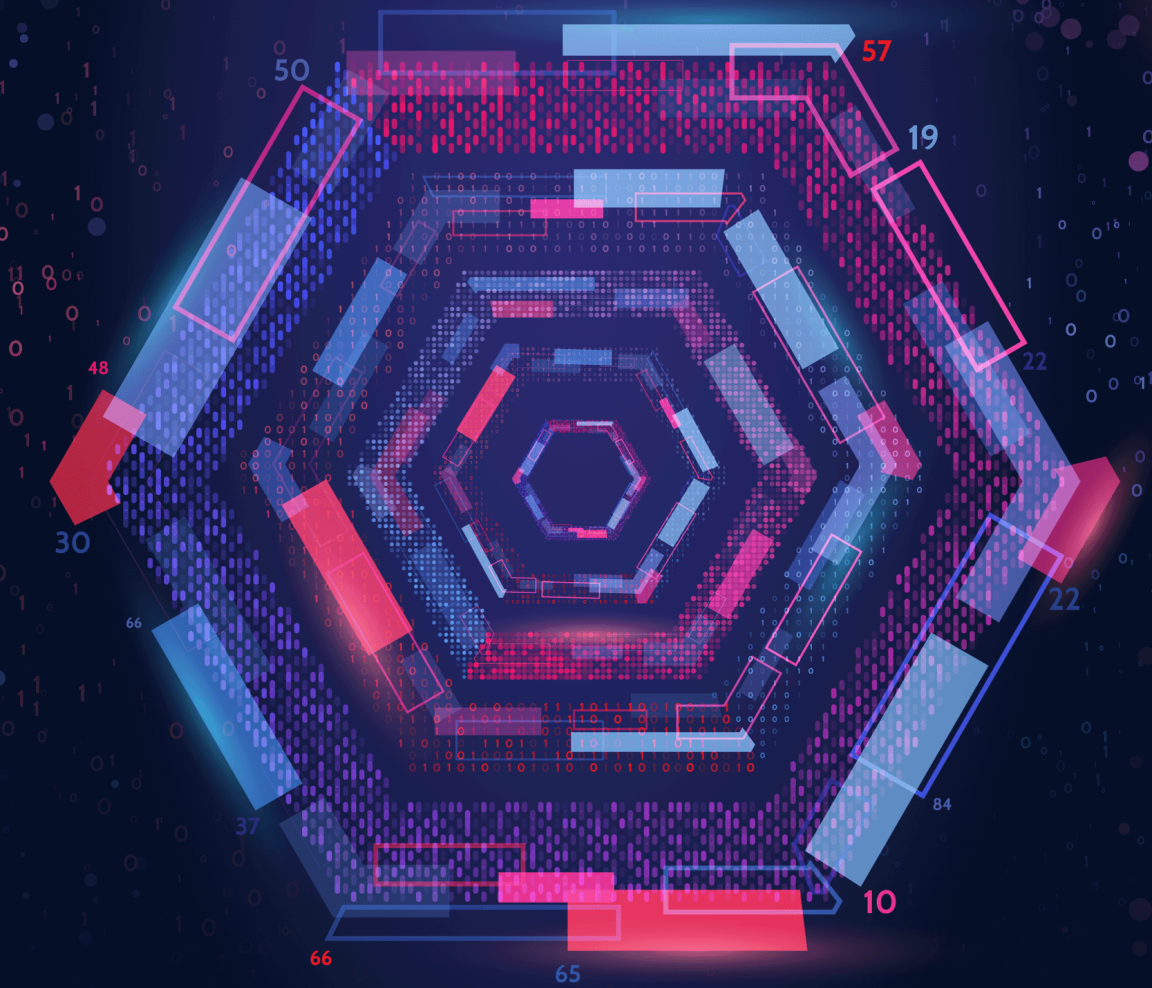
Dive Into DESIGN PATTERNS



Alexander Shvets

Dive Into

DESIGN PATTERNS

A Few Words on Copyright

Hi! My name is Alexander Shvets. I'm the author of the book **Dive Into Design Patterns** and the online course **Dive Into Refactoring**.

