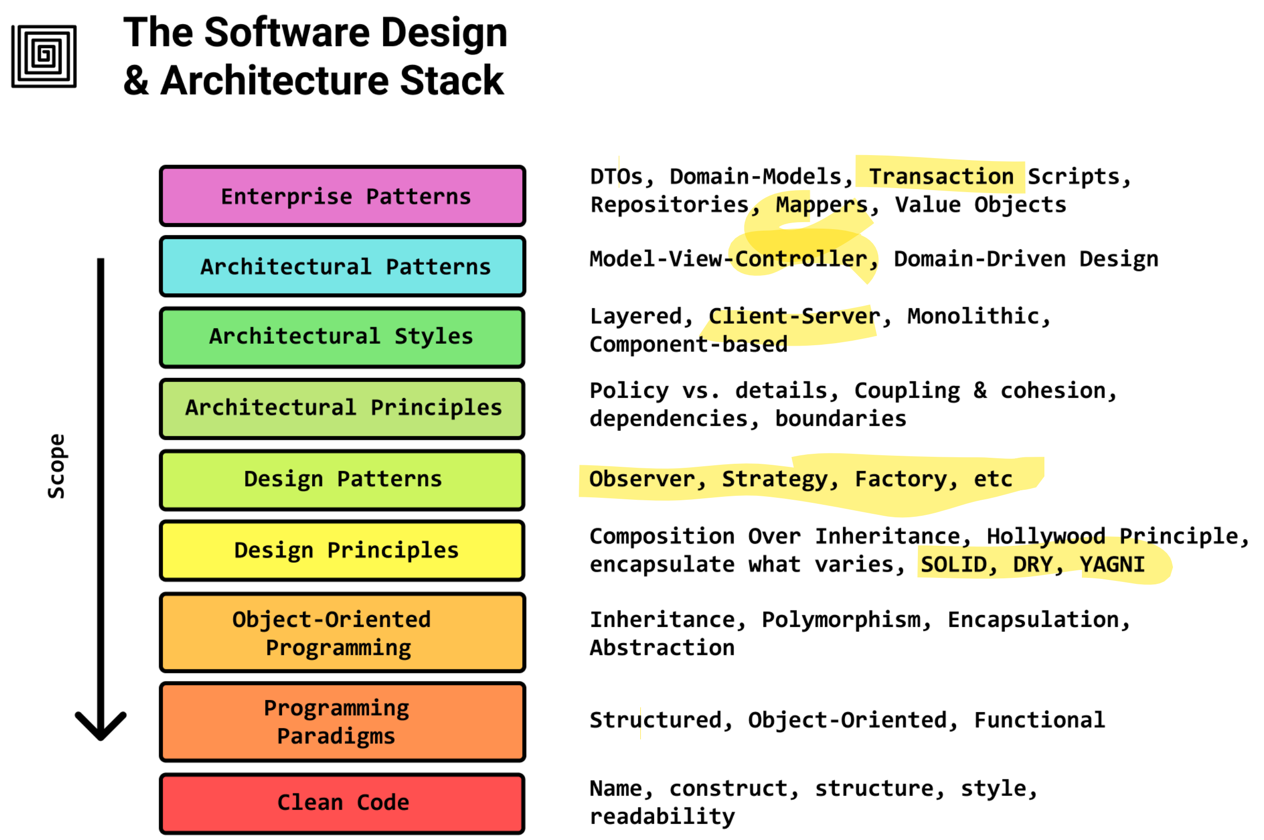
Software Design

By: Mohammad AbouElSherbini

Software Design and Architecture Stack



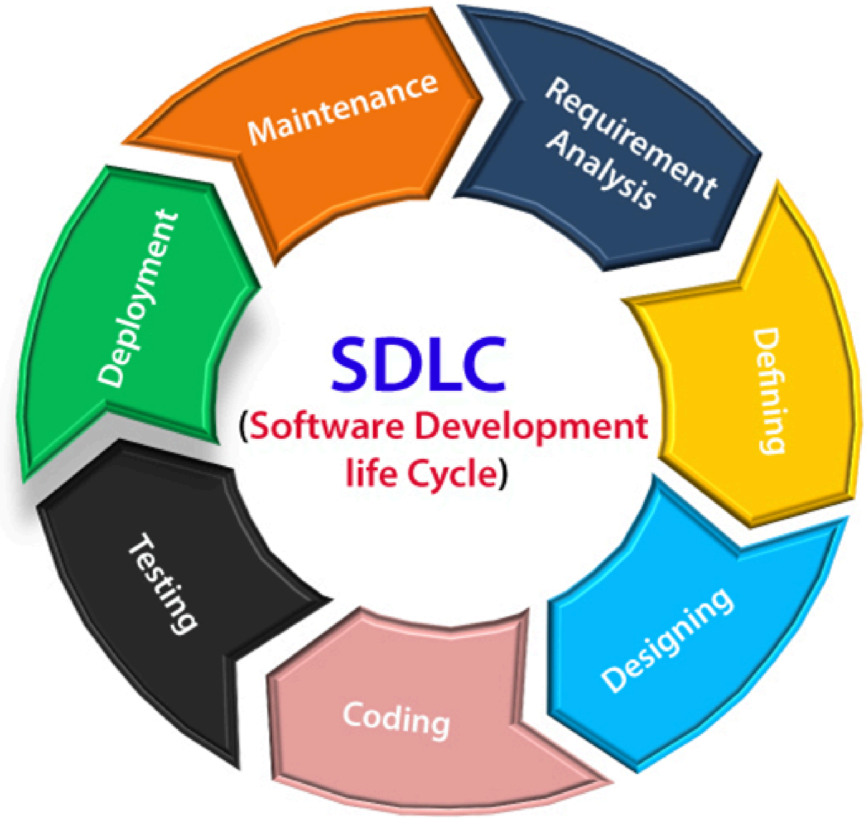
The Software Design & Architecture Stack

Enterprise Patterns	DTOs, Domain-Models, Transaction Scripts, Repositories, Mappers, Value Objects
Architectural Patterns	Model-View-Controller, Domain-Driven Design
Architectural Styles	Layered, Client-Server, Monolithic, Component-based
Architectural Principles	Policy vs. details, Coupling & cohesion, dependencies, boundaries
Design Patterns	Observer, Strategy, Factory, etc
Design Principles	Composition Over Inheritance, Hollywood Principle, encapsulate what varies, SOLID, DRY, YAGNI
Object-Oriented Programming	Inheritance, Polymorphism, Encapsulation, Abstraction
Programming Paradigms	Structured, Object-Oriented, Functional
Clean Code	Name, construct, structure, style, readability

- Similar to the OSI Model in networking, each layer builds on top of the foundation of the previous one.
- We refer to the higher level design decisions that are harder to change as the architecture of the software
- When designing a project we decide from top to bottom, but when learning we start from bottom to top

Software Development Life Cycle (SDLC)

Software development is a continuous cycle



Software rotting

What goes Wrong with Software over Time

- At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through.
- Yet, over time as the rotting continues, the ugly festering accumulate until they dominate the design of the application.
- The program becomes a festering mass of code that the developers find increasingly hard to maintain

Identifying a rotting system

- changes hard to apply?
- simple change impacts numerous modules?
- implementing simple changes takes forever?
- changing one place harms another?
- fixing a bug causes x others?
- modules are not reusable because of their dependencies?
- rewriting a code instead of reusing existing one?
- easier to do "hacks" than go "by the book"?
- environment is slow and inefficient?

Congrats your system is showing symptoms of rotting

How did we get here

Requirements change

When requirements change in ways that the initial design did not anticipate

- Often these changes need to be made quickly and may be made by engineers who are not familiar with the original design philosophy
- So, though the change to the design works, it somehow violates the original design
- Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in
- However, software is bound to keep changing so the design has to smoothly accommodate these changes

Environment change

When changes occur in the program's environment which the designer did not anticipate

- The software may no longer operate as originally intended
- For example, many early computer game designers used the CPU clock speed as a timer in their games (as frames increased so did the speed of the game making it unusable)

Improper dependency management

When the dependency architecture degrades, and with it the ability of the software to be maintained

- To stop this, the dependencies between modules in an application must have well defined boundaries

Main symptoms of rotting design

Rigidity

is the tendency for software to be difficult to change, even in simple ways

- Every change causes a cascade of subsequent changes in dependent modules
- What begins as a simple two-day change to one module grows into a multiweek marathon of change in module after module as the engineers chase the thread of the change through the application

Fragility

is the tendency of the software to break in many places every time it is changed

- Often the breakage occurs in areas that have no conceptual relationship with the area that was changed
- Every time we authorize a fix, we fear that the software will break in some unexpected way
- Every fix makes it worse, introducing more problems than are solved

Immobility

software that is hard to reuse

- When we need to use a module similar to an existing one but the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate
- We then have to rewrite the software instead of reusing making it immobile

Viscosity

the ease at which a developer can add design-preserving code to a system

- If it is easier to add a hack than it is to add code that fits into the program's design, then the system has high viscosity

- If it is easy to add new code to the program while maintaining the design, then the program has low viscosity

---

Design Principles

Guidelines that assist developers in creating a good system design that is clear, manageable, and scalable

---

Some of the key principles are:

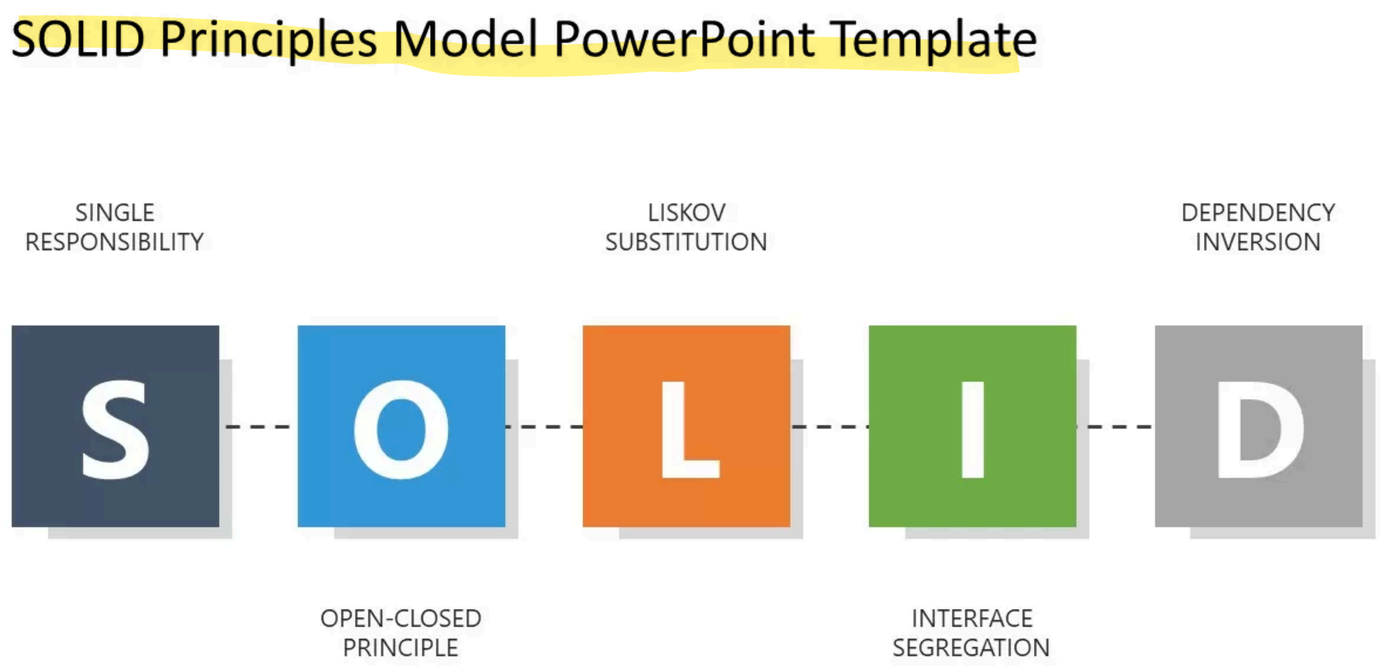
- SOLID (ensuring modular and flexible code)
- DRY (reducing code redundancy)
- KISS (keeping designs simple)
- YAGNI (avoiding unnecessary features)
- Favor composition over inheritance

Adhering to these principles improves code quality, simplifies maintenance, and increases the overall strength of software systems

---

1. SOLID

The most crucial one and consists of five basic designing principles

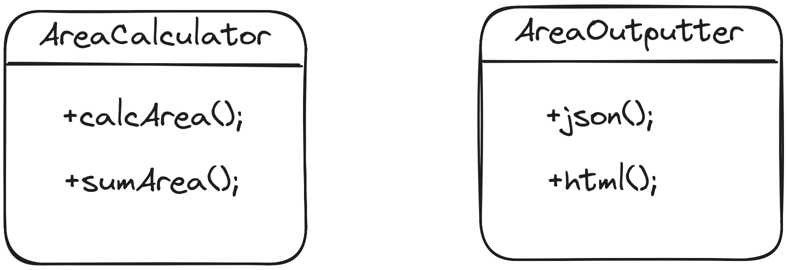


SOLID Principles Model PowerPoint Template

Single Responsibility Principle (SRP)

- This principle states that there should never be more than one reason for a class to change.
- This means that you should design your classes in such a way that each class should have a single purpose.

