 2: Implement the Concrete Mediator class
class Chatroom : public Mediator {
private:
    std::vector<Colleague*> colleagues;

public:
    void addColleague(Colleague* colleague) {
        colleagues.push_back(colleague);
    }

    void sendMessage(const std::string& message, Colleague* sender, Colleague* recipient) override {
        for (Colleague* colleague : colleagues) {
            if (colleague == recipient)
                colleague->receive(message);
        }
    }
};
```

Step 3: Implement the Concrete Colleague class

```cpp
// Step 3: Implement the Concrete Colleague class
class User : public Colleague {
public:
    User(const std::string& name, Mediator* mediator) : Colleague(name, mediator) {}

    void send(const std::string& message, Colleague* recipient) override {
        mediator->sendMessage(message, this, recipient);
    }

    void receive(const std::string& message) override {
        std::cout << "Received message: " << message << " (User: " << name << ")" << std::endl;
    }
};
```

Step 4: Usage example

```cpp
int main() {
    // Step 4: Usage example
    Mediator* chatroom = new Chatroom();

    Colleague* user1 = new User("Alice", chatroom);
    Colleague* user2 = new User("Bob", chatroom);
    Colleague* user3 = new User("Charlie", chatroom);

    chatroom->addColleague(user1);
    chatroom->addColleague(user2);
    chatroom->addColleague(user3);

    user1->send("Hello, Bob!", user2);
    user2->send("Hi, Alice!", user1);
    user3->send("Hey, guys!");
```

```
16
17      delete chatroom;
18      delete user1;
19      delete user2;
20      delete user3;
21
22      return 0;
23  }
```

3.3.7 Memento

1 Definition

The **Memento** is a behavioral design pattern that allows you to preserve and recover the previous state of an object, all while keeping the implementation details hidden.

2 Problem

Imagine developing a text editor application that offers text editing, formatting, and image insertion capabilities. To meet user expectations, you decide to incorporate an undo feature that allows users to revert any performed operations on the text. Initially, you opt for a direct approach where the application records the state of all objects before executing an operation and saves it in storage. Subsequently, when a user wishes to undo an action, the application retrieves the latest snapshot from the history and utilizes it to restore the state of all objects.

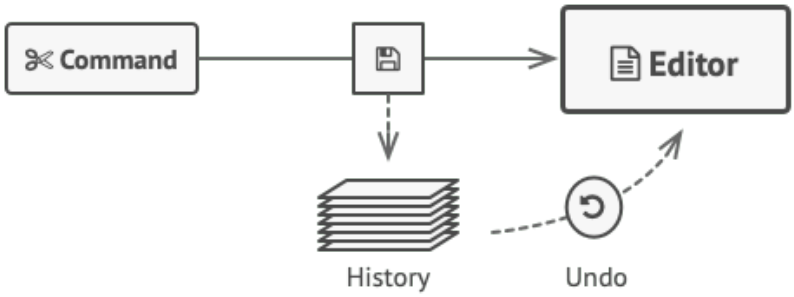


Figure 3.67: Before executing an operation, the app saves a snapshot of the objects' state, which can later be used to restore objects to their previous state.

Considering the process of creating state snapshots, you would typically need to traverse all the fields in an object and copy their values into storage. However, this approach assumes relaxed access restrictions, which may not be feasible for most real objects with private fields concealing significant data. For the sake of discussion, let's assume that our objects have open relations and expose their state publicly. Although this would allow producing snapshots of object states without immediate issues, it poses challenges in scenarios where class refactoring or modification of fields becomes necessary. In such cases, the classes responsible for copying the state of affected objects would also require changes.

Moreover, let's contemplate the composition of the editor's state snapshots. What information should these snapshots contain? At the very least, they need to encompass the actual text, cursor coordinates, current scroll position, and other relevant data. To generate a snapshot, you would gather these values and organize them within a container of some sort.

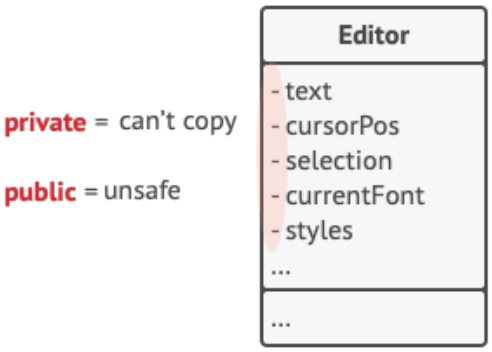


Figure 3.68: How to make a copy of the object's private state?

In all likelihood, you would store multiple instances of these container objects in a list representing the history. Consequently, the containers would likely belong to a single class, comprising numerous fields

that mirror the editor's state. To enable other objects to read from and write to a snapshot, you may need to make its fields public. However, this approach would expose all the editor's states, regardless of their privacy level. As a result, any changes to the snapshot class would create dependencies in other classes, even for private fields and methods that would otherwise remain unaffected.

This situation seems to present a deadlock: either expose all internal details of classes, rendering them fragile, or restrict access to their state, making it impossible to produce snapshots. Are there alternative approaches for implementing the "undo" functionality?

3 Solution

The issues encountered earlier stem from broken encapsulation, where certain objects exceed their intended responsibilities. In order to collect data required for specific actions, these objects intrude into the private space of other objects instead of allowing them to perform the actions themselves.

The Memento pattern addresses these problems by delegating the creation of state snapshots to the actual owner of the state, the originator object. Consequently, instead of external objects attempting to copy the editor's state, the editor class itself can generate the snapshot since it possesses full access to its own state.

The pattern proposes storing a copy of the object's state in a specialized object called the memento. The contents of the memento are inaccessible to any other object except the one that produced it. Other objects must interact with mementos through a limited interface, which may allow fetching metadata about the snapshot (such as creation time and the name of the operation performed), but not the original object's state contained within the snapshot.

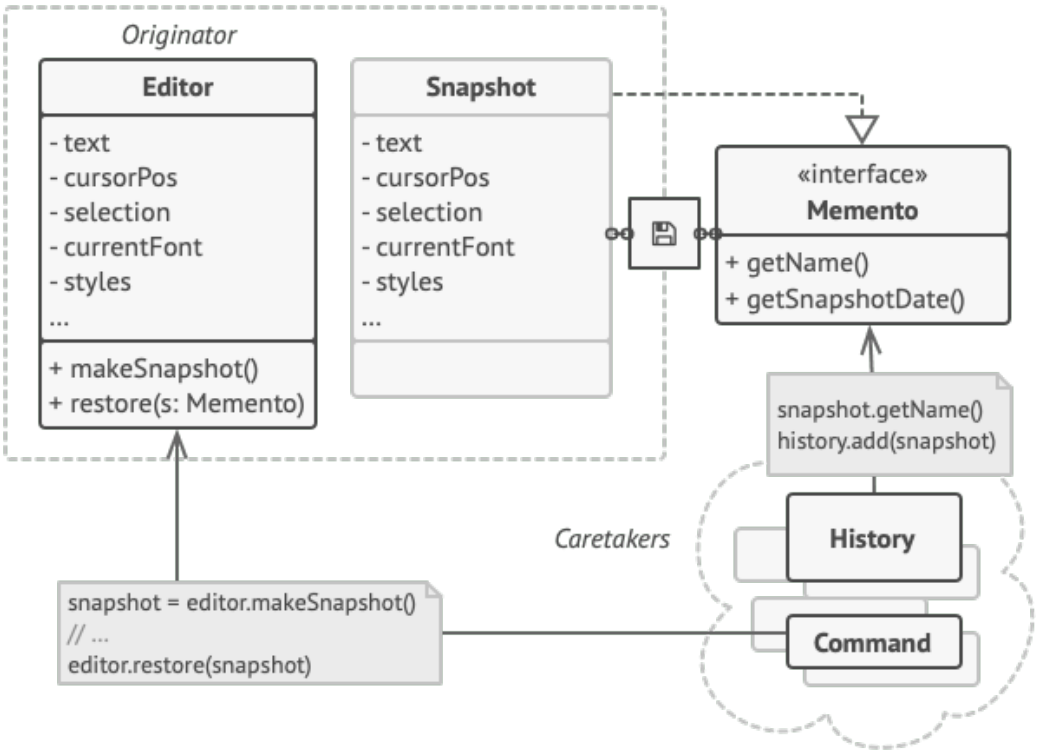


Figure 3.69: The originator has full access to the memento, whereas the caretaker can only access the metadata.

This restrictive approach enables storing mementos within other objects, typically referred to as caretakers. As the caretaker only interacts with the memento via the limited interface, it cannot modify the state stored within the memento. Meanwhile, the originator has access to all fields inside the memento, enabling it to restore its previous state whenever necessary.

In the context of our text editor example, we can create a separate history class to serve as the caretaker. A stack of mementos stored within the caretaker will grow as the editor executes operations. This stack could even be visualized within the application's user interface, displaying the history of previously performed operations to the user.

When a user triggers the undo functionality, the history retrieves the most recent memento from the stack and returns it to the editor, requesting a rollback. Since the editor has complete access to the memento, it modifies its own state using the values extracted from the memento.

4 Structure

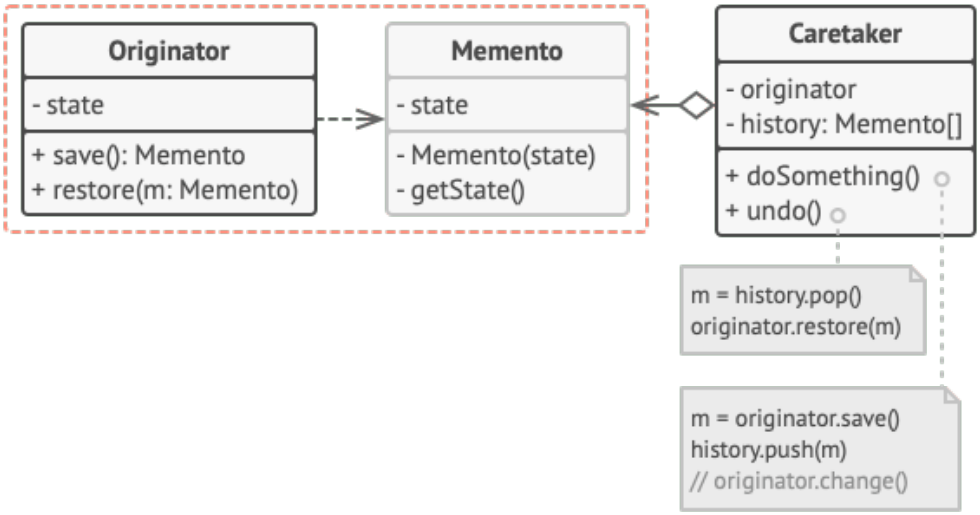


Figure 3.70: General structure of Memento

The Originator class possesses the ability to create snapshots of its own state and restore its state from these snapshots as required.

The Memento, acting as a value object, serves as a snapshot of the Originator's state. It is commonly implemented as an immutable object, receiving the data only once through its constructor.

The Caretaker is responsible for determining when and why to capture the Originator's state, as well as when to restore the state.

To keep track of the Originator's history, the Caretaker maintains a stack of mementos. When the Originator needs to revert to a previous state, the Caretaker retrieves the topmost memento from the stack and passes it to the Originator's restoration method.

In this implementation, the Memento class is nested within the Originator. This allows the Originator to access the fields and methods of the Memento, even though they are declared private. Conversely, the Caretaker has limited access to the Memento's fields and methods, enabling it to store mementos in a stack without altering their state.

5 Implementation

1. Identify the class that will serve as the Originator. Determine whether the program will utilize a single central object of this type or multiple smaller ones.

2. Create the Memento class and declare a set of fields within it that mirror the fields declared in the Originator class.

3. Ensure that the Memento class is immutable. The Memento should accept the data only once through its constructor and should not have any setters.

4. If your programming language supports nested classes, nest the Memento class inside the Originator. If not, extract a blank interface from the Memento class and have other objects use it to refer to the Memento. You can add metadata operations to the interface, but avoid exposing the Originator's state.
5. Add a method to the Originator class for producing Mementos. The Originator should pass its state to the Memento through one or multiple arguments of the Memento's constructor.
6. The return type of the Memento-producing method should be the interface extracted in the previous step (if applicable). Under the hood, the method should directly work with the Memento class.
7. Add a method to the Originator class for restoring its state. This method should accept a Memento object as an argument. If an interface was extracted in the previous step, make it the type of the parameter. In this case, you will need to typecast the incoming object to the Memento class, as the Originator requires full access to that object.
8. The Caretaker, whether it represents a command object, a history, or any other entity, should know when to request new Mementos from the Originator, how to store them, and when to restore the Originator with a particular Memento.
9. The link between Caretakers and Originators could be established within the Memento class. Each Memento would then be associated with the Originator that created it. The restoration method would also be moved to the Memento class. However, this approach is only viable if the Memento class is nested inside the Originator or if the Originator provides sufficient setters for overriding its state.

6 Advantages and disadvantages

Advantages:

- Snapshots of an object's state can be generated without violating its encapsulation.
- The originator's code can be simplified by delegating the responsibility of maintaining the history of the originator's state to the caretaker.

Disadvantages:

- The application may consume significant amounts of RAM if clients create mementos too frequently.
- Caretakers need to track the originator's lifecycle in order to dispose of obsolete mementos.
- Most dynamic programming languages, such as PHP, Python, and JavaScript, cannot guarantee that the state within the memento remains unchanged.

7 Example

Suppose we have a text editor application that allows users to write and modify text. We want to implement an undo feature that allows users to revert to previous states of the text editor.

Step 1: Define the Originator class and Memento class

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

class Memento {
    std::string text;