All profit from the sale of my books and courses is spent on the development of **Refactoring.Guru**. Each copy sold helps the project immensely and brings the moment of a new book release a little bit closer.

support@refactoring.guru

Illustrations: Dmitry Zhart
Editing: Andrew Wetmore, Rhyan Solomon

I dedicate this book to my wife, Maria. If it hadn't been for her, I'd probably have finished the book some 30 years later.

Table of Contents

How to Read This Book

This book contains the descriptions of 22 classic design patterns formulated by the “Gang of Four” (or simply GoF) in 1994.

Each chapter explores a particular pattern. Therefore, you can read from cover to cover or by picking the patterns you’re interested in.

Many patterns are related, so you can easily jump from topic to topic using numerous anchors. The end of each chapter has a list of links between the current pattern and others. If you see the name of a pattern that you haven’t seen yet, just keep reading—this item will appear in one of the next chapters.

Design patterns are universal. Therefore, all code samples in this book are written in pseudocode that doesn’t constrain the material to a particular programming language.

Prior to studying patterns, you can refresh your memory by going over the **key terms of object-oriented programming**. That chapter also explains the basics of UML diagrams, which is useful because the book has tons of them. Of course, if you already know all of that, you can proceed to **learning patterns** right away.

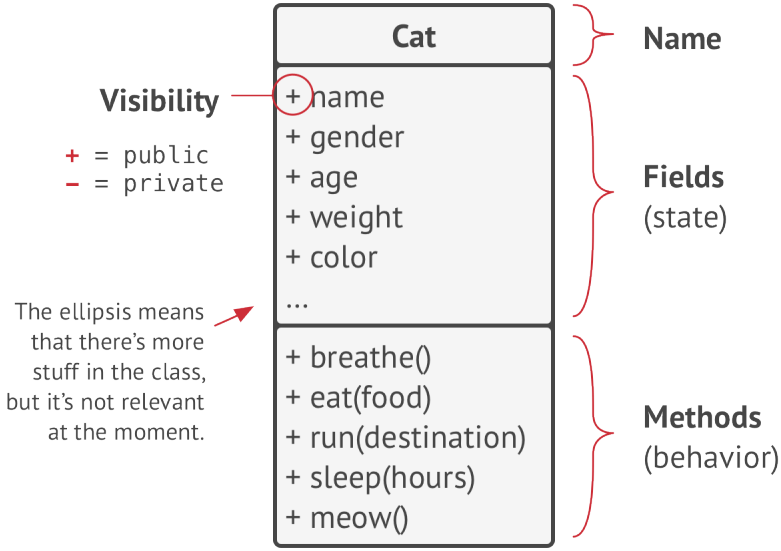
INTRODUCTION TO OOP

Basics of OOP

***O**bject-**o**riented **p**rogramming* is a paradigm based on the concept of wrapping pieces of data, and behavior related to that data, into special bundles called **objects**, which are constructed from a set of "blueprints", defined by a programmer, called **classes**.

Objects, classes

Do you like cats? I hope you do because I'll try to explain the OOP concepts using various cat examples.



This is a UML class diagram. You'll see a lot of such diagrams in the book.

Say you have a cat named Oscar. Oscar is an object, an instance of the `Cat` class. Every cat has a lot of standard attributes: name, sex, age, weight, color, favorite food, etc. These are the class's *fields*.

All cats also behave similarly: they breathe, eat, run, sleep and meow. These are the class's *methods*. Collectively, fields and methods can be referenced as the *members* of their class.

> Data stored inside the object's fields is often referenced as *state*, and all the object's methods define its *behavior*.

Oscar: Cat

```
name    = "Oscar"
sex     = "male"
age     =  3
weight  =  7
color   =  brown
texture =  striped
```

Luna: Cat

```
name    = "Luna"
sex     = "female"
age     =  2
weight  =  5
color   =  gray
texture =  plain
```

Objects are instances of classes.

Luna, your friend's cat, is also an instance of the `Cat` class. It has the same set of attributes as Oscar. The difference is in values of these attributes: her sex is female, she has a different color, and weighs less.