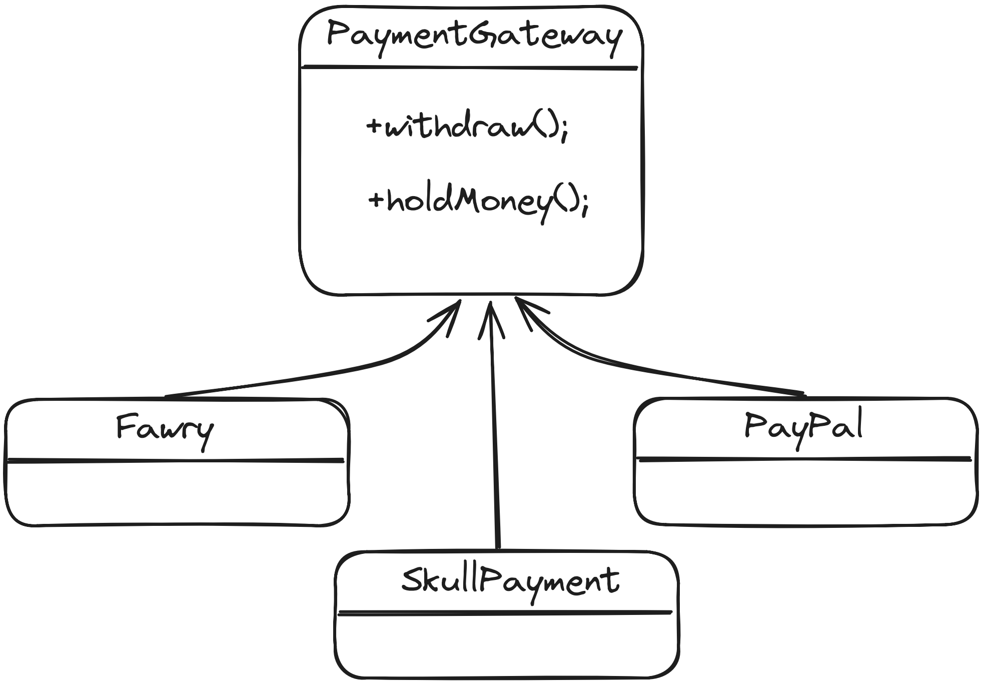
Example - An AreaCalculator class is only responsible for calculating the area, but outputting the values to different formats should be handled by another class. Hence, both are responsible for a single purpose only. Therefore, we are moving towards specialization.



Open/Closed Principle (OCP)

- This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- The "closed" part of the rule states that once a module has been developed and tested, the code should only be changed to correct bugs. The "open" part says that you should be able to extend existing code to introduce new functionality.

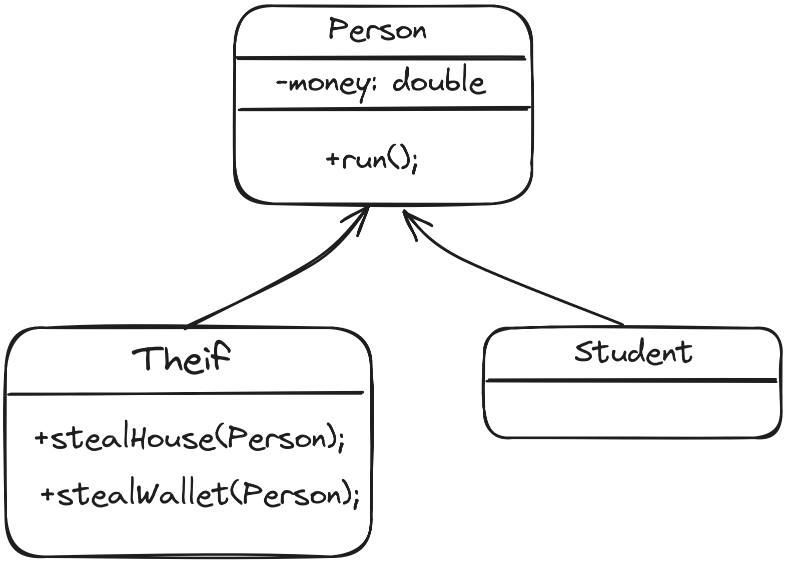
Example - A PaymentGateway base class contains all basic payment-related properties and methods. It can be extended by different PaymentGateway classes for different payment gateway vendors to achieve their functionalities. Hence, it is open for extension but closed for modification.



Liscov Substitution Principle (LSP)

- This principle states that functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Example - Assume that you have an inheritance hierarchy between Person and Student. Wherever you can use Person, you should also be able to use a Student because Student is a subclass of Person.

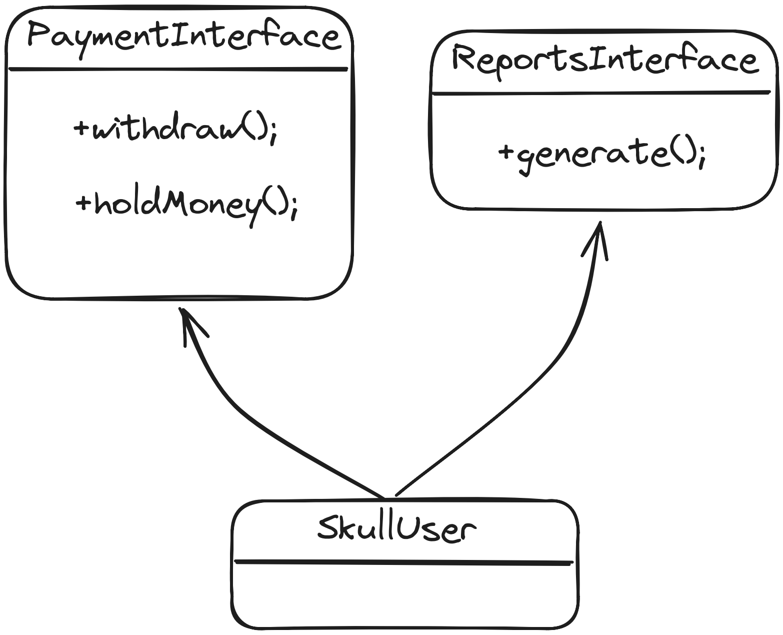


Bird -> FlyingBird ->Pigeon
Bird -> Ostrich

Interface Segregation Principle (ISP)

- This principle states that Clients should not be forced to depend upon interfaces that they don't use.
- This means minimizing the number of members in the interface visible to the dependent class.

Example - The service interface that is exposed to the client should contain only client-related methods, not all.

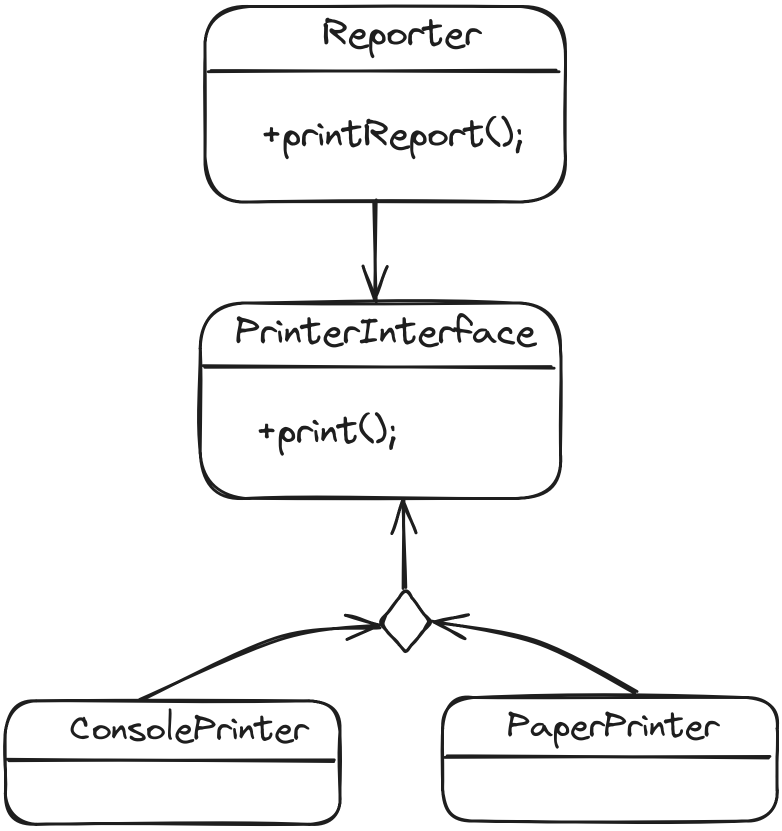


Dependency Inversion Principle (DIP)

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

It helps us to develop loosely coupled code by ensuring that high-level modules depend on abstractions rather than concrete implementations of lower-level modules. The Dependency Injection pattern is an implementation of this principle

Example - The Dependency Injection pattern is an implementation of this principle



2. DRY (Don’t Repeat Yourself)

DRY (Don't Repeat Yourself) is a fundamental software design approach that aims to reduce code redundancy.

- It underlines the importance of having a single, unambiguous representation of all information or logic within a system.
- The DRY concept helps to eliminate errors by reducing code duplication and consolidating comparable code into reusable functions or components.
- It also simplifies maintenance and makes the codebase more cohesive and manageable.

3. KISS (Keep it simple, Stupid!)

Keep It Simple, Stupid (KISS) is a software design guideline that encourages simplicity in code and design.

- It highlights the importance of keeping systems as simple as feasible while avoiding unnecessary complexity and over-engineering.
- The KISS principle encourages improved readability, easier maintenance, and a lower risk of errors, resulting in more resilient and understandable software.

4. YAGNI (You aren't going to need it)

You Aren't Gonna Need It (YAGNI) is a software design theory that recommends not implementing capability until it is genuinely needed.

- It highlights that developers should avoid adding features or capabilities based on fictional demands or future requirements.
- By following YAGNI, developers can simplify their codebase, prevent over-engineering, and focus on solving existing problems, resulting in more manageable and efficient code.

5. Favor composition over inheritance (IS-A vs HAS-A)

Favor composition over inheritance is a design principle that suggests it’s better to compose objects to achieve polymorphic behavior and code reuse rather than inheriting from a base class.

- Despite the benefits of inheritance, it has its downsides. A major one is the tight coupling it creates, making a system rigid and harder to modify and can lead to a confusing hierarchy when overused.
- With composition, you build complex objects by composing them of simple ones. Changing the behavior of a system involves changing the components, which are easier to manage than tangled inheritance hierarchies.

Design patterns

What to Expect from Design Patterns

Here are several ways in which the design patterns can affect the way you design object-oriented software.

A Common Design Vocabulary

Computer scientists name and catalog algorithms and data structures, but we don't often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives. Design patterns make a system seem less complex by letting you talk about it at a higher level of abstraction than that of a design notation or programming language. Design patterns raise the level at which you design and discuss design with your colleagues.

A Documentation and Learning Aid

- Knowing the design patterns makes it easier to understand existing systems. Most large object-oriented systems use these design patterns. People learning object-oriented programming often complain that the systems they're working with use inheritance in convoluted ways and that it's difficult to follow the flow of control. In large part this is because they do not understand the design patterns in the system. Learning these design patterns will help you understand existing object-oriented systems.

- These design patterns can also make you a better designer. They provide solutions to common problems. If you work with object-oriented systems long enough, you'll probably learn these design patterns on your own. Learning these patterns will help a novice act more like an expert.

- Describing a system in terms of the design patterns that it uses will make it a lot easier to understand. Otherwise, people will have to reverse-engineer the design to unearth the patterns it uses. Having a common vocabulary means you don't have to describe the whole design pattern; you can just name it and expect your reader to know it. A reader who doesn't know the patterns will have to look them up at first, but that's still easier than reverse-engineering.

A Target for Refactoring

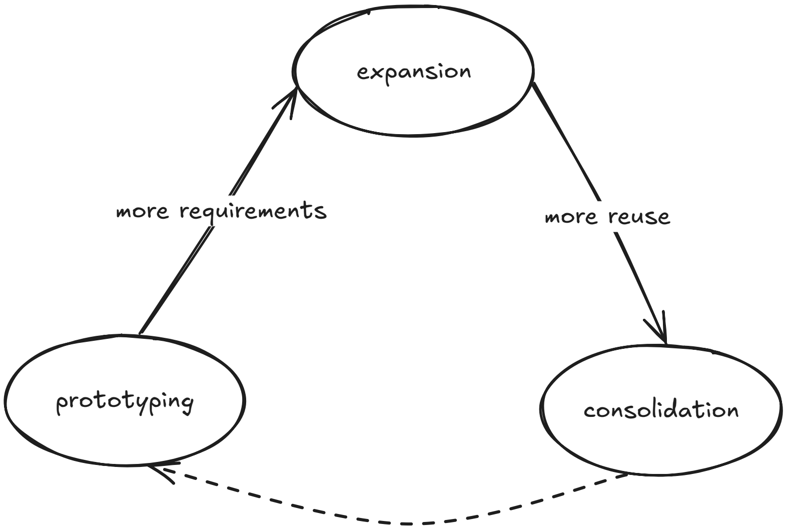
- One of the problems in developing reusable software is that it often has to be reorganized or **refactored.** Design patterns help you determine how to reorganize a design, and they can reduce the amount of refactoring you need to do later.
- The lifecycle of object-oriented software has several phases. Brian Foote identifies these phases as the **prototyping**, **expansionary**, and **consolidating** phases.

- The prototyping phase is a flurry of activity as the software is brought to life through rapid prototyping and incremental changes, until it meets an initial set of requirements and reaches adolescence. At this point, the software usually consists of class hierarchies that closely reflect entities in the initial problem domain. The main kind of reuse is white-box reuse by inheritance.