```

```
public:
    Memento(const std::string& t) : text(t) {}

    std::string getText() const {
        return text;
    }
};

class TextEditor {
    std::string text;

public:
    void setText(const std::string& t) {
        text = t;
    }

    std::string getText() const {
        return text;
    }

    Memento createMemento() const {
        return Memento(text);
    }

    void restore(const Memento& memento) {
        text = memento.getText();
    }
};
```

Step 2: Create a Caretaker class

```
class Caretaker {
    Memento memento;

public:
    void saveMemento(const TextEditor& textEditor) {
        memento = textEditor.createMemento();
    }

    void restoreText(TextEditor& textEditor) {
        textEditor.restore(memento);
    }
};
```

Step 3: Usage example

```
int main() {
    TextEditor textEditor;
    Caretaker caretaker;

    // Type and modify the text
    textEditor.setText("Hello");
    std::cout << "Current Text: " << textEditor.getText() << std::endl;

    // Save the current state
```

```
    caretaker.saveMemento(textEditor);

    // Modify the text further
    textEditor.setText("Hello, World!");

    std::cout << "Current Text: " << textEditor.getText() << std::endl;

    // Restore the previous state
    caretaker.restoreText(textEditor);

    std::cout << "Restored Text: " << textEditor.getText() << std::endl;

    return 0;
}
```

3.3.8 Observer

1 Definition

The Observer pattern is a behavioral design pattern that allows for the definition of a subscription mechanism. This mechanism notifies multiple objects about any events that occur in the object they are observing.

2 Problem

Consider a scenario with two types of objects: a `Customer` and a `Store`. The customer is highly interested in a specific brand of product, such as a new model of the iPhone, which is expected to arrive at the store soon.