- Once the software has reached adolescence and is put into service, its evolution is governed by two conflicting needs:
  1. the software must satisfy more requirements, and;
  2. the software must be more reusable.

New requirements usually add new classes and operations and perhaps whole class hierarchies. The software goes through an expansionary phase to meet new requirements. This can't continue for long, however. Eventually the software will become too inflexible and arthritic for further change. The class hierarchies will no longer match any problem domain. Instead they'll reflect many problem domains, and classes will define many unrelated operations and instance variables.

- To continue to evolve, the software must be reorganized in a process known as *refactoring*. This is the phase in which frameworks often emerge. Refactoring involves tearing apart classes into special- and general-purpose components, moving operations up or down the class hierarchy, and rationalizing the interfaces of classes. This consolidation phase produces many new kinds of objects, often by decomposing existing objects and using object composition instead of inheritance. Hence black-box reuse replaces white-box reuse. The continual need to satisfy more requirements along with the need for more reuse propels object-oriented software through repeated phases of expansion and consolidation—expansion as new requirements are satisfied, and consolidation as the software becomes more general.



- This cycle is unavoidable. But good designers are aware of the changes that can prompt refactorings. Good designers also know class and object structures that can help avoid refactorings—their designs are robust in the face of requirement changes. A thorough requirements analysis will highlight those requirements that are likely to change during the life of the software, and a good design will be robust to them.

- Our design patterns capture many of the structures that result from refactoring. Using these patterns early in the life of a design prevents later refactorings. But even if you don't see how to apply a pattern until after you've built your system, the pattern can still show you how to change it. Design patterns thus provide targets for your refactorings.

Types of Design Patterns

- **Creational Patterns**: Concerned with object creation mechanisms, trying to create objects in a manner suitable for the situation.
- **Structural Patterns**: Deal with object composition, typically how classes and objects are composed to form larger structures.
- **Behavioral Patterns:** Focus on communication between objects, defining how they interact and communicate.

Creational Design Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.

1. A **class creational** pattern uses inheritance to vary the class that's instantiated, whereas;
2. an **object creational** pattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be static (that is, specified at compile-time) or dynamic (at run-time).

Singleton

(Object Creational)

Intent

Ensure a class only has one instance, and provide a global point of access to it

---

Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A **global variable** makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Applicability

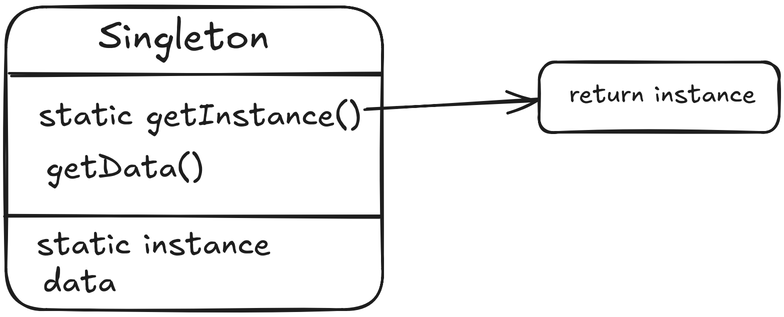
Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

---

Structure



Participants

1. Singleton
   - defines an Instance operation that lets clients access its unique instance.
   - may be responsible for creating its own unique instance

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation

---

Consequences

The Singleton pattern has several benefits:

1. **Controlled access to sole instance**. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. **Reduced name space**. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

---

3. **Permits refinement of operations and representation**. The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

```
MySingleton::MySingleton() {
        Singleton::Register("MySingleton", this);
        // ...
}

Singleton* Singleton::getInstance () {
        if (_instance == 0) {
                const char* singletonName = getenv("SINGLETON");
                // user or environment supplies this at startup

                _instance = Lookup(singletonName);
                // Lookup returns 0 if there's no such singleton
        }
        return _instance;
}
```

4. **Permits a variable number of instances**. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. **More flexible than class operations**. Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++). But this language technique makes it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

Implementation

Here are implementation issues to consider when using the Singleton pattern:

1. **Ensuring a unique instance**. The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation (that is, either a static member function or a class method) that guarantees only one instance is created.

This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a singleton is created and initialized before its first use.
You can define the class operation with a static member function Instance of the Singleton class. Singleton also defines a static member variable `instance` that contains a pointer to its unique instance.

2. **Instantiation costs**. There are 2 ways of instantiation

```
1. **Eager Instantiation:** creation of instance at load time
```

```
class Singleton {
        private static Singleton instance = new Singleton();
        private Singleton(){}
        public static Singleton getInstance(){
                        return instance;
        }
};
```

```
2. **Lazy Instantiation:** creation of instance when first required
```

```
class Singleton {
        private static Singleton instance;
        private Singleton(){}

        public static Singleton getInstance() {
                if (instance == null) {
                        synchronized(Singleton.class){
                                if (instance == null){
                                        instance = new Singleton();
                                }
                   }
                }
          return instance;
        }
};
```

```
Notice that the constructor is private. A client that tries to instantiate
Singleton directly will get an error at compile-time. This ensures that only one
instance can ever get created.
```

3. **Handling the serialization**. If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

   To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

   ```
    class Singleton implements Serializable {
           protected Object readResolve() {
           return getInstance();
        }
   };
   ```

Related Patterns

Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder, and Prototype.

Factory Method

(Class Creational)

also known as: Virtual Constructor

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

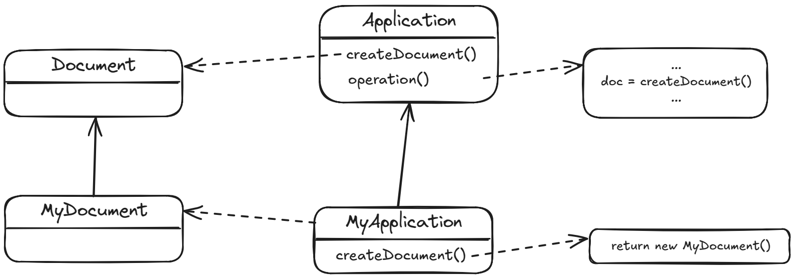
Motivation

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations. To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.

Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when* a new document should be created, not *what kind* of Document to create. This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

The Factory Method pattern offers a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.



Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call CreateDocument a **factory method** because it's responsible for "manufacturing" an object.

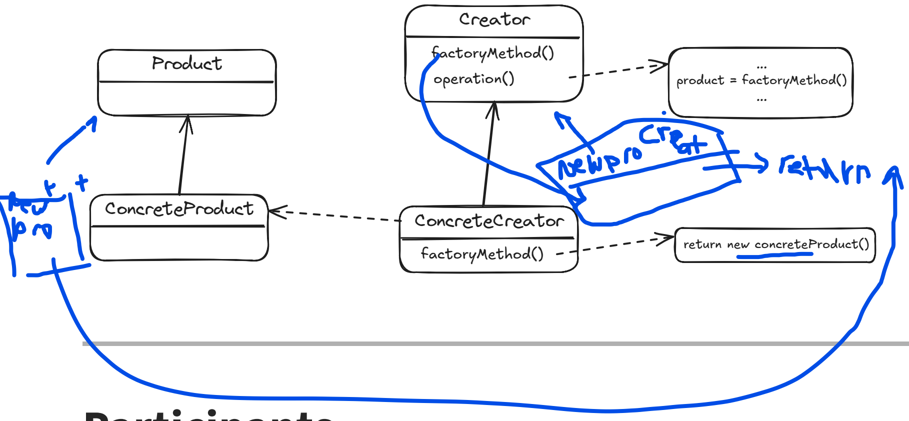
Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.

- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants

1. Product (Document)
   - defines the interface of objects the factory method creates.
2. ConcreteProduct (MyDocument)
   - implements the Product interface.
3. Creator (Application)
   - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
   - may call the factory method to create a Product object.
4. ConcreteCreator (MyApplication)
   - overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

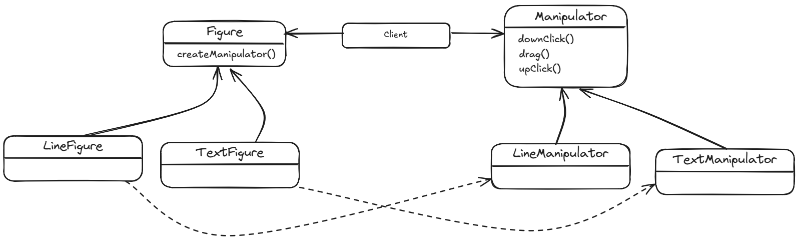
Here are two additional consequences of the Factory Method pattern:

1. **Provides hooks for subclasses.** Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.
   In the Document example, the Document class could define a factory method called CreateFileDialog that creates a default file dialog object for opening an existing document. A Document subclass can define an application-specific file dialog by overriding this factory method. In this case the factory method is not abstract but provides a reasonable default implementation.

---

2. **Connects parallel class hierarchies.** In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in the case of parallel class hierarchies.
   Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class. Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse. Implementing such interactions isn't always easy. It often requires storing and updating information that records the state of the manipulation at a given time. This state is needed only during manipulation; therefore it needn't be kept in the figure object. Moreover, different figures behave differently when the user manipulates them. For example, stretching a line figure might have the effect of moving an endpoint, whereas stretching a text figure may change its line spacing.

With these constraints, it's better to use a separate Manipulator object that implements the interaction and keeps track of any manipulation-specific state that's needed. Different figures will use different Manipulator subclasses to handle particular interactions. The resulting Manipulator class hierarchy parallels (at least partially) the Figure class hierarchy:



The Figure class provides a CreateManipulator factory method that lets clients create a Figure's corresponding Manipulator. Figure subclasses override this method to return an instance of the Manipulator subclass that's right for them. Alternatively, the Figure class may implement CreateManipulator to return a default Manipulator instance, and Figure subclasses may simply inherit that default. The Figure classes that do so need no corresponding Manipulator subclass—hence the hierarchies are only partially parallel.
Notice how the factory method defines the connection between the two class hierarchies. It localizes knowledge of which classes belong together.

Implementation

Consider the following issues when applying the Factory Method pattern:

1. **Two major varieties.** The two main variations of the Factory Method pattern are
    1. the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and;
    2. the case when the Creator is a concrete class and provides a default implementation for the factory method. It's also possible to have an abstract class that defines a default implementation, but this is less common.

```
The first case requires subclasses to define an implementation, because there's
no reasonable default. It gets around the dilemma of having to instantiate
unforeseeable classes. In the second case, the concrete Creator uses the factory
method primarily for flexibility. It's following a rule that says, "Create
objects in a separate operation so that subclasses can override the way they're
created." This rule ensures that designers of subclasses can change the class of
objects their parent class instantiates if necessary.
```

---

2. **Parameterized factory methods.** Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface. In the Document example, Application might support different kinds of Documents. You pass Create- Document an extra parameter to specify the kind of document to create.
   A parameterized factory method has the following general form, where ProductA and ProductB are subclasses of Product:

```
class Creator {
        Product create(ProductType type) {
                if (type == ProductType.TypeA) return new ProductA();
                if (type == ProductType.TypeB) return new ProductB();
                // repeat for remaining products...
        }
};
```

Overriding a parameterized factory method lets you easily and selectively extend or change the products that a Creator produces. You can introduce new identifiers for new kinds of products, or you can associate existing identifiers with different products.
For example, a subclass MyCreator could swap ProductA and ProductB and support a new ProductC subclass:

```
class Creator {
        Product create(ProductType type) {
                if (type == ProductType.TypeA) return new ProductB();
                if (type == ProductType.TypeB) return new ProductA();
                if (type == ProductType.TypeC) return new ProductC();

                return super.create();
        }
};
```

Notice that the last thing this operation does is call Create on the parent class. That's because `MyCreator::Create` handles only ProductA, ProductB, and ProductC differently than the parent class. It isn't interested in other classes.
Hence MyCreator extends the kinds of products created, and it defers responsibility for creating all but a few products to its parent.

3. **Naming conventions.** It's good practice to use naming conventions that make it clear you're using factory methods. For example, the MacApp Macintosh application framework always declares the abstract operation that defines the factory method as Class* DoMakeClass ( ) , where Class is the Product class.

Related Patterns

- Abstract Factory is often implemented with factory methods.
- Factory methods are usually called within Template Methods. In the Structure example above, `operation()` is a template method.
- Prototypes don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

Abstract Factory

(Object Creational)

also known as: Kit

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

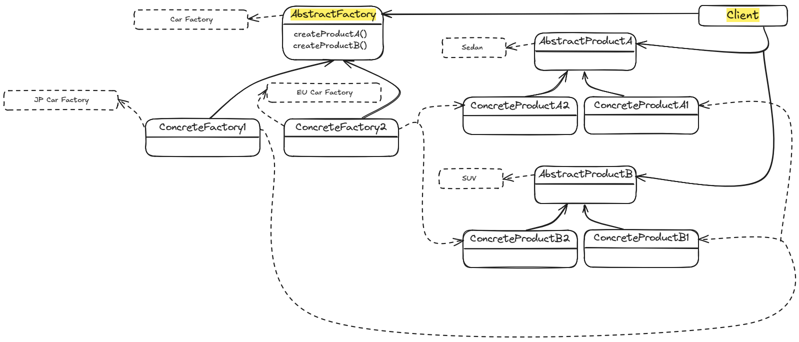
Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.

- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure



Participants

1. AbstractFactory (CarFactory)
   - declares an interface for operations that create abstract product objects(families).
2. ConcreteFactory (JP Car Factory, EU Car Factory)
   - implements the operations to create concrete product objects.
3. AbstractProduct (Sedan, SUV)
   - declares an interface for a type of product object.
4. ConcreteProduct (EU Sedan, EU SUV, JP Sedan, JP SUV)
   - defines a product object to be created by the corresponding concrete factory.
   - implements the AbstractProduct interface.
5. Client
   - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects(families), clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

1. **It isolates concrete classes.** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

---

2. **It makes exchanging product families easy.** The class of a concrete factory appears only once in an application—that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once. In our car example, we can switch from JP cars to EU cars simply by switching the corresponding factory objects and recreating the interface.

---

3. **It promotes consistency among products.** When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.
4. **Supporting new kinds of products is difficult.** Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses. We discuss one solution to this problem in the Implementation section.

Implementation

Here are some useful techniques for implementing the Abstract Factory pattern.

1. **Factories as singletons.** An application typically needs only one instance of a ConcreteFactory per product family. So it's usually best implemented as a Singleton.

---

2. **Creating the products.** AbstractFactory only declares an interface for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method (see Factory Method) for each product. A concrete factory will specify its products by overriding the factory method for each. While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.
   If many product families are possible, the concrete factory can be implemented using the Prototype pattern. The concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.

Here's a way to implement a Prototype-based factory in Java. The concrete factory stores the prototypes to be cloned in a dictionary called `prototypes`. The method `createPrototype` retrieves the prototype and clones it:

```
public class PrototypeFactory {
    private Map<String, Prototype> prototypes = new HashMap<>();

    public void addPrototype(String key, Prototype prototype) {
        prototypes.put(key, prototype);
    }

    public Prototype createPrototype(String key) {
        Prototype prototype = prototypes.get(key);
        if (prototype != null) {
            return prototype.clone();
        }
        throw new IllegalArgumentException("Prototype not found for key: " +
key);
    }
}
```

---

3. **Defining extensible factories.** AbstractFactory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it. A more flexible but less safe design is to add a parameter to operations that create objects. This parameter specifies the kind of object to be created. It could be a class identifier, an integer, a string, or anything else that identifies the kind of product. In fact with this approach, AbstractFactory only needs a single "Make" operation with a parameter indicating the kind of object to create. This is the technique used in the Prototype- and the class-based abstract factories discussed earlier.

This variation is easier to use in a dynamically typed language like Smalltalk than in a statically typed language like C++. You can use it in C++ only when all objects have the same abstract base class or when the product objects can be safely coerced to the correct type by the client that requested them. The implementation section of Factory Method shows how to implement such parameterized operations in C++.

But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the same abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass- specific operations, they won't be accessible through the abstract interface. Although the client could perform a downcast (e.g., with dynamic-cast in C++), that's not always feasible or safe, because the downcast can fail. This is the classic trade-off for a highly flexible and extensible interface.

Related Patterns

- AbstractFactory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype.
- A concrete factory is often a singleton (Singleton).

Builder

(Object Creational)

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

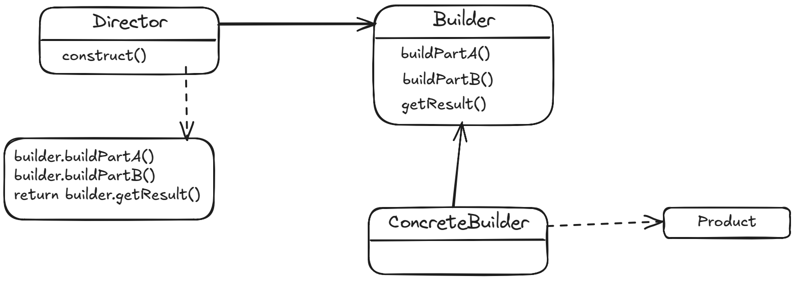
Motivation

Applicability

Use the Builder pattern when

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.
- the constructed object needs to be immutable

Structure



Participants

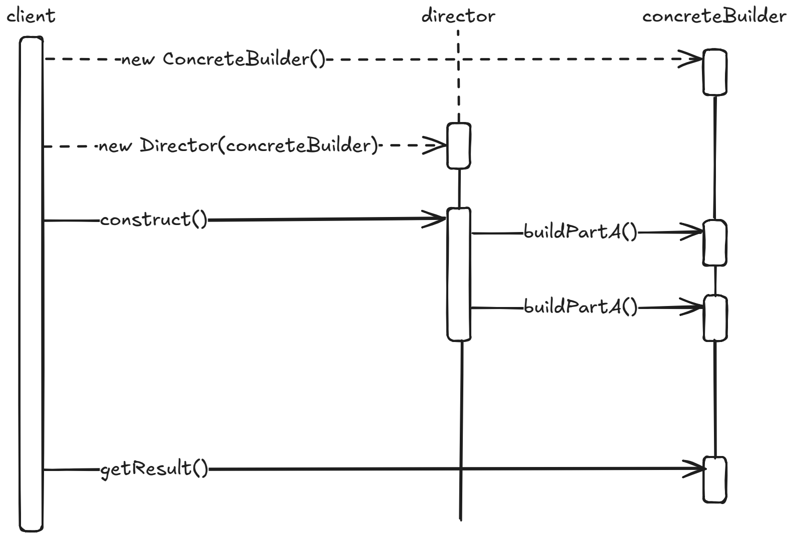
1. Builder (TextConverter)
    - specifies an abstract interface for creating parts of a Product object.
2. ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)
    - constructs and assembles parts of the product by implementing the Builder interface.
    - defines and keeps track of the representation it creates.
    - provides an interface for retrieving the product (e.g., GetASCIIText, Get-Text Widget).
3. Director (RTFReader)
    - constructs an object using the Builder interface.
4. Product (ASCIIText, TeXText, TextWidget)

- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

The following interaction diagram illustrates how Builder and Director cooperate with a client.



Consequences

Here are key consequences of the Builder pattern:

1. **It lets you vary a product's internal representation**. The Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled. Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder.

2. **It isolates code for construction and representation**. The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.
Each ConcreteBuilder contains all the code to create and assemble a particular kind of product. The

code is written once; then different Directors can reuse it to build Product variants from the same set of parts. In the earlier RTF example, we could define a reader for a format other than RTF, say, an SGMLReader, and use the same TextConverters to generate ASCIIText, TeXText, and TextWidget renditions of SGML documents.

3. **It gives you finer control over the construction process**. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder. Hence the Builder interface reflects the process of constructing the product more than other creational patterns. This gives you finer control over the construction process and consequently the internal structure of the resulting product.

Implementation

Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default.
A ConcreteBuilder class overrides operations for components it's interested in creating.

Here are other implementation issues to consider:

1. **Assembly and construction interface.** Builders construct their products in step- by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A key design issue concerns the model for the construction and assembly process. A model where the results of construction requests are simply appended to the product is usually sufficient. In the RTF example, the builder converts and appends the next token to the text it has converted so far. But sometimes you might need access to parts of the product constructed earlier. In the Maze example, the MazeBuilder interface lets you add a door between existing rooms. Tree structures such as parse trees that are built bottom-up are another example. In that case, the builder would return child nodes to the director, which then would pass them back to the builder to build the parent nodes.

2. **Why no abstract class for products**? (put build method in interface or not). In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class. In the RTF example, the ASCIIText and the TextWidget objects are unlikely to have a common interface, nor do they need one. Because the client usually configures the director with the proper concrete builder, the client is in a position to know which concrete subclass of Builder is in use and can handle its products accordingly.

3. **Empty methods as default in Builder**. In C++, the build methods are intentionally not declared pure virtual member functions. They're defined as empty methods instead, letting clients override only the operations they're interested in.

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
        builder.BuildMaze();

        builder.BuildRoom(1);
        buiIder.BuiIdRoom(2);
        builder.BuildDoor(1, 2);

        return builder.GetMaze();
}

void StandardMazeBuilder::BuildRoom (int n) {
        if (!_currentMaze->RoomNo(n)) {
                Room* room - new Room(n);
                _currentMaze->AddRoom(room);

                room->SetSide(North, new Wall());
                room->SetSide(South, new Wall());
                room->SetSide(East, new Wall());
                room->SetSide(West, new Wall());
        }
}
```

Related Patterns

- Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.
- A Composite is what the builder often builds.

Prototype

(Object Creational)

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

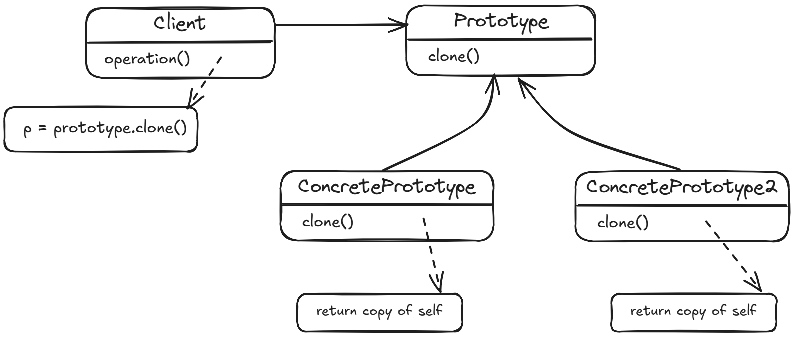
Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or

- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Structure



Participants

1. Prototype (Graphic)
    - declares an interface for cloning itself.
2. ConcretePrototype (Staff, WholeNote, HalfNote)
    - implements an operation for cloning itself.
3. Client (GraphicTool)
    - creates a new object by asking a prototype to clone itself.

Collaborations

- A client asks a prototype to clone itself.

Consequences

Prototype has many of the same consequences that Abstract Factory and Builder have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

Additional benefits of the Prototype pattern are listed below.

1. **Adding and removing products at runtime**. Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at runtime.

2. **Specifying new objects by varying values**. Highly dynamic systems let you define new behavior through object composition—by specifying values for an object's variables, for example—and not by defining new classes. You effectively define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype.
   This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs. In our music editor, one GraphicTool class can create a limitless variety of objects.

3. **Specifying new objects by varying structure.** Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits. For convenience, such applications often let you instantiate complex, user-defined structures, say, to use a specific subcircuit again and again.
   The Prototype pattern supports this as well. We simply add this subcircuit as a prototype to the palette of available circuit elements. As long as the composite circuit object implements Clone as a deep copy, circuits with different structures can be prototypes.

4. **Reduced subclassing.** Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects. Languages that do, like Smalltalk and Objective C, derive less benefit, since you can always use a class object as a creator. Class objects already act like prototypes in these languages.

5. **Configuring an application with classes dynamically**. Some run-time environments let you load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++.
   An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager (see the Implementation section). Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. For example, adding Clone is difficult when the classes under consideration already exist. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references

Implementation

Prototype is particularly useful with static languages like C++, where classes are not objects, and little or no type information is available at run-time. It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class. This pattern is built into prototype-based languages like Self, in which all object creation happens by cloning a prototype.

Consider the following issues when implementing prototypes:

1. **Using a prototype manager**. When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a prototype manager. A prototype manager is an associative store that returns the prototype matching a given key. It has operations for registering a prototype under a key and for unregistering it. Clients can change or even browse through the registry at run-time. This lets clients extend and take inventory on the system without writing code.

---

2. **Implementing the Clone operation**. The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references. Most languages provide some support for cloning objects. For example, Smalltalk provides an implementation of copy that's inherited by all sub-classes of Object. C++ provides a copy constructor. But these facilities don't solve the "shallow copy versus deep copy" problem. That is, does cloning an object in turn clone its instance variables, or do the clone and original just share the variables?

A shallow copy is simple and often sufficient, and that's what Smalltalk provides by default. The default copy constructor in C++ does a member-wise copy, which means pointers will be shared between the copy and the original. But cloning prototypes with complex structures usually requires a deep copy, because the clone and the original must be independent. Therefore you must ensure that the clone's components are clones of the prototype's components. Cloning forces you to decide what if anything will be shared.
If objects in the system provide Save and Load operations, then you can use them to provide a default implementation of Clone simply by saving the object and loading it back immediately. The Save operation saves the object into a memory buffer, and Load creates a duplicate by reconstructing the object from the buffer.

---

3. **Initializing clones**. While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing. You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any. Passing parameters in the Clone operation precludes a uniform cloning interface.

It might be the case that your prototype classes already define operations for (re)setting key pieces of state. If so, clients may use these operations immediately after cloning. If not, then you may have to introduce an Initialize operation that takes initialization parameters as arguments and sets the clone's internal state accordingly. Beware of deep-copying Clone operations—the copies may have to be deleted (either explicitly or within Initialize) before you reinitialize them (to avoid memory leaks).

Related Patterns

- Prototype and Abstract Factory are competing patterns in some ways. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. (**point no.3 in Consequences**)

Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations.

Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

Adapter

(Class, Object Structural)

also known as: Wrapper

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

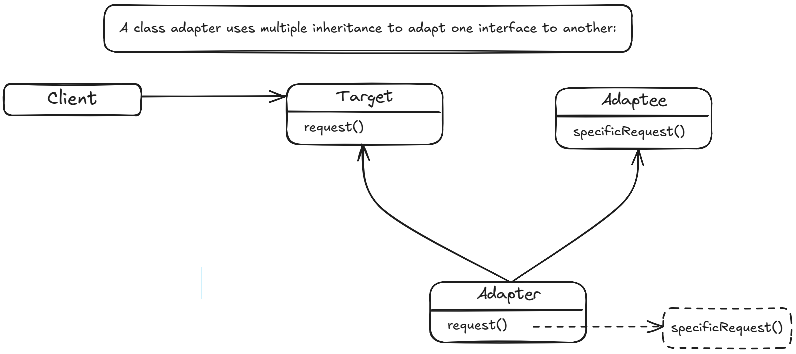
Applicability

Use the Adapter pattern when

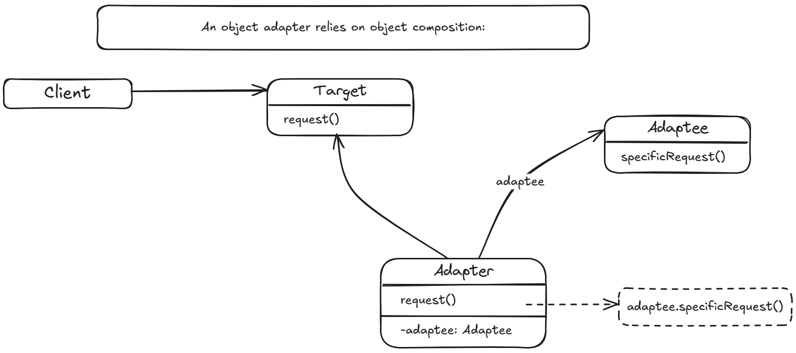
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure

Class adapter:



Object adapter:



Participants

1. **Target** (Shape)
   - defines the domain-specific interface that Client uses.
2. **Client** (DrawingEditor)
   - collaborates with objects conforming to the Target interface.
3. **Adaptee** (TextView)
   - defines an existing interface that needs adapting.
4. **Adapter** (TextShape)
   - adapts the interface of Adaptee to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

Class and object adapters have different trade-offs.

A class adapter

- adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Here are other issues to consider when using the Adapter pattern:

1. **How much adapting does Adapter do?** Adapters vary in the amount of work they do to adapt Adaptee to the Target interface. There is a spectrum of possible work, from simple interface conversion—for example, changing the names of operations—to supporting an entirely different set of operations. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

2. **Pluggable adapters.** A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface. Put another way, interface adaptation lets us incorporate our class into existing systems that might expect different interfaces to the class. Smalltalk uses the term **pluggable adapter** to describe classes with built-in interface adaptation.
   Consider a TreeDisplay widget that can display tree structures graphically. If this were a special-purpose widget for use in just one application, then we might require the objects that it displays to have a specific interface; that is, all must descend from a Tree abstract class. But if we wanted to make TreeDisplay more reusable (say we wanted to make it part of a toolkit of useful widgets), then that requirement would be unreasonable. Applications will define their own classes for tree structures. They shouldn't be forced to use our Tree abstract class. Different tree structures will have different interfaces. In a directory hierarchy, for example, children might be accessed with a GetSubdirectories operation, whereas in an inheritance hierarchy, the corresponding operation might be called GetSubclasses. A reusable TreeDisplay widget must be able to display both kinds of hierarchies even if they use different interfaces. In other words, the TreeDisplay should have interface adaptation built into it.
   We'll look at different ways to build interface adaptation into classes in the Implementation section.

3. **Using two-way adapters to provide transparency.** A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be

used as is wherever an Adaptee object can. **Two-way adapters** can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

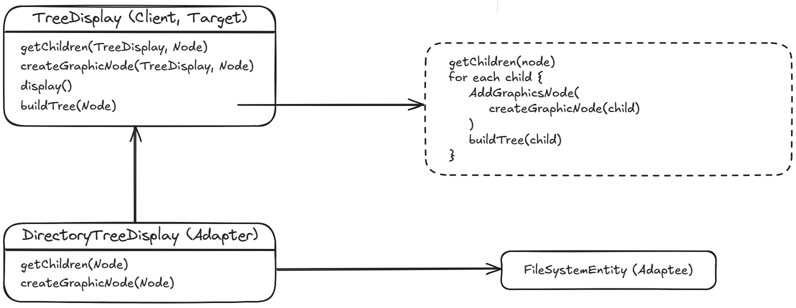
Implementation

Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

1. **Implementing class adapters in C++.** In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

2. **Pluggable adapters.** Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically.
The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation. A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations. For TreeDisplay, the adaptee is any hierarchical structure. A minimalist interface might include two operations, one that defines how to present a node in the hierarchical structure graphically, and another that retrieves the node's children.

The narrow interface leads to three implementation approaches:
1. **Using abstract operations.** Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object. For example, a DirectoryTreeDisplay subclass will implement these operations by accessing the directory structure.
DirectoryTreeDisplay specializes the narrow interface so that it can display directory structures made up of FileSystemEntity objects.


TreeDisplay (Client, Target)
getChildren(TreeDisplay, Node)
createGraphicNode(TreeDisplay, Node)
display()
buildTree(Node)
DirectoryTreeDisplay (Adapter)
getChildren(Node)
createGraphicNode(Node)
FileSystemEntity (Adaptee)