The customer has the option to visit the store regularly to check if the desired product is available. However, this approach would be inefficient and futile for most visits, as the product is still in transit.

Alternatively, the store could choose to send numerous emails to all customers whenever a new product becomes available. While this may save some customers from unnecessary visits, it could also be perceived as spam and upset those who have no interest in new products.

This situation presents a conflict: either the customer wastes time checking availability or the store wastes resources by notifying customers who have no interest in the product.

3 Solution

The object that possesses an interesting state is commonly referred to as the subject. However, since it also notifies other objects about changes in its state, we will refer to it as the publisher. The other objects that wish to track changes in the publisher's state are known as subscribers.

The Observer pattern proposes incorporating a subscription mechanism within the publisher class, enabling individual objects to subscribe to or unsubscribe from a stream of events generated by the publisher. Rest assured, this mechanism is not as complex as it may initially seem. In reality, it involves: 1) an array field to store references to subscriber objects, and 2) several public methods that allow adding and removing subscribers from the list.

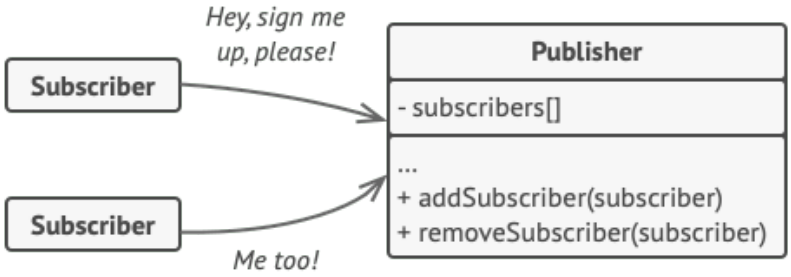


Figure 3.71: A subscription mechanism lets individual objects subscribe to event notifications.

Whenever a significant event occurs within the publisher, it iterates through its list of subscribers and invokes the specific notification method on their respective objects.

Real-world applications may feature numerous subscriber classes interested in tracking events from the same publisher class. It is undesirable to tightly couple the publisher with all these classes, especially when some of them may not be known in advance if the publisher class is intended for use by external parties.

Hence, it is crucial for all subscribers to implement the same interface, and the publisher should communicate with them solely through that interface. This interface should declare the notification method and include a set of parameters for the publisher to pass contextual data along with the notification.

If your application encompasses various types of publishers, and you aim to make the subscribers compatible with all of them, you can take a step further and ensure all publishers adhere to the same

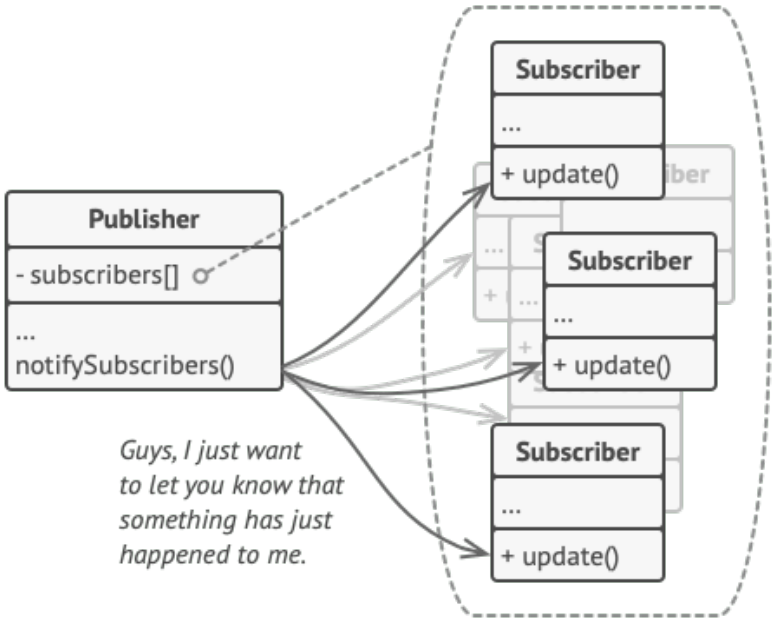


Figure 3.72: Publisher notifies subscribers by calling the specific notification method on their objects.

interface. This interface would solely need to describe a few subscription methods, enabling subscribers to observe the state of publishers without coupling to their specific classes.

4 Structure

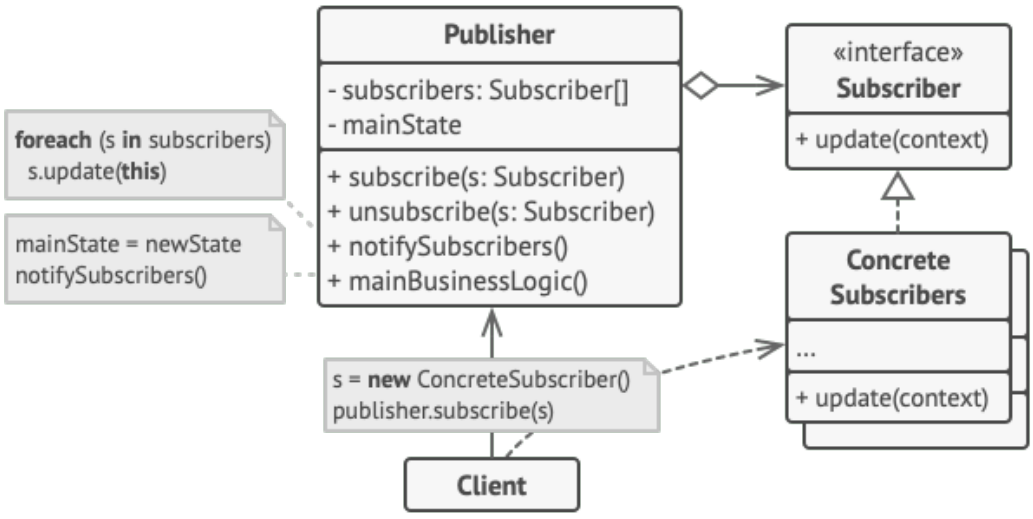


Figure 3.73: Structure of Observer

The publisher emits events of interest to other objects. These events occur when the publisher undergoes state changes or performs certain behaviors. Publishers include a subscription infrastructure that enables new subscribers to join the list and existing subscribers to leave.

When a new event occurs, the publisher iterates through the subscription list and invokes the notification method, as declared in the subscriber interface, on each subscriber object.

The Subscriber interface declares the notification method, typically consisting of a single update method. This method may accept multiple parameters, allowing the publisher to provide event details alongside the update.

Concrete Subscribers execute specific actions in response to notifications issued by the publisher. All

of these classes must implement the same interface to avoid coupling between the publisher and concrete classes.

Subscribers often require contextual information to handle updates correctly. For this reason, publishers frequently pass context data as arguments to the notification method. The publisher can include itself as an argument, enabling subscribers to directly access any required data.

The Client creates publisher and subscriber objects separately and then registers subscribers for receiving updates from the publisher.

5 Implementation

1. Examine the business logic and divide it into two parts: the core functionality, which will serve as the publisher, and the remaining code, which will become a set of subscriber classes.

2. Declare the subscriber interface, which should minimally include a single update method.

3. Declare the publisher interface and define a pair of methods for adding a subscriber object to and removing it from the list. Remember that publishers should interact with subscribers solely through the subscriber interface.

4. Determine the appropriate location for the subscription list and the implementation of subscription methods. Typically, this code remains consistent across different types of publishers, making it suitable for placement in an abstract class directly derived from the publisher interface. Concrete publishers can then extend this class to inherit the subscription behavior.

5. However, if you are applying the pattern to an existing class hierarchy, consider using a composition-based approach. Place the subscription logic in a separate object and have all actual publishers utilize it.

6. Create concrete publisher classes. Whenever a significant event occurs within a publisher, it should notify all of its subscribers.

7. Implement the update notification methods in the concrete subscriber classes. Most subscribers will require some context data regarding the event, which can be passed as an argument to the notification method.

8. Alternatively, the subscriber can directly fetch any required data from the notification upon receiving it. In this case, the publisher must pass itself as an argument via the update method. Another, less flexible option is to establish a permanent link between the publisher and the subscriber through the constructor.

9. The client is responsible for creating all necessary subscribers and registering them with the appropriate publishers.

6 Advantages and disadvantages

Advantages:

- Open/Closed Principle: The Observer pattern adheres to the Open/Closed Principle, allowing the introduction of new subscriber classes without requiring modifications to the publisher's code. Similarly, changes to the publisher interface can be made without affecting existing subscribers.

- Runtime Object Relations: The Observer pattern enables the establishment of object relations at runtime. Subscribers can dynamically subscribe to or unsubscribe from a publisher's events, providing flexibility in managing these relationships.

Disadvantages:

- Random Order of Notification: One drawback of the Observer pattern is that subscribers are notified in a random order. This lack of control over the sequence in which subscribers receive notifications may pose challenges in certain scenarios where the order of notification is crucial.

7 Example

Suppose we have a weather monitoring system that needs to notify multiple display devices whenever the weather conditions change. Each display device should be able to subscribe to the weather updates and receive notifications in real-time.

Solution

Step 1: Define the Subject interface and its implementations.

```cpp
// Step 1: Define the Subject interface and its implementations
#include <iostream>
#include <vector>

// Forward declaration
class Observer;

class Subject {
protected:
    std::vector<Observer*> observers;

public:
    virtual void attach(Observer* observer) = 0;
    virtual void detach(Observer* observer) = 0;
    virtual void notify() = 0;
};

class WeatherStation : public Subject {
private:
    int temperature;
    int humidity;

public:
    void setWeatherConditions(int temperature, int humidity) {
        this->temperature = temperature;
        this->humidity = humidity;
        notify();
    }

    void attach(Observer* observer) override {
        observers.push_back(observer);
    }

    void detach(Observer* observer) override {
        // Find and remove the observer
        for (auto it = observers.begin(); it != observers.end(); ++it) {
            if (*it == observer) {
                observers.erase(it);
                break;
            }
        }
    }
```

```
43
44      void notify() override {
45          for (auto observer : observers) {
46              observer->update(temperature, humidity);
47          }
48      }
49  };
```

Step 2: Define the Observer interface and its implementations.

```
1   // Step 2: Define the Observer interface and its implementations
2   class Observer {
3   public:
4       virtual void update(int temperature, int humidity) = 0;
5   };
6
7   class DisplayDevice : public Observer {
8   public:
9       void update(int temperature, int humidity) override {
10          std::cout << "Displaying weather conditions. Temperature: " << ↩
               temperature
11                  << "C, Humidity: " << humidity << "%" << std::endl;
12      }
13  };
```

Step 3: Usage example.

```
1   // Step 3: Usage example
2   int main() {
3       // Create weather station and display devices
4       WeatherStation weatherStation;
5       DisplayDevice displayDevice1;
6       DisplayDevice displayDevice2;
7
8       // Attach display devices to the weather station
9       weatherStation.attach(&displayDevice1);
10      weatherStation.attach(&displayDevice2);
11
12      // Set weather conditions
13      weatherStation.setWeatherConditions(25, 70);
14
15      // Detach displayDevice1
16      weatherStation.detach(&displayDevice1);
17
18      // Set new weather conditions
19      weatherStation.setWeatherConditions(30, 80);
20
21      return 0;
22  }
```

3.3.9 State

1 Definition

The **State** pattern is a behavioral design pattern that enables an object to modify its behavior based on changes in its internal state. This allows the object to appear as if it has changed its class.

2 Problem

The **State** pattern is closely associated with the concept of a Finite-State Machine.

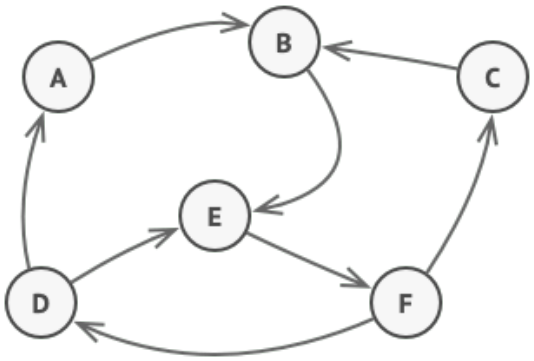


Figure 3.74: Finite-State Machine.

The fundamental idea behind this pattern is that a program can exist in a finite number of states, where each state results in different behavior. The program can transition between states instantaneously, but these transitions are determined by predetermined rules.

This approach can also be applied to objects. Consider a hypothetical example with a `Document` class that can be in one of three states: `Draft`, `Moderation`, and `Published`. The `publish` method of the document behaves differently depending on the current state:

- In `Draft` state, it transitions the document to `Moderation`.
- In `Moderation` state, it makes the document public only if the current user is an administrator.
- In `Published` state, it doesn't perform any action.

State machines are typically implemented using conditional statements (`if` or `switch`) that determine the appropriate behavior based on the object's current state. Usually, this "state" is represented by a set of values in the object's fields. Even if you haven't heard of finite-state machines before, you might have implemented a similar concept at some point. Does the following code structure look familiar?

```
if (state == State.Draft) {
    // Perform actions for Draft state
}
else if (state == State.Moderation) {
    // Perform actions for Moderation state
}
else if (state == State.Published) {
    // Perform actions for Published state
}
```

However, a significant drawback of using conditional statements for state machines becomes apparent as the number of states and state-dependent behaviors in the `Document` class grows. Most methods end

Figure 3.75: Possible states and transitions of a document object.

up with complex conditional logic to determine the appropriate behavior based on the current state. Maintaining code like this becomes challenging because any changes to the transition logic require modifying the state conditionals in every method.

As the project evolves, the problem becomes more significant. It's challenging to anticipate all possible states and transitions during the design stage. Consequently, a state machine initially implemented with a limited set of conditionals can become excessively complex over time.

3 Solution

The State pattern proposes the creation of separate classes for each possible state of an object and extracting state-specific behaviors into these classes. Instead of implementing all behaviors within the object itself (known as the context), the context holds a reference to one of the state objects representing its current state. It delegates all state-related tasks to this object.

To transition the context into a different state, the active state object is replaced with another object that represents the new state. This transition is only possible if all state classes adhere to the same interface, and the context interacts with these objects through that interface.

This structure may resemble the Strategy pattern, but there is one crucial distinction. In the State pattern, the individual states may be aware of each other and initiate transitions from one state to another. In contrast, strategies typically have no knowledge of other strategies.

4 Structure

The context object in the State pattern maintains a reference to a specific concrete state object and delegates all state-specific tasks to it. The communication between the context and the state object is done through the state interface. The context provides a setter method to update its state by passing a new state object.

The state interface declares the methods that are specific to each state. These methods should be meaningful and applicable to all concrete states to avoid having irrelevant methods in certain states that will never be called.