```
getChildren(node)
for each child {
    AddGraphicsNode(
        createGraphicNode(child)
    )
    buildTree(child)
}
```

```
2. **Using delegate objects.** In this approach, TreeDisplay forwards requests
for accessing the hierarchical structure to a **delegate** object. TreeDisplay
can use a different adaptation strategy by substituting a different delegate.
   For example, suppose there exists a DirectoryBrowser that uses a TreeDisplay.
DirectoryBrowser might make a good delegate for adapting TreeDisplay to the
hierarchical directory structure. In dynamically typed languages like Smalltalk
```

```
or Objective C, this approach only requires an interface for registering the
delegate with the adapter. Then TreeDisplay simply forwards the requests to the
delegate. NEXTSTEP uses this approach heavily to reduce subclassing.
   Statically typed languages like C++ require an explicit interface definition
for the delegate. We can specify such an interface by putting the narrow
interface that TreeDisplay requires into an abstract TreeAccessorDelegate class.
Then we can mix this interface into the delegate of our choice—DirectoryBrowser
in this case—using inheritance. We use single inheritance if the
DirectoryBrowser has no existing parent class, multiple inheritance if it does.
Mixing classes together like this is easier than introducing a new TreeDisplay
subclass and implementing its operations individually.
   ![[Software Design 2024-08-13 10.44.49.excalidraw]]
```

```
3. **Parameterized adapters.** The usual way to support pluggable adapters in
Smalltalk is to parameterize an adapter with one or more blocks. The block
construct supports adaptation without subclassing. A block can adapt a request,
and the adapter can store a block for each individual request. In our example,
this means TreeDisplay stores one block for converting a node into a GraphicNode
and another block for accessing a node's children.
   For example, to create TreeDisplay on a directory hierarchy, we write
```

```
        public TreeDisplay(
                Function<TreeNode, GraphicNode> createGraphicNodeBlock,
                Function<TreeNode, List<TreeNode>> getChildrenBlock) {
                        this.nodeConverter = nodeConverter;
this.childrenAccessor = childrenAccessor;
         }

        TreeDisplay display = new TreeDisplay(
                node -> new GraphicNode("Node: " + node.getValue()), //
ConvertTreeNode to GraphicNode
                TreeNode::getChildren // Access children of TreeNode
        );
```

Related Patterns

Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object.

Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

Proxy defines a representative or surrogate for another object and does not change its interface

Bridge

(Object Structural)

also known as: Handle/Body

Intent

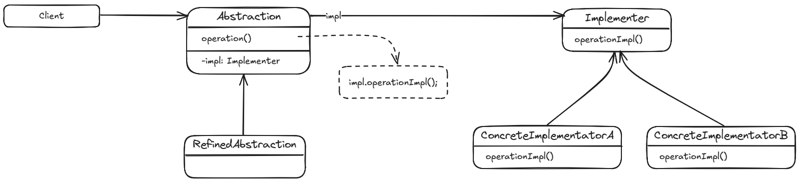
Decouple an abstraction from its implementation so that the two can vary independently.

Applicability

Use the Bridge pattern when

- you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
- you have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts. Rumbaugh uses the term "nested generalizations" to refer to such class hierarchies.
- you want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class, in which multiple objects can share the same string representation (StringRep).

Structure



Participants

Collaborations

Consequences

Implementation

Related Patterns

An Abstract Factory can create and configure a particular Bridge.
The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Composite

(Object Structural)

Intent

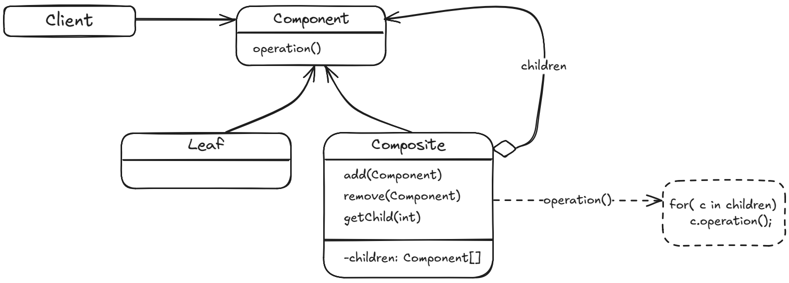
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Applicability

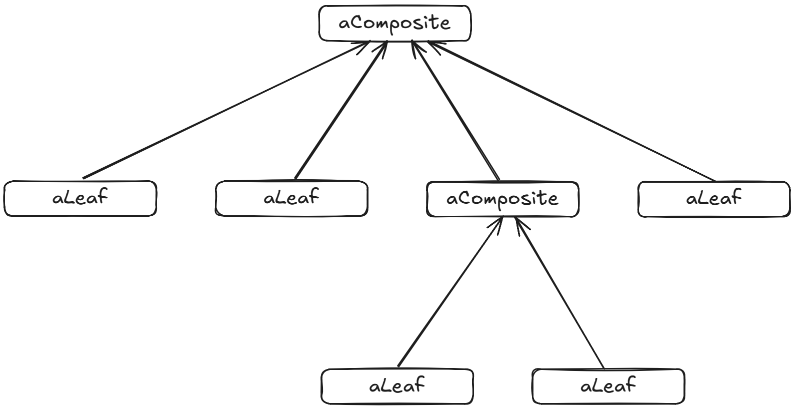
Use the Composite pattern when

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure



A typical Composite object structure might look like this:



Participants

1. **Component** (Graphic)
   - declares the interface for objects in the composition.
   - implements default behavior for the interface common to all classes, as appropriate.
   - declares an interface for accessing and managing its child components.
   - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
2. **Leaf** (Rectangle, Line, Text, etc.)
   - represents leaf objects in the composition. A leaf has no children.
   - defines behavior for primitive objects in the composition.
3. **Composite** (Picture)
   - defines behavior for components having children.
   - stores child components.
   - implements child-related operations in the Component interface.
4. **Client**
   - manipulates objects in the composition through the Component interface.

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

The Composite pattern

- **defines class hierarchies consisting of primitive objects and composite objects.** Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

- **makes the client simple.** Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

- **makes it easier to add new kinds of components.** Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
- **can make your design overly general.** The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Implementation

There are many issues to consider when implementing the Composite pattern:

1. **Explicit parent references**. Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility pattern.
   The usual place to define the parent reference is in the Component class. Leaf and Composite classes can inherit the reference and the operations that manage it.
   With parent references, it's essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children. The easiest way to ensure this is to change a component's parent *only* when it's being added or removed from a composite. If this can be implemented once in the Add and Remove operations of the Composite class, then it can be inherited by all the subclasses, and the invariant will be maintained automatically.

---

2. **Sharing components**. It's often useful to share components, for example, to reduce storage requirements. But when a component can have no more than one parent, sharing components becomes difficult.
   A possible solution is for children to store multiple parents. But that can lead to ambiguities as a request propagates up the structure. The Flyweight pattern shows how to rework a design to avoid storing parents altogether. It works in cases where children can avoid sending parent requests by externalizing some or all of their state.

---

3. **Maximizing the Component interface**. One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible. The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.

However, this goal will sometimes conflict with the principle of class hierarchy design that says a class should only define operations that are meaningful to its subclasses. There are many operations that Component supports that don't seem to make sense for Leaf classes. How can Component provide a default implementation for them?

Sometimes a little creativity shows how an operation that would appear to make sense only for Composites can be implemented for all Components by moving it to the Component class. For example, the interface for accessing children is a fundamental part of a Composite class but not necessarily Leaf classes. But if we view a Leaf as a Component that *never* has children, then we can define a default operation for child access in the Component class that never *returns* any children. Leaf classes can use the default implementation, but Composite classes will reimplement it to return their children.

The child management operations are more troublesome and are discussed in the next item.

---

4. **Declaring the child management operations.** Although the Composite class *implements* the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes *declare* these operations in the Composite class hierarchy. Should we declare these operations in the Component and make them meaningful for Leaf classes, or should we declare and define them only in Composite and its subclasses?

   The decision involves a trade-off between safety and transparency:

   - Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
   - Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

   We have emphasized transparency over safety in this pattern. If you opt for safety, then at times you may lose type information and have to convert a component into a composite. How can you do this without resorting to a type-unsafe cast?

   One approach is to declare an operation `Composite* GetComposite()` in the Component class. Component provides a default operation that returns a null pointer. The Composite class redefines this operation to return itself through the `this` pointer:

```
class Composite;

class Component {
public:
        //...
        virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
        void Add(Component*);
        // ...
        virtual Composite* GetComposite() { return this; }
};
```

```
class Leaf : public Component {
        // ...
};
```

`GetComposite` lets you query a component to see if it's a composite. You can perform `Add` and `Remove` safely on the composite it returns.

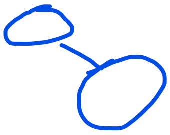
```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
        test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
        test->Add(new Leaf); // will not add leaf
}
```



Similar tests for a Composite can be done using the C++ `dynamic_cast` construct.

Of course, the problem here is that we don't treat all components uniformly. We have to revert to testing for different types before taking the appropriate action.

The only way to provide transparency is to define default `Add` and `Remove` operations in Component. That creates a new problem: There's no way to implement `Component::Add` without introducing the possibility of it failing. You could make it do nothing, but that ignores an important consideration; that is, an attempt to add something to a leaf probably indicates a bug. In that case, the `Add` operation produces garbage. You could make it delete its argument, but that might not be what clients expect.

Usually it's better to make `Add` and `Remove` fail by default (perhaps by raising an exception) if the component isn't allowed to have children or if the argument of `Remove` isn't a child of the component, respectively.

Another alternative is to change the meaning of "remove" slightly. If the component maintains a parent reference, then we could redefine `Component::Remove` to remove itself from its parent. However, there still isn't a meaningful interpretation for a corresponding `Add`.

5. **Should Component implement a list of Components?** You might be tempted to define the set of children as an instance variable in the Component class where the child access and management operations are declared. But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children. This is worthwhile only if there are relatively few children in the structure.
6. **Child ordering.** Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program.

When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children. The Iterator pattern can guide you in this.

7. **Caching to improve performance.** If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search. For example, the Picture class from the Motivation example could cache the bounding box of its children. During drawing or selection, this cached bounding box lets the Picture avoid drawing or searching when its children aren't visible in the current window.

   Changes to a component will require invalidating the caches of its parents. This works best when components know their parents. So if you're using caching, you need to define an interface for telling composites that their caches are invalid.
8. **Who should delete components?** In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exception to this rule is when Leaf objects are immutable and thus can be shared.
9. **What's the best data structure for storing components?** Composites may use a variety of data structures to store their children, including linked lists, trees, arrays, and hash tables. The choice of data structure depends (as always) on efficiency. In fact, it isn't even necessary to use a general-purpose data structure at all. Sometimes composites have a variable for each child, although this requires each subclass of Composite to implement its own management interface. See Interpreter for an example.

Related Patterns

Often the component-parent link is used for a Chain of Responsibility.
Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
Flyweight lets you share components, but they can no longer refer to their parents.
Iterator can be used to traverse composites.
Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

Decorator

(Object Structural)

also known as: Wrapper

Intent

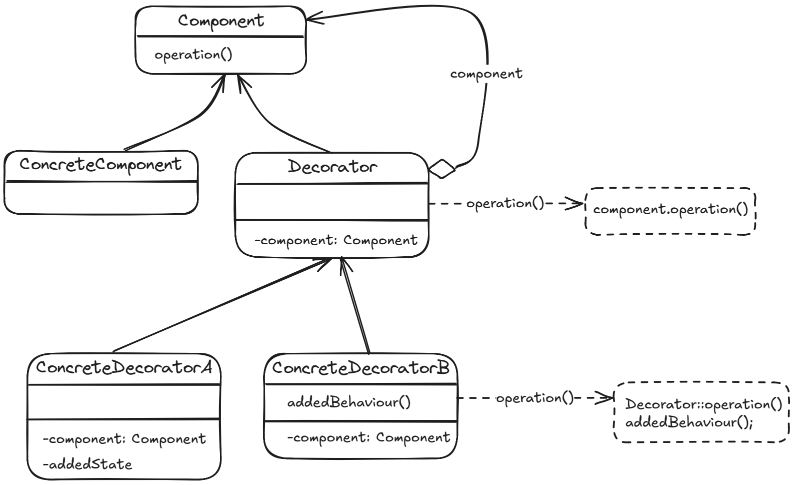
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Applicability

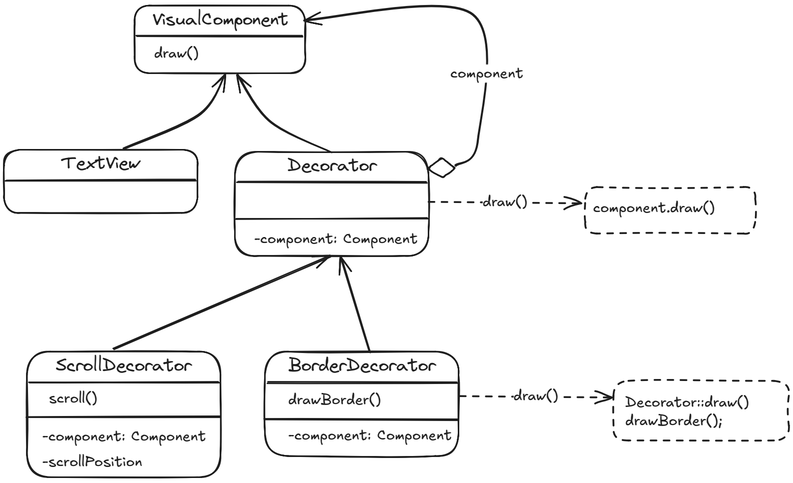
Use Decorator

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition maybe hidden or otherwise unavailable for subclassing.

Structure



A typical Decorator object structure might look like this:



Participants

1. **Component** (VisualComponent)
   - defines the interface for objects that can have responsibilities added to them dynamically.
2. **ConcreteComponent** (TextView)
   - defines an object to which additional responsibilities can be attached.
3. **Decorator**

- maintains a reference to a Component object and defines an interface that conforms to Component's interface.

4. **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
    - adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

1. **More flexibility than static inheritance**. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.
   Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.
2. **Avoids feature-laden classes high up in the hierarchy**. Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.
3. **A decorator and its component aren't identical**. A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.
4. **Lots of little objects**. A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

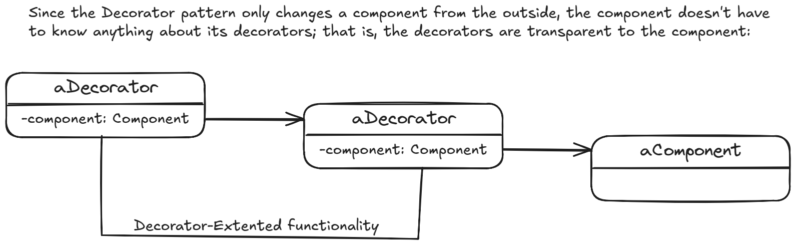
1. **Interface conformance**. A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

2. **Omitting the abstract Decorator class**. There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.
3. **Keeping Component classes lightweight**. To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.
4. **Changing the skin of an object versus changing its guts**. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts.

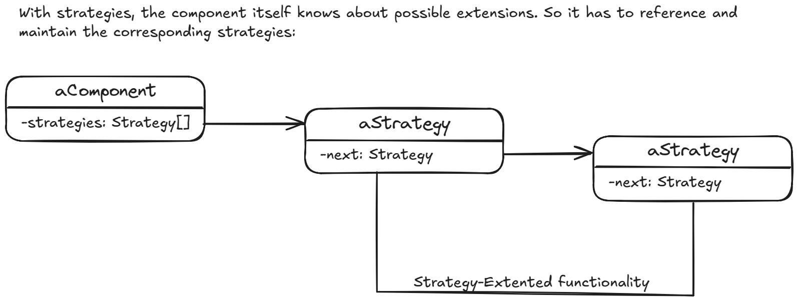
   Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

   For example, we can support different border styles by having the component defer border-drawing to a separate Border object. The Border object is a Strategy object that encapsulates a border-drawing strategy. By extending the number of strategies from just one to an open-ended list, we achieve the same effect as nesting decorators recursively.

   In MacApp 3.0 and Bedrock, for example, graphical components (called "views") maintain a list of "adorner" objects that can attach additional adornments like borders to a view component. If a view has any adorners attached, then it gives them a chance to draw additional embellishments. MacApp and Bedrock must use this approach because the View class is heavyweight. It would be too expensive to use a full-fledged View just to add a border.



Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component:



With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies:

The Strategy-based approach might require modifying the component to accommodate new extensions. On the other hand, a strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's. A strategy for rendering a border, for example, need only define the interface for rendering a border (DrawBorder, GetWidth, etc.), which means that the strategy can be lightweight even if the Component class is heavyweight.

MacApp and Bedrock use this approach for more than just adorning views. They also use it to augment the event-handling behavior of objects. In both systems, a view maintains a list of "behavior" objects that can modify and intercept events. The view gives each of the registered behavior objects a chance to handle the event before nonregistered behaviors, effectively overriding them. You can decorate a view with special keyboard-handling support, for example, by registering a behavior object that intercepts and handles key events.

Related Patterns

Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite: A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

Strategy: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

Facade

(Object Structural)

also known as: Wrapper

Intent

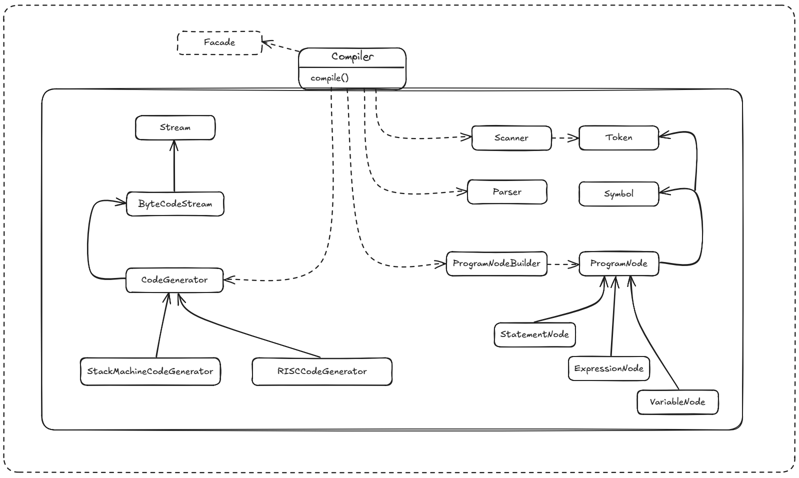
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants

1. **Facade** (Compiler)
   - knows which subsystem classes are responsible for a request.
   - delegates client requests to appropriate subsystem objects.
2. **subsystem classes** (Scanner, Parser, ProgramNode, etc.)
   - implement subsystem functionality.
   - handle work assigned by the Facade object.
   - have no knowledge of the facade; that is, they keep no references to it.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

1. It **shields clients from subsystem components**, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. **It promotes weak coupling between the subsystem and its clients**. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.
   Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with

facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.
3. It doesn't prevent applications from using subsystem classes if they need to. Thus you **can choose between** ease of use and generality.

Implementation

Consider the following issues when implementing a facade:

1. **Reducing client-subsystem coupling.** The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
   An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.
2. **Public versus private subsystem classes.** A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.
   The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.
   Making subsystem classes private would be useful, but few object-oriented languages support it. Both C++ and Smalltalk traditionally have had a global name space for classes. Recently, however, the C++ standardization committee added name spaces to the language, which will let you expose just the public subsystem classes.

Related Patterns

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.
Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication be- tween colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.
Usually only one Facade object is required. Thus Facade objects are often Singletons.

Flyweight

(Object Structural)

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.
It's often best to implement State and Strategy objects as flyweights.

Proxy

(Object Structural)

also known as: Surrogate

Intent

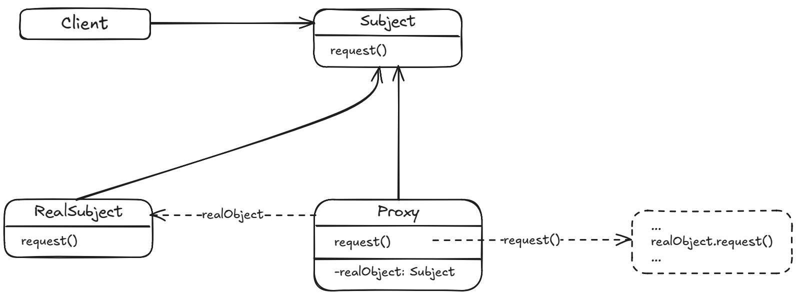
Provide a surrogate or placeholder for another object to control access to it.

Applicability

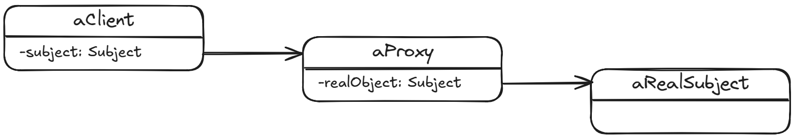
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP uses the class NXProxy for this purpose.
   Coplien calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. Like an ImageProxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system provide protected access to operating system objects.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers).
   - loading a persistent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Here's a possible object diagram of a proxy structure at run-time:



Participants

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - **remote proxies** are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - **virtual proxies** may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
    - **protection proxies** check that the caller has the access permissions required to perform a request.
- **Subject** (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject** (Image)
  - defines the real object that the proxy represents.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write**, and it's related to creation on demand. Copying a large and complicated object can be an expensive operation. If the copy is never modified, then there's no need to incur this cost. By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it's modified.

To make copy-on-write work, the subject must be reference counted. Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it. In that case the proxy must also decrement the subject's reference count. When the reference count goes to zero, the subject gets deleted.

Copy-on-write can reduce the cost of copying heavyweight subjects significantly.

Implementation

The Proxy pattern can exploit the following language features:

1. **Overloading the member access operator in C++.** C++ supports overloading `operator->`, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced. This can be helpful for implementing some kinds of proxy; the proxy behaves just like a pointer.
   The following example illustrates how to use this technique to implement a virtual proxy called `ImagePtr`.

   ```
   class Image;
   extern Image* LoadAnImageFile(const char*);
       // external function

   class ImagePtr {
   public:
       ImagePtr(const char* imageFile);
       virtual ~ImagePtr();

       virtual Image* operator->();
       virtual Image& operator*();
   private:
       Image* LoadImage();
   private:
       Image* _image;
       const char* _imageFile;
   };

   ImagePtr::ImagePtr (const char* theImageFile) {
       _imageFile = theImageFile;
       _image = 0;
   }

   Image* ImagePtr::LoadImage () {
       if (_image == 0) {
           _image = LoadAnImageFile(_imageFile);
       }
       return _image;
   }
   ```

   The overloaded `->` and `*` operators use `LoadImage` to return `_image` to callers (loading it if necessary).

   ```
   Image* ImagePtr::operator-> () {
           return LoadImage();
   }

   Image& ImagePtr::operator* () {
           return *LoadImage();
   }
   ```

This approach lets you call `Image` operations through `ImagePtr` objects without going to the trouble of making the operations part of the `ImagePtr` interface:

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
        // (image.operator->())->Draw(Point(50, 100))
```

Notice how the `image` proxy acts like a pointer, but it's not declared to be a pointer to an `Image`. That means you can't use it exactly like a real pointer to an `Image`. Hence clients must treat `Image` and `ImagePtr` objects differently in this approach.