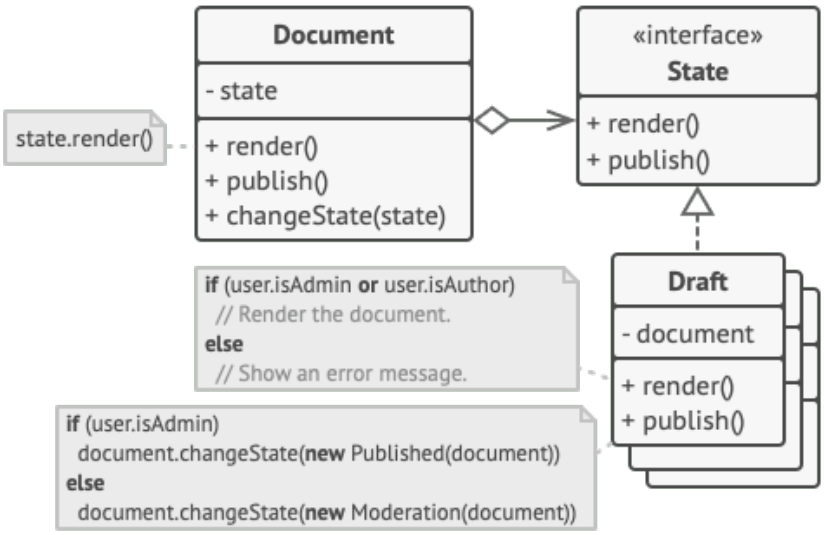


Figure 3.76: Document delegates the work to a state object.

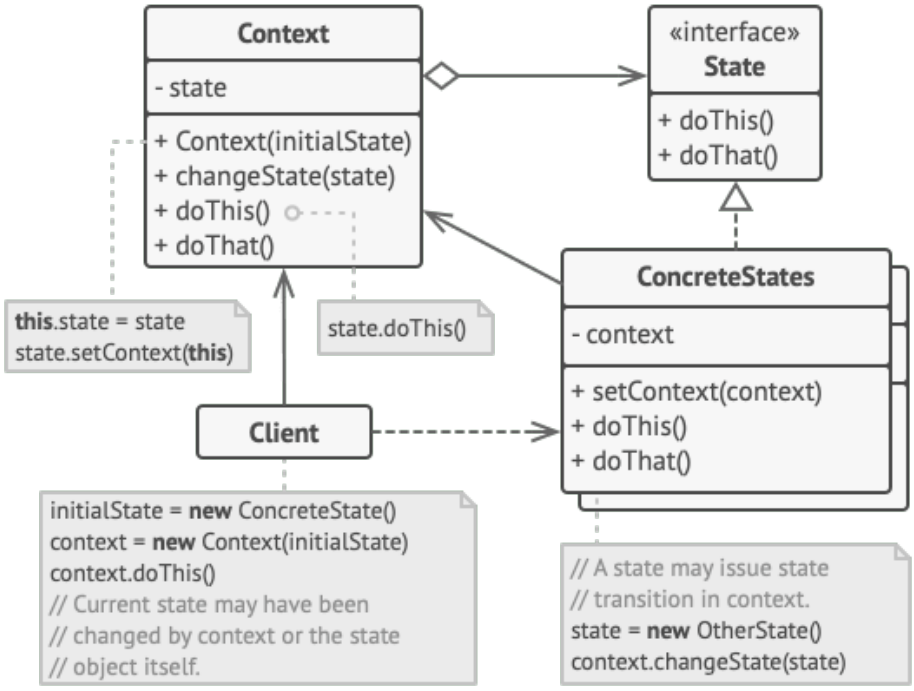


Figure 3.77: Structure of State

Concrete states implement their own versions of the state-specific methods. To eliminate code duplication across multiple states, intermediate abstract classes can be used to encapsulate common behaviors.

State objects may also store a backreference to the context object. This enables the state to retrieve any necessary information from the context and initiate state transitions.

Both the context and concrete states have the ability to set the next state of the context and perform the actual state transition by replacing the state object associated with the context.

5 Implementation

1. Determine which class will serve as the context. This can be an existing class that already contains state-dependent code or a new class if the state-specific code is spread across multiple classes.

2. Declare the state interface. It should include methods that may contain state-specific behavior. The

interface may mirror the methods declared in the context.

3. Create a class for each actual state by deriving them from the state interface. Extract the code related to each state from the context and place it in the respective state class.

4. During the code extraction process, you may encounter dependencies on private members of the context. You can handle this in different ways:

   - Make the required fields or methods public.
   - Turn the extracted behavior into a public method within the context and call it from the state class. Although not ideal, this approach can be a quick workaround that can be improved later.
   - Nest the state classes into the context class if your programming language supports nested classes.
   - Add a reference field of the state interface type in the context class, along with a public setter to override the value of that field.

5. Review the methods of the context class and replace empty state conditionals with calls to corresponding methods of the state object.

6. To switch the state of the context, create an instance of one of the state classes and pass it to the context. This can be done within the context itself, within various states, or in the client code. Keep in mind that the class performing the state switch becomes dependent on the specific concrete state class it instantiates.

6 Advantages and disadvantages

Advantages:

- **Single Responsibility Principle:** The State pattern allows for organizing code related to specific states into separate classes, ensuring that each class has a single responsibility.
- **Open/Closed Principle:** By using the State pattern, new states can be introduced without modifying existing state classes or the context. This promotes the principle of being open for extension but closed for modification.
- **Code Simplification:** The State pattern simplifies the code within the context by removing the need for complex state machine conditionals. Each state class handles its own behavior, resulting in a more modular and maintainable codebase.

Disadvantages:

- **Overkill for Simple State Machines:** Applying the State pattern may be excessive for state machines with only a few states or infrequent state changes. In such cases, the additional complexity introduced by the pattern may outweigh the benefits it provides.

7 Example

Suppose we have a vending machine that dispenses different products, such as snacks and beverages. The machine should change its behavior based on its current state, such as when it's out of stock or when it's ready to dispense a product. We want to implement a flexible solution that allows easy addition of new states and behavior without modifying the existing code.

Solution:
Step 1: Define the state interface and its implementations.

```cpp
// Step 1: Define the state interface and its implementations
class VendingMachineState {
    public:
        virtual void insertCoin() = 0;
        virtual void selectProduct() = 0;
        virtual void dispenseProduct() = 0;
        virtual void refill() = 0;
};

class OutOfStockState : public VendingMachineState {
    public:
        void insertCoin() override {
        // Display "Out of stock" message
        }

        void selectProduct() override {
            // Display "Out of stock" message
        }

        void dispenseProduct() override {
            // Display "Out of stock" message
        }

        void refill() override {
            // Refill the vending machine
            // Transition to another state
        }
};

class ReadyToDispenseState : public VendingMachineState {
    public:
        void insertCoin() override {
            // Display "Already selected a product" message
        }

        void selectProduct() override {
            // Display "Already selected a product" message
        }

        void dispenseProduct() override {
            // Dispense the selected product
            // Transition to another state
        }

        void refill() override {
            // Display "Cannot refill while ready to dispense" message
        }
};

// Add more state classes if needed
```

Step 2: Implement the context class that uses the state interface.

```cpp
// Step 2: Implement the context class that uses the state interface
class VendingMachine {
```

```
    private:
        VendingMachineState* currentState;

    public:
        VendingMachine() {
            // Set the initial state
            currentState = new OutOfStockState();
        }

        void changeState(VendingMachineState* newState) {
            delete currentState;
            currentState = newState;
        }

        void insertCoin() {
            currentState->insertCoin();
        }

        void selectProduct() {
            currentState->selectProduct();
        }

        void dispenseProduct() {
            currentState->dispenseProduct();
        }

        void refill() {
            currentState->refill();
        }
};
```

Step 3: Usage example.

```
int main() {
    VendingMachine vendingMachine;
    vendingMachine.insertCoin(); // Displays "Out of stock" message

    vendingMachine.refill();

    vendingMachine.selectProduct();
    vendingMachine.insertCoin();
    vendingMachine.dispenseProduct(); // Dispenses the selected product

    vendingMachine.refill();

    vendingMachine.selectProduct();
    vendingMachine.dispenseProduct(); // Displays "Already selected a ↩
       product" message

    return 0;
}
```

3.3.10 Strategy

1 Definition

The strategy pattern is a design pattern that enables you to establish a group of algorithms. Each algorithm is encapsulated within its own class, allowing for interchangeability of objects representing these algorithms.