Overloading the member access operator isn't a good solution for every kind of proxy. Some proxies need to know precisely *which* operation is called, and overloading the member access operator doesn't work in those cases.

Consider a virtual proxy example. The image should be loaded at a specific time—namely when the Draw operation is called—and not whenever the image is referenced. Overloading the access operator doesn't allow this distinction. In that case we must manually implement each proxy operation that forwards the request to the subject.

These operations are usually very similar to each other.

Typically all operations verify that the request is legal, that the original object exists, etc., before forwarding the request to the subject. It's tedious to write this code again and again. So it's common to use a preprocessor to generate it automatically.

```
Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}

const Point& ImageProxy::GetExtent () {
    if  Zero  {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

2. **Using `doesNotUnderstand` in Smalltalk.** Smalltalk provides a hook that you can use to support automatic forwarding of requests. Smalltalk calls `doesNotUnderstand: aMessage` when a client sends a message to a receiver that has no corresponding method. The Proxy class can redefine `doesNotUnderstand` so that the message is forwarded to its subject.

   To ensure that a request is forwarded to the subject and not just absorbed by the proxy silently, you can define a Proxy class that doesn't understand *any* messages. Smalltalk lets you do this by defining Proxy

as a class with no superclass.
The main disadvantage of `doesNotUnderstand:` is that most Smalltalk systems have a few special messages that are handled directly by the virtual machine, and these do not cause the usual method look-up. The only one that's usually implemented in Object (and so can affect proxies) is the identity operation = =.
If you're going to use `doesNotUnderstand:` to implement Proxy, then you must design around this problem. You can't expect identity on proxies to mean identity on their real subjects. An added disadvantage is that `doesNotUnderstand:` was developed for error handling, not for building proxies, and so it's generally not very fast.

3. **Proxy doesn't always have to know the type of real subject.** If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.
Another implementation issue involves how to refer to the subject before it's instantiated. Some proxies have to refer to their subject whether it's on disk or in memory. That means they must use some form of address space-independent object identifiers. We can use a file name for this purpose.

Related Patterns

Adapter: An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.
Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator. On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host ID and local address on host." A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Chain of Responsibility

(Object Behavioral)

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Command

(Object Behavioral)

also known as: Action, Transaction

Intent

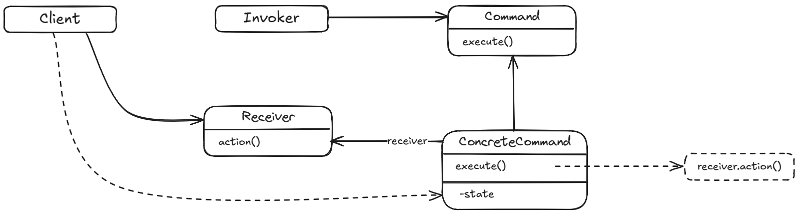
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Applicability

Use the Command pattern when you want to

- parameterize objects by an action to perform, as MenuItem objects did above. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- support undo. The Command's Execute operation can store state for revers- ing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.
- support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and re-executing them with the Execute operation.
- structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

Structure



Participants

1. **Command**
   - declares an interface for executing an operation.
2. **ConcreteCommand** (PasteCommand, OpenCommand)
   - defines a binding between a Receiver object and an action.

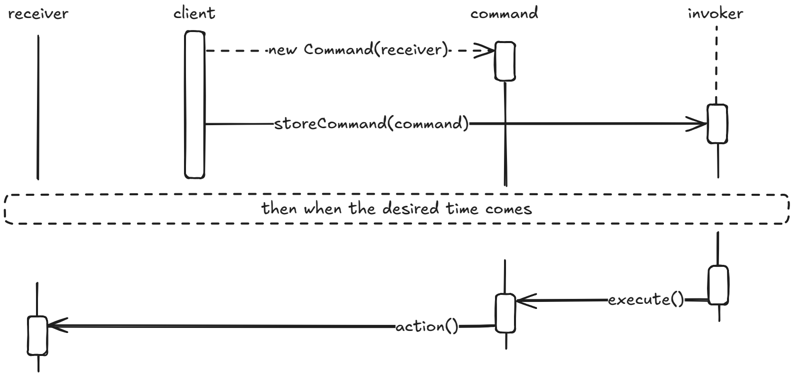
   - implements Execute by invoking the corresponding operation(s) on Receiver.

3. **Client** (Application)
   - creates a ConcreteCommand object and sets its receiver.
4. **Invoker** (MenuItem)
   - asks the command to carry out the request.
5. **Receiver** (Document, Application)
   - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

The following diagram shows the interactions between these objects. It illustrates how Command decouples the invoker from the receiver (and the request it carries out).



Consequences

The Command pattern has the following consequences:

1. Command decouples the object that invokes the operation from the one that knows how to perform it.
2. Commands are first-class objects. They can be manipulated and extended like any other object.
3. You can assemble commands into a composite command. An example is a MacroCommand class. In general, composite commands are an instance of the Composite pattern.

```
void MacroCommand::Execute () {
        for (cmds.First(); !cmds.IsDone(); cmds.Next()) {
                Command* c = cmds.CurrentItem();
                c->Execute();
        }
}
```

4. It's easy to add new Commands, because you don't have to change existing classes.

Implementation

Consider the following issues when implementing the Command pattern:

1. **How intelligent should a command be?** A command can have a wide range of abilities. At one extreme it merely defines a binding between a receiver and the actions that carry out the request. At the other extreme it implements everything itself without delegating to a receiver at all. The latter extreme is useful when you want to define commands that are independent of existing classes, when no suitable receiver exists, or when a command knows its receiver implicitly. For example, a command that creates another application window may be just as capable of creating the window as any other object. Somewhere in between these extremes are commands that have enough knowledge to find their receiver dynamically.
2. **Supporting undo and redo.** Commands can support undo and redo capabilities if they provide a way to reverse their execution (e.g., an Unexecute or Undo operation). A ConcreteCommand class might need to store additional state to do so. This state can include
    - the Receiver object, which actually carries out operations in response to the request,
    - the arguments to the operation performed on the receiver, and
    - any original values in the receiver that can change as a result of handling the request. The receiver must provide operations that let the command return the receiver to its prior state.
      To support one level of undo, an application needs to store only the command that was executed last. For multiple-level undo and redo, the application needs a **history list** of commands that have been executed, where the maximum length of the list determines the number of undo/redo levels. The history list stores sequences of commands that have been executed. Traversing backward through the list and reverse-executing commands cancels their effect; traversing forward and executing commands reexecutes them.
      An undoable command might have to be copied before it can be placed on the history list. That's because the command object that carried out the original request, say, from a MenuItem, will perform other requests at later times. Copying is required to distinguish different invocations of the same command if its state can vary across invocations.
      For example, a DeleteCommand that deletes selected objects must store different sets of objects each time it's executed. Therefore the DeleteCommand object must be copied following execution, and the copy is placed on the history list. If the command's state never changes on execution, then copying is not required—only a reference to the command need be placed on the history list. Commands that must be copied before being placed on the history list act as prototypes (see Prototype).
3. **Avoiding error accumulation in the undo process.** Hysteresis can be a problem in ensuring a reliable, semantics-preserving undo/redo mechanism. Errors can accumulate as commands are executed, unexecuted, and reexecuted repeatedly so that an application's state eventually diverges from original values. It may be necessary therefore to store more information in the command to ensure that objects are restored to their original state. The Memento pattern can be applied to give the command access to this information without exposing the internals of other objects.
4. **Using C++ templates.** For commands that
   1. aren't undoable and;

2. don't require arguments, we can use C++ templates to avoid creating a Command subclass for every kind of action and receiver. We show how to do this in the Sample Code section.

```cpp
template
class SimpleCommand : public Command {
public:
typedef void

- Action

();

````
    SimpleCommand(Receiver* r, Action a) : _receiver(r), _action(a) { }

    virtual void execute() {
        (_receiver->*_action)();
    }
private:
    Action _action;
    Receiver* _receiver;
};
```
````

To create a command that calls `Action` on an instance of class `MyClass`, a client simply writers

```
MyClass* receiver = new MyClass();
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Related Patterns

A Composite can be used to implement MacroCommands.
A Memento can keep state the command requires to undo its effect.
A command that must be copied before being placed on the history list acts as a Prototype.

Interpreter

(Class Behavioral)

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when

- the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
- efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often trans- formed into state machines. But even then, the translator can be implemented by the Interpreter pattern, so the pattern is still applicable.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Composite: The abstract syntax tree is an instance of the Composite pattern.
Flyweight shows how to share terminal symbols within the abstract syntax tree.
Iterator: The interpreter can use an Iterator to traverse the structure.
Visitor can be used to maintain the behavior in each node in the abstract syntax tree in one class.

Iterator

(Object Behavioral)

also known as: Cursor

Intent

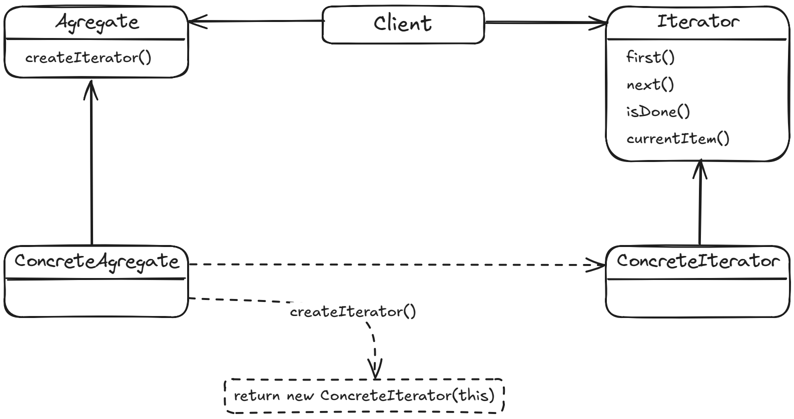
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure



Participants

1. **Iterator**
   - defines an interface for accessing and traversing elements.
2. **ConcreteIterator**
   - implements the Iterator interface.
   - keeps track of the current position in the traversal of the aggregate.
3. **Aggregate**
   - defines an interface for creating an Iterator object.
4. **ConcreteAggregate**
   - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

The Iterator pattern has three important consequences:

1. **It supports variations in the traversal of an aggregate**. Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.
2. **Iterators simplify the Aggregate interface**. Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. **More than one traversal can be pending on an aggregate**. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation

```
void PrintEmployees (Iterator<Employee*>& i) {
        for (i.first(); !i.isDone(); i.next()) {
                i.currentItem()->print();
        }
}
```

Iterator has many implementation variants and alternatives. Some important ones follow. The trade-offs often depend on the control structures your language provides. Some languages, for example) even support this pattern directly.

1. **Who controls the iteration?** A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**. Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.
   External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like C++ that does not provide anonymous functions, closures, or continuations like Smalltalk and CLOS. But on the other hand, internal iterators are easier to use, because they define the iteration logic for you.
2. **Who defines the traversal algorithm?** The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.
   If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the

aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

3. **How robust is the iterator?** It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might end up accessing an element twice or missing it completely. A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.

   A **robust iterator** ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. There are many ways to implement robust iterators. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.

   Kofler provides a good discussion of how robust iterators are implemented in ET++. Murray discusses the implementation of robust iterators for the USL StandardComponents' List class.

4. **Additional Iterator operations.** The minimal interface to Iterator consists of the operations First, `Next`, `IsDone`, and `CurrentItem`. Some additional operations might prove useful. For example, ordered aggregates can have a Previous operation that positions the iterator to the previous element. A SkipTo operation is useful for sorted or indexed collections. SkipTo positions the iterator to an object matching specific criteria.

5. **Using polymorphic iterators in C++**. Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack.

   Polymorphic iterators have another drawback: the client is responsible for deleting them. This is error-prone, because it's easy to forget to free a heap-allocated iterator object when you're finished with it. That's especially likely when there are multiple exit points in an operation. And if an exception is triggered, the iterator object will never be freed.

   The Proxy pattern provides a remedy. We can use a stack-allocated proxy as a stand-in for the real iterator. The proxy deletes the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions. This is an application of the well-known C++ technique "resource allocation is initialization".

6. **Iterators may have privileged access**. An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. We can express this close relationship in C++ by making the iterator a `friend` of its aggregate. Then you don't need to define aggregate operations whose sole purpose is to let iterators implement traversal efficiently.

   However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid this problem, the Iterator class can include `protected` operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and *only* Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.

7. **Iterators for composites**. External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates. Therefore an external iterator has to store a path through the Composite to keep track of the current object. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack. If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and

   children, then a cursor-based iterator may offer a better alternative. The cursor only needs to keep track of the current node; it can rely on the node interface to traverse the Composite.

   Composites often need to be traversed in more than one way. Preorder, postorder, inorder, and breadth-first traversals are common. You can support each kind of traversal with a different class of iterator.

8. **Null iterators**. A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal; that is, its IsDone operation always evaluates to true.

   NullIterator can make traversing tree-structured aggregates (like Composites) easier. At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator as usual. But leaf elements return an instance of NullIterator. That lets us implement traversal over the entire structure in a uniform way.

```
void PrintEmployees (Iterator<Component*>& i) {
        for (i.first(); !i.isDone(); i.next()) {
                i.currentItem()->print();
                PrintEmployees(i.getIterator());
        }
}
```

Related Patterns

Composite: Iterators are often applied to recursive structures such as Composites.

Factory Method: Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.

Memento is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.

Mediator

(Object Behavioral)

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Applicability

Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.

- a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.
Colleagues can communicate with the mediator using the Observer pattern.

Memento

(Object Behavioral)

also known as: Token

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Command: Commands can use mementos to maintain state for undoable operations.
Iterator: Mementos can be used for iteration as described earlier.

Observer

(Object Behavioral)

also known as: Dependents, Publish-Subscribe

Intent

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

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.
Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

State

(Object Behavioral)

also known as: Objects for States

Intent

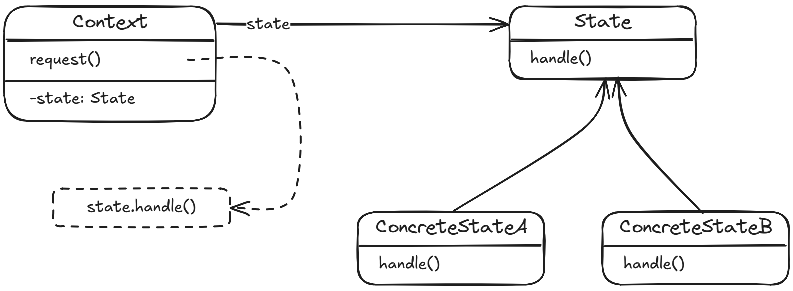
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure



Participants

1. **Context** (TCPConnection)
   - defines the interface of interest to clients.
   - maintains an instance of a ConcreteState subclass that defines the current state.
2. **State** (TCPState)
   - defines an interface for encapsulating the behavior associated with a particular state of the Context.
3. **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
   - each subclass implements a behavior associated with a state of the Context.

Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

The State pattern has the following consequences:

1. **It localizes state-specific behavior and partitions behavior for different states.** The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.
   An alternative is to use data values to define internal states and have Context operations check the data explicitly. But then we'd have look-alike conditional or case statements scattered throughout Context's implementation. Adding a new state could require changing several operations, which complicates maintenance.
   The State pattern avoids this problem but might introduce another, because the pattern distributes behavior for different states across several State subclasses. This increases the number of classes and is less compact than a single class. But such distribution is actually good if there are many states, which would otherwise necessitate large conditional statements.
   Like long procedures, large conditional statements are undesirable. They're monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend. The State pattern offers a better way to structure state-specific code. The logic that determines the state transitions doesn't reside in monolithic `if` or `switch` statements but instead is partitioned between the State subclasses. Encapsulating each state transition and action in a class elevates the idea of an execution state to full object status. That imposes structure on the code and makes its intent clearer.
2. **It makes state transitions explicit.** When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states, because state transitions are atomic from the Context's perspective—they happen by rebinding *one* variable (the Context's State object variable), not several.
3. **State objects can be shared.** If State objects have no instance variables—that is, the state they represent is encoded entirely in their type—then contexts can share a State object. When states are shared in this way, they are essentially flyweights (see Flyweight) with no intrinsic state, only behavior.

Implementation

The State pattern raises a variety of implementation issues:

1. **Who defines the state transitions?** The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context. It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
   Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses. A disadvantage of decentralization is that one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.
2. **A table-based alternative.** In *C++ Programming Style*, Cargill describes another way to impose structure on state-driven code: He uses tables to map inputs to state transitions. For each state, a table maps every possible input to a succeeding state. In effect, this approach converts conditional code (and virtual functions, in the case of the State pattern) into a table look-up.
   The main advantage of tables is their regularity: You can change the transition criteria by modifying data instead of changing program code. There are some disadvantages, however:

   - A table look-up is often less efficient than a (virtual) function call.
   - Putting transition logic into a uniform, tabular format makes the transition criteria less explicit and therefore harder to understand.
   - It's usually difficult to add actions to accompany the state transitions. The table-driven approach captures the states and their transitions, but it must be augmented to perform arbitrary computation on each transition.
     The key difference between table-driven state machines and the State pattern can be summed up like this: The State pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions.

3. **Creating and destroying State objects.** A common implementation trade-off worth considering is whether
   1. to create State objects only when they are needed and destroy them thereafter versus
   2. creating them ahead of time and never destroying them.
      The first choice is preferable when the states that will be entered aren't known at runtime, *and* contexts change state infrequently. This approach avoids creating objects that won't be used, which is important if the State objects store a lot of information. The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly. Instantiation costs are paid once up-front, and there are no destruction costs at all. This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.
4. **Using dynamic inheritance.** Changing the behavior for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most object-oriented programming languages. Exceptions include Self and other delegation-based languages that provide such a mechanism and hence support the State pattern directly. Objects in Self can delegate operations to other objects to achieve a form of dynamic inheritance. Changing the delegation target at run-time effectively changes the inheritance structure. This mechanism lets objects change their behavior and amounts to changing their class.

Related Patterns

The Flyweight pattern explains when and how State objects can be shared.
State objects are often Singletons.

Strategy

(Object Behavioral)

also known as: Policy

Intent

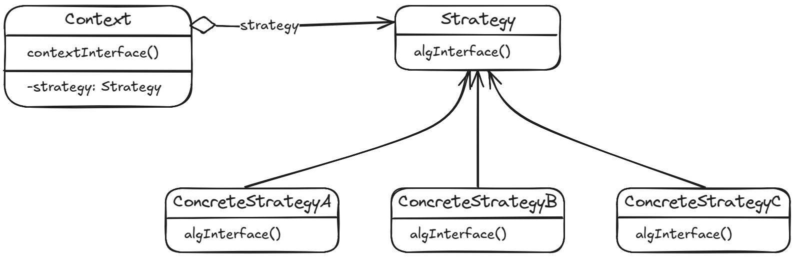
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

The Strategy pattern has the following benefits and drawbacks:

1. **Families of related algorithms**. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
2. **An alternative to subclassing**. Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. **Strategies eliminate conditional statements**. The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.

   For example, without strategies, the code for breaking text into lines could look like

   ```
   void Composition::Repair () {
           switch (_breakingStrategy) {
           case SimpleStrategy:
                   ComposeWithSimpleCompositor();
                   break;
           case TeXStrategy:
                   ComposeWithTeXCompositor();
                   break;
           // ...
           }
           // merge results with existing composition, if necessary
   }
   ```

   The Strategy pattern eliminates this case statement by delegating the linebreaking task to a Strategy object:

   ```
   void Composition::Repair () {
           _compositor->Compose();
           // merge results with existing composition, if necessary
   }
   ```

   Code containing many conditional statements often indicates the need to apply the Strategy pattern.
4. **A choice of implementations**. Strategies can provide different implementations of the *same* behavior. The client can choose among strategies with different time and space trade-offs.
5. **Clients must be aware of different Strategies**. The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients.

6. **Communication overhead between Strategy and Context**. The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; simple ConcreteStrategies may use none of it! That means there will be times when the context creates and initializes parameters that never get used. If this is an issue, then you'll need tighter coupling between Strategy and Context.
7. **Increased number of objects**. Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object. Shared strategies should not maintain state across invocations. The Flyweight pattern describes this approach in more detail.

Implementation

Consider the following implementation issues:

1. **Defining the Strategy and Context interfaces**. The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
   One approach is to have Context pass data in parameters to Strategy operations—in other words, take the data to the strategy. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.
   Another technique has a context pass *itself* as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. Either way, the strategy can request exactly what it needs. But now Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.
   The needs of the particular algorithm and its data requirements will determine the best technique.
2. **Strategies as template parameters**. In C++ templates can be used to configure a class with a strategy. This technique is only applicable if

   1. the Strategy can be selected at compile-time, and;
   2. it does not have to be changed at run-time. In this case, the class to be configured (e.g., `Context`) is defined as a template class that has a `Strategy` class as a parameter:

   ```
   template <class AStrategy>
   class Context {
           void Operation() { theStrategy.DoAlgorithm(); }
           // ...
   private:
           AStrategy theStrategy;
   };
   ```

   The class is then configured with a `Strategy` class when it's instantiated:

   ```
   class MyStrategy {
   public:
           void DoAlgorithm();
   };
   ```

```
Context<MyStrategy> aContext;
```

With templates, there's no need to define an abstract class that defines the interface to the `Strategy.` Using `Strategy` as a template parameter also lets you bind a `Strategy` to its `Context` statically, which can increase efficiency.

3. **Making Strategy objects optional**. The Context class may be simplified if it's meaningful *not* to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If there is one, then Context uses it normally. If there isn't a strategy, then Context carries out default behavior. The benefit of this approach is that clients don't have to deal with Strategy objects at all *unless* they don't like the default behavior.

Related Patterns

Flyweight: Strategy objects often make good flyweights.

Template Method

(Class Behavioral)

Intent

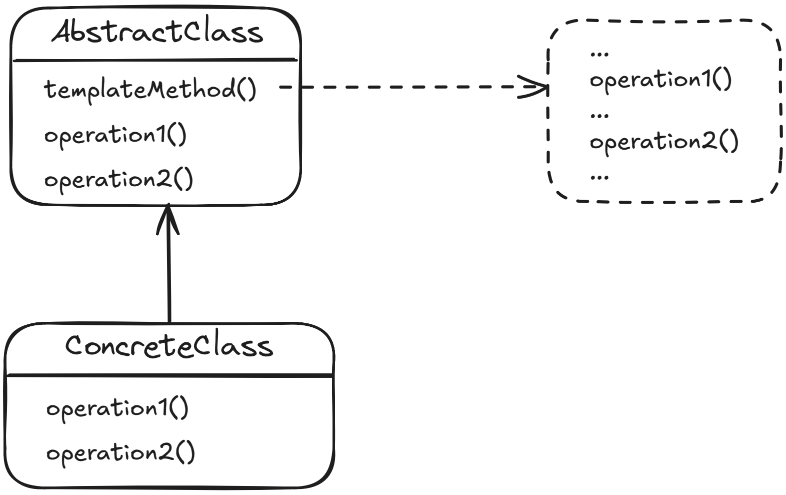
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Applicability

The Template Method pattern should be used

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
- to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Structure



Participants

1. **AbstractClass** (Recipe)
   - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
   - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
2. **ConcreteClass** (Chicken Recipe)
   - implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.

Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you". This refers to how a parent class calls the operations of a subclass and not the other way around.

Template methods call the following kinds of operations:

- concrete operations (either on the ConcreteClass or on client classes);
- concrete AbstractClass operations (i.e., operations that are generally useful to subclasses);
- primitive operations (i.e., abstract operations);
- factory methods (see Factory Method); and
- **hook operations**, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.

It's important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

A subclass can *extend* a parent class operation's behavior by overriding the operation and calling the parent operation explicitly:

```
void DerivedClass::Operation () {
        // DerivedClass extended behavior
        ParentClass::Operation();
}
```

Unfortunately, it's easy to forget to call the inherited operation. We can transform such an operation into a template method to give the parent control over how subclasses extend it. The idea is to call a hook operation from a template method in the parent class. Then subclasses can then override this hook operation:

```
void ParentClass::Operation () {
        // ParentClass behavior
        HookOperation();
}
```

`HookOperation` does nothing in `ParentClass`:

```
void ParentClass::HookOperation () { }
```

Subclasses override `HookOperation` to extend its behavior:

```
void DerivedClass::HookOperation () {
         // derived class extension
}
```

Implementation

Three implementation issues are worth noting:

1. **Using C++ access control.** In C++, the primitive operations that a template method calls can be declared protected members. This ensures that they are only called by the template method. Primitive operations that *must* be overridden are declared pure virtual. The template method itself should not be overridden; therefore you can make the template method a nonvirtual member function.
2. **Minimizing primitive operations.** An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.
3. **Naming conventions.** You can identify the operations that should be overridden by adding a prefix to their names. For example, the MacApp framework for Macintosh applications prefixes template method names with "Do-": "DoCreateDocument", "DoRead", and so forth.

Related Patterns

Factory Methods are often called by template methods. In the Motivation example, the factory method DoCreateDocument is called by the template method OpenDocument.
Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

Visitor

(Object Behavioral)

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

Structure

Participants

Collaborations

Consequences

Implementation

Related Patterns

Composite: Visitors can be used to apply an operation over an object structure defined by the Composite pattern.
Interpreter: Visitor may be applied to do the interpretation.

Thanks