2 Problem

One day, you made the decision to develop a navigation application specifically tailored for casual travelers. The application revolved around a visually appealing map that aided users in swiftly orienting themselves within any city.

Among the most sought-after functionalities for the application was automated route planning. Users should have the ability to input an address and have the fastest route to their destination displayed on the map.

Initially, the app could only generate routes for road travel. This pleased car travelers immensely. However, it became apparent that not everyone enjoys driving during their vacations. Consequently, in the subsequent update, you introduced the option to create walking routes. Shortly thereafter, an additional option was included to allow users to incorporate public transportation into their routes.

Nevertheless, this was only the beginning. Future plans included incorporating route planning for cyclists and, eventually, implementing a feature that would generate routes encompassing all of a city's tourist attractions.

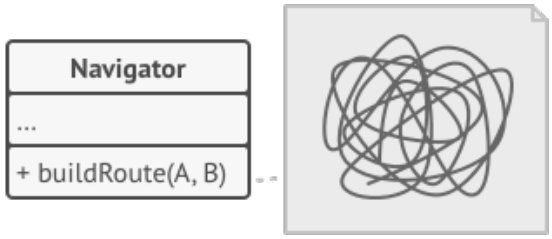


Figure 3.78: The code of the navigator became bloated.

Although the app achieved success in terms of business, the technical aspect posed numerous challenges. Each time a new routing algorithm was added, the primary navigator class doubled in size. Eventually, it became unmanageable.

Any modification made to one of the algorithms, whether it was a simple bug fix or a minor adjustment to the street scoring system, impacted the entire class, increasing the risk of introducing errors into previously functional code.

Furthermore, collaboration among team members became inefficient. Colleagues, who were recruited following the app's successful launch, expressed frustrations regarding the amount of time spent resolving merge conflicts. Introducing a new feature necessitated modifying the same extensive class, leading to conflicts with code produced by other individuals.

3 Solution

The Strategy pattern proposes the extraction of various algorithms performed by a specific class into separate strategy classes.

The original class, known as the context, includes a field for storing a reference to one of the strategies. Instead of executing the task internally, the context delegates the work to the linked strategy object.

The responsibility of selecting an appropriate algorithm does not lie with the context but with the client. The client passes the desired strategy to the context. In fact, the context possesses limited

knowledge about the strategies and interacts with all strategies through a unified interface. This interface exposes a single method for triggering the algorithm encapsulated within the selected strategy.

By adopting this approach, the context becomes independent of concrete strategies, allowing for the addition of new algorithms or modifications to existing ones without requiring changes to the context or other strategies.

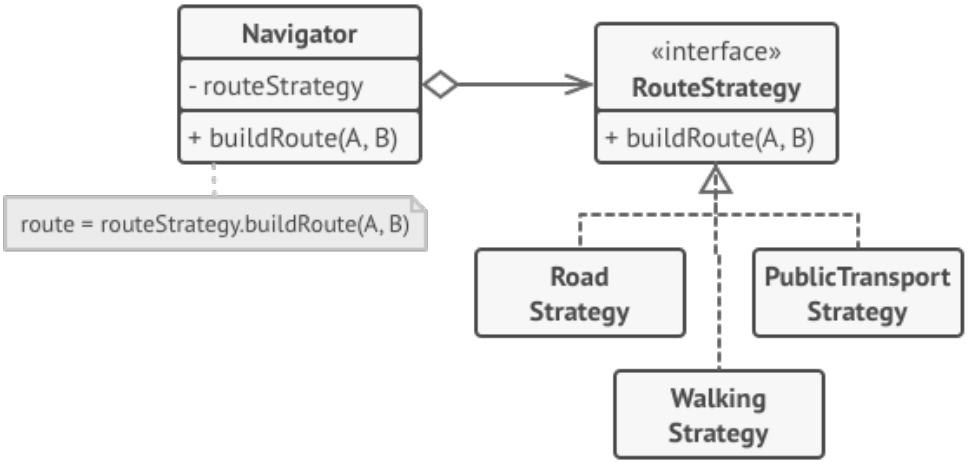


Figure 3.79: Route planning strategies.

In our navigation app, each routing algorithm can be abstracted into its own class, featuring a single method named "buildRoute." This method accepts an origin and destination, returning a collection of checkpoints comprising the route.

Although each routing class may generate a distinct route given the same arguments, the main navigator class remains agnostic to the selected algorithm. Its primary responsibility is to render a set of checkpoints on the map. The class includes a method for switching the active routing strategy, enabling clients (e.g., buttons in the user interface) to replace the currently chosen routing behavior with another.

4 Structure

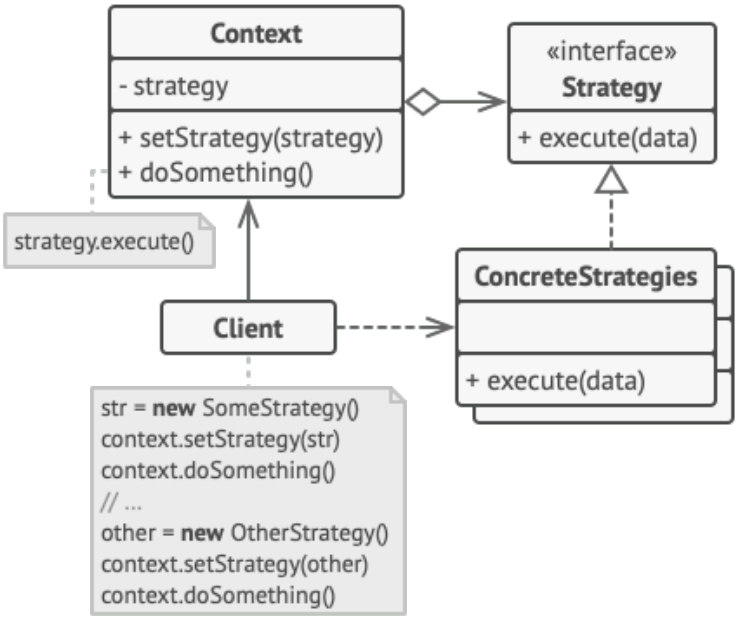


Figure 3.80: Structure of Strategy

The Context class maintains a reference to a specific concrete strategy and communicates exclusively through the strategy interface.

The Strategy interface is shared among all concrete strategies. It defines a method that the context utilizes to execute a strategy.

Concrete strategies implement diverse variations of the algorithm employed by the context.

Whenever the algorithm needs to be executed, the context invokes the execution method on the linked strategy object. The context remains unaware of the strategy type or the internal workings of the algorithm.

The Client is responsible for creating a particular strategy object and passing it to the context. The context provides a setter method that enables clients to dynamically replace the strategy associated with the context at runtime.

5 Implementation

In the context class, identify an algorithm that's prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.

Declare the strategy interface common to all variants of the algorithm.

One by one, extract all algorithms into their own classes. They should all implement the strategy interface.

In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.

Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

6 Advantages and disadvantages

Advantages:

- Runtime Algorithm Swapping: The strategy pattern allows for the dynamic swapping of algorithms within an object during runtime.
- Implementation Isolation: The pattern separates the implementation details of an algorithm from the code that utilizes it, promoting encapsulation and maintainability.
- Composition over Inheritance: The strategy pattern favors composition over inheritance, enabling flexible and modular design.
- Open/Closed Principle: New strategies can be introduced without modifying the existing context code, promoting the open/closed principle.

Disadvantages:

- Unnecessary Complexity: If there are only a few algorithms that rarely change, implementing the strategy pattern may introduce unnecessary complexity with additional classes and interfaces.
- Client Awareness: Clients need to be knowledgeable about the differences between strategies in order to select the appropriate one.
- Functional Alternatives: Many modern programming languages support functional types, allowing for the implementation of various algorithm versions using anonymous functions. This alternative approach can be used without the need for additional classes and interfaces.

7 Example

You are developing a video game that features different types of enemies. Each enemy has a unique behavior when it comes to attacking the player. However, you want to implement a flexible and extensible system that allows you to easily add new enemy types and their corresponding attack behaviors without modifying the existing code.

Solution

Step 1: Define the interface for the attack behavior.

```
// Step 1: Define the interface for the attack behavior
class AttackBehavior {
public:
    virtual void attack() = 0;
    virtual ~AttackBehavior() {}
};
```

Step 2: Implement concrete attack behavior classes.

```
// Step 2: Implement concrete attack behavior classes
class MeleeAttack : public AttackBehavior {
public:
    void attack() override {
        // Implement melee attack logic
        std::cout << "Performing melee attack!" << std::endl;
    }
};

class RangedAttack : public AttackBehavior {
public:
    void attack() override {
        // Implement ranged attack logic
        std::cout << "Performing ranged attack!" << std::endl;
    }
};

class MagicAttack : public AttackBehavior {
public:
    void attack() override {
        // Implement magic attack logic
        std::cout << "Performing magic attack!" << std::endl;
    }
};
```

Step 3: Create a class that uses the attack behavior.

```
// Step 3: Create a class that uses the attack behavior
class Enemy {
private:
    AttackBehavior* attackBehavior;

public:
    Enemy(AttackBehavior* behavior) : attackBehavior(behavior) {}
```

```
8
9       void setAttackBehavior(AttackBehavior* behavior) {
10          attackBehavior = behavior;
11      }
12
13      void performAttack() {
14          attackBehavior->attack();
15      }
16  };
```

Step 4: Demonstrate the usage of the Strategy pattern.

```
1   // Step 4: Demonstrate the usage of the Strategy pattern
2   int main() {
3       // Create enemy objects
4       Enemy enemy1(new MeleeAttack());
5       Enemy enemy2(new RangedAttack());
6       Enemy enemy3(new MagicAttack());
7
8       // Perform attacks
9       enemy1.performAttack();  // Output: Performing melee attack!
10      enemy2.performAttack();  // Output: Performing ranged attack!
11      enemy3.performAttack();  // Output: Performing magic attack!
12
13      // Change attack behavior dynamically
14      enemy1.setAttackBehavior(new RangedAttack());
15      enemy1.performAttack();  // Output: Performing ranged attack!
16
17      return 0;
18  }
```

3.3.11 Visitor

1 Definition

The Visitor pattern is a behavioral design pattern that enables the separation of algorithms from the objects they act upon.

2 Problem

Imagine that your team develops an application that handles geographic information stored as a large graph structure. The graph consists of nodes representing various entities, such as cities, industries, and sightseeing areas. These nodes are interconnected based on real-world relationships. Each node type is represented by its own class, and specific nodes are instantiated as objects.

Figure 3.81: Exporting the graph into XML.

Your task is to implement the functionality to export the graph into XML format. Initially, you planned to add an export method to each node class and recursively traverse the graph, calling the export method on each node. This solution utilized polymorphism, allowing the export code to be decoupled from specific node classes.

However, the system architect rejected the idea of modifying the existing node classes due to the risk of introducing bugs into the production code. Additionally, he raised concerns about placing XML export code within the node classes, as their primary purpose was to handle geodata, making the inclusion of XML export behavior seem out of place.

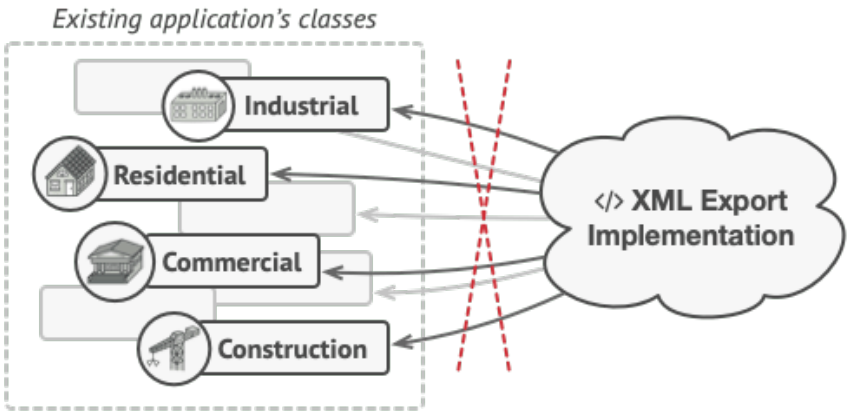


Figure 3.82: The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

Another reason for the refusal was the anticipation of future requests from the marketing department, such as exporting to different formats or implementing other non-standard features. These potential

changes would require further modifications to the fragile node classes.

3 Solution

The Visitor pattern suggests segregating new behavior into a separate class called a visitor, rather than attempting to integrate it into existing classes. The original object responsible for performing the behavior is now passed as an argument to one of the visitor's methods, granting the method access to all the necessary data contained within the object.

However, what if this behavior can be applied to objects of different classes? For instance, in our case of XML export, the implementation might vary across different node classes. Thus, the visitor class can define not just one, but a set of methods, each capable of accepting arguments of different types:

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

But how do we call these methods, particularly when dealing with the entire graph? As the methods have different signatures, we cannot rely on polymorphism. To select the appropriate visitor method for processing a given object, we would need to check its class, which can become unwieldy.

```
foreach (Node node in graph)
    if (node instanceof City)
    exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

You may wonder why we don't use method overloading, where multiple methods share the same name but support different parameter sets. Unfortunately, even if our programming language supports method overloading (such as Java and C#), it does not solve our problem. Since the exact class of a node object is unknown in advance, the overloading mechanism cannot determine the correct method to execute and defaults to the method that accepts an object of the base Node class.

However, the Visitor pattern tackles this issue by employing a technique called Double Dispatch, which enables executing the appropriate method on an object without relying on cumbersome conditionals. Instead of the client selecting the method to call, we delegate this choice to the objects we pass to the visitor as arguments. Since these objects are aware of their own classes, they can more gracefully select the appropriate method on the visitor. They "accept" a visitor and inform it which visiting method should be executed.

```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

// City
class City is
    method accept(Visitor v) is
    v.doForCity(this)
    // ...

```

```
// Industry
class Industry is
    method accept(Visitor v) is
    v.doForIndustry(this)
    // ...
```

I admit that we had to make changes to the node classes after all. However, the change is minimal and allows us to add further behaviors without modifying the code once again.

By extracting a common interface for all visitors, all existing nodes can interact with any visitor introduced into the application. Whenever a new behavior related to nodes is needed, implementing a new visitor class is all that is required.

4 Structure

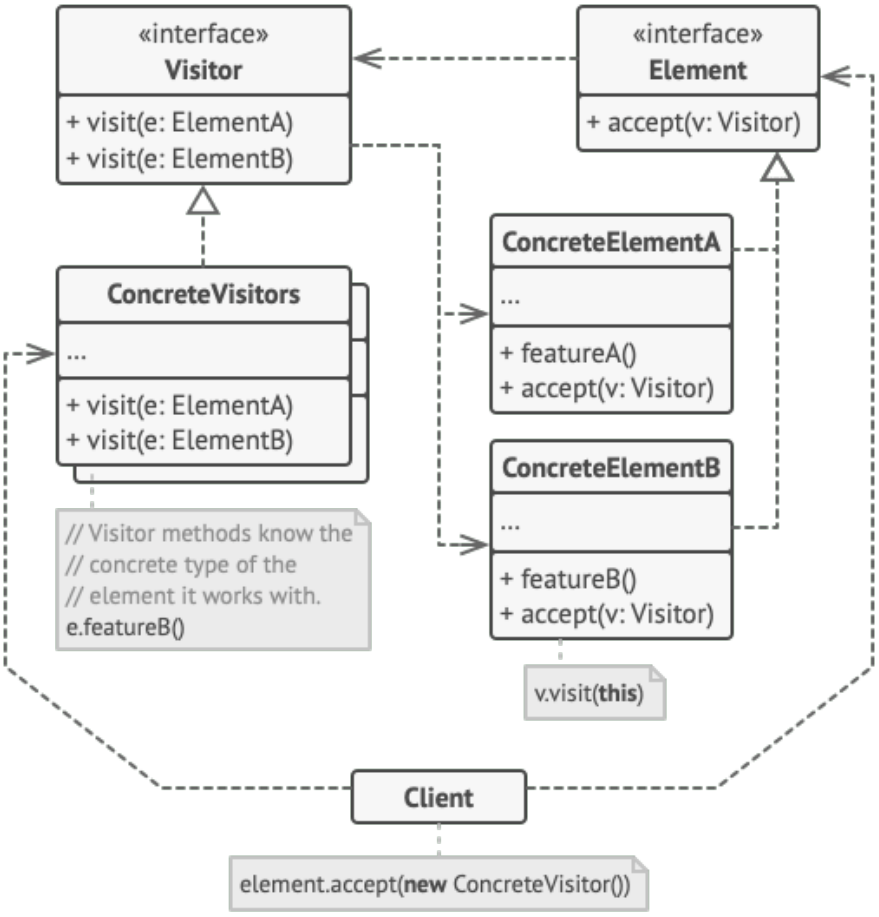


Figure 3.83: Structure of Visitor

The `Visitor` interface declares a set of visiting methods that can accept concrete elements of an object structure as arguments. These methods can have the same names in programming languages that support overloading, but the parameters must have different types.

Each `Concrete Visitor` implements multiple versions of the same behaviors, specifically tailored for different concrete element classes.

The `Element` interface declares an "accept" method, which should have a parameter declared with the type of the visitor interface.

Each `Concrete Element` must implement the accept method. The purpose of this method is to forward the call to the appropriate visitor's method corresponding to the current element class. It is

important to note that even if a base element class implements this method, all subclasses must still override this method in their own classes and invoke the corresponding method on the visitor object.

The `Client` typically represents a collection or another complex object (e.g., a Composite tree). Usually, clients are not aware of all the concrete element classes, as they interact with objects from the collection through some abstract interface.

5 Implementation

1. Declare the visitor interface with a set of "visiting" methods, one per each concrete element class that exists in the program.
2. Declare the element interface. If you're working with an existing element class hierarchy, add the abstract "acceptance" method to the base class of the hierarchy. This method should accept a visitor object as an argument.
3. Implement the acceptance methods in all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object that matches the class of the current element.
4. The element classes should only work with visitors via the visitor interface. Visitors, however, must be aware of all concrete element classes, referenced as parameter types of the visiting methods.
5. For each behavior that cannot be implemented inside the element hierarchy, create a new concrete visitor class and implement all of the visiting methods.
6. You might encounter a situation where the visitor will need access to some private members of the element class. In this case, you can either make these fields or methods public, violating the element's encapsulation, or nest the visitor class in the element class. The latter is only possible if you're fortunate to work with a programming language that supports nested classes.
7. The client must create visitor objects and pass them into elements via "acceptance" methods.

6 Advantages and disadvantages

Advantages:

- **Open/Closed Principle:** You can introduce a new behavior that can operate on objects of different classes without modifying those classes.
- **Single Responsibility Principle:** You can consolidate multiple versions of the same behavior into a single class.
- A visitor object can accumulate useful information while working with various objects. This can be beneficial when traversing a complex object structure, such as an object tree, and applying the visitor to each object within the structure.

Disadvantages:

- Each time a class is added to or removed from the element hierarchy, you need to update all visitors.
- Visitors may lack access to the private fields and methods of the elements they are intended to work with.

7 Example

Suppose we have a program that represents a hierarchy of geometric shapes such as circles, rectangles, and triangles. We want to perform different operations on these shapes, such as calculating their area or perimeter, without modifying the shape classes themselves. Additionally, we may need to add new operations in the future.

Solution:
Step 1: Define the shape classes hierarchy.

```
// Step 1: Define the shape classes hierarchy
class Circle;
class Rectangle;
class Triangle;

class Shape {
    public:
        virtual void accept(class ShapeVisitor& visitor) = 0;
};

class Circle : public Shape {
    public:
        void accept(ShapeVisitor& visitor) override;
};

class Rectangle : public Shape {
    public:
        void accept(ShapeVisitor& visitor) override;
};

class Triangle : public Shape {
    public:
        void accept(ShapeVisitor& visitor) override;
};
```

Step 2: Define the visitor interface and its implementations.

```
// Step 2: Define the visitor interface and its implementations
class ShapeVisitor {
    public:
        virtual void visit(Circle& circle) = 0;
        virtual void visit(Rectangle& rectangle) = 0;
        virtual void visit(Triangle& triangle) = 0;
};

class AreaVisitor : public ShapeVisitor {
    public:
        void visit(Circle& circle) override;
        void visit(Rectangle& rectangle) override;
        void visit(Triangle& triangle) override;
};

class PerimeterVisitor : public ShapeVisitor {
    public:
```

```
        void visit(Circle& circle) override;
        void visit(Rectangle& rectangle) override;
        void visit(Triangle& triangle) override;
};
```

Step 3: Implement the shape classes' `accept()` method.

```
// Step 3: Implement the shape classes' accept() method
void Circle::accept(ShapeVisitor& visitor) {
    visitor.visit(*this);
}

void Rectangle::accept(ShapeVisitor& visitor) {
    visitor.visit(*this);
}

void Triangle::accept(ShapeVisitor& visitor) {
    visitor.visit(*this);
}
```

Step 4: Implement the visitor methods.

```
// Step 4: Implement the visitor methods
void AreaVisitor::visit(Circle& circle) {
    // Calculate area of circle
    // ...
}

void AreaVisitor::visit(Rectangle& rectangle) {
    // Calculate area of rectangle
    // ...
}

void AreaVisitor::visit(Triangle& triangle) {
    // Calculate area of triangle
    // ...
}

void PerimeterVisitor::visit(Circle& circle) {
    // Calculate perimeter of circle
    // ...
}

void PerimeterVisitor::visit(Rectangle& rectangle) {
    // Calculate perimeter of rectangle
    // ...
}

void PerimeterVisitor::visit(Triangle& triangle) {
    // Calculate perimeter of triangle
    // ...
}
```

Step 5: Usage example.

```
int main() {
    // Create shapes
    Circle circle;
    Rectangle rectangle;
    Triangle triangle;

    // Create visitors
    AreaVisitor areaVisitor;
    PerimeterVisitor perimeterVisitor;

    // Apply visitors to shapes
    circle.accept(areaVisitor);
    rectangle.accept(areaVisitor);
    triangle.accept(areaVisitor);

    circle.accept(perimeterVisitor);
    rectangle.accept(perimeterVisitor);
    triangle.accept(perimeterVisitor);

    return 0;
}
```

Bibliography

[1] *Design Patterns*. https://refactoring.guru/design-patterns.

[2] *Smart Contract*. https://www.ibm.com/topics/smart-contracts.