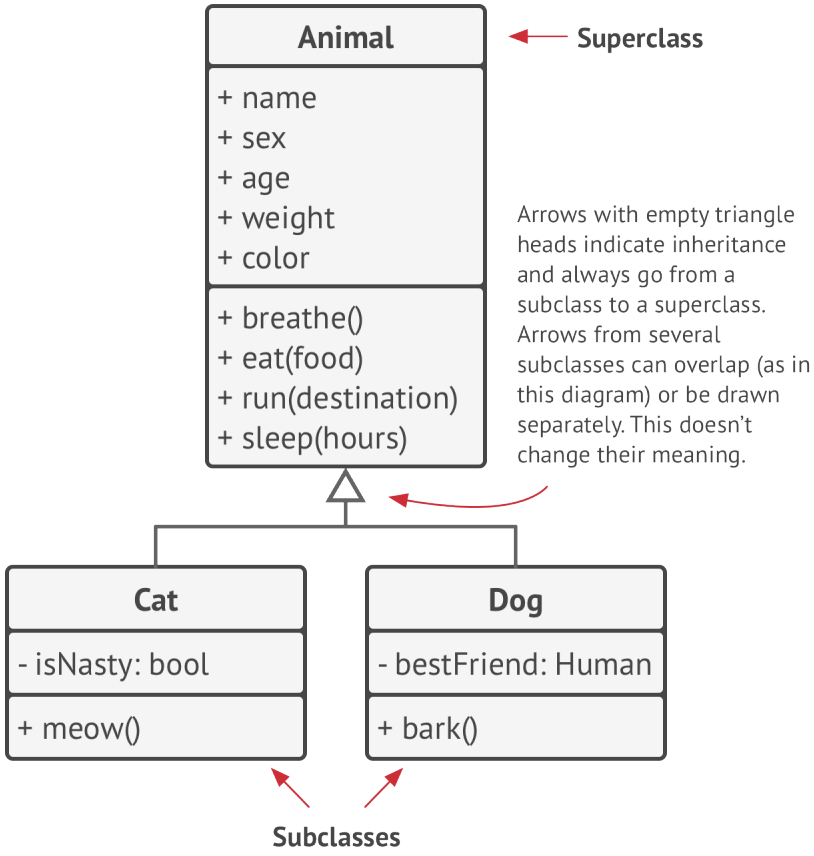
So a *class* is like a blueprint that defines the structure for *objects*, which are concrete instances of that class.

Class hierarchies

Everything fine and dandy when we talk about one class. Naturally, a real program contains more than a single class. Some of these classes might be organized into **class hierarchies**. Let's find out what that means.

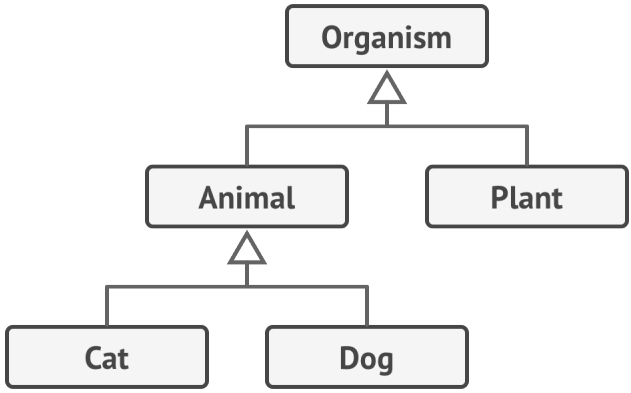
Say your neighbor has a dog called Fido. It turns out, dogs and cats have a lot in common: name, sex, age, and color are attributes of both dogs and cats. Dogs can breathe, sleep and run the same way cats do. So it seems that we can define the base `Animal` class that would list the common attributes and behaviors.

A parent class, like the one we've just defined, is called a **superclass**. Its children are **subclasses**. Subclasses inherit state and behavior from their parent, defining only attributes or behaviors that differ. Thus, the `Cat` class would have the `meow` method, and the `Dog` class the `bark` method.



UML diagram of a class hierarchy. All classes in this diagram are part of the `Animal` class hierarchy.

Assuming that we have a related business requirement, we can go even further and extract a more general class for all living `Organisms` which will become a superclass for `Animals` and `Plants`. Such a pyramid of classes is a **hierarchy**. In such a hierarchy, the `Cat` class inherits everything from both the `Animal` and `Organism` classes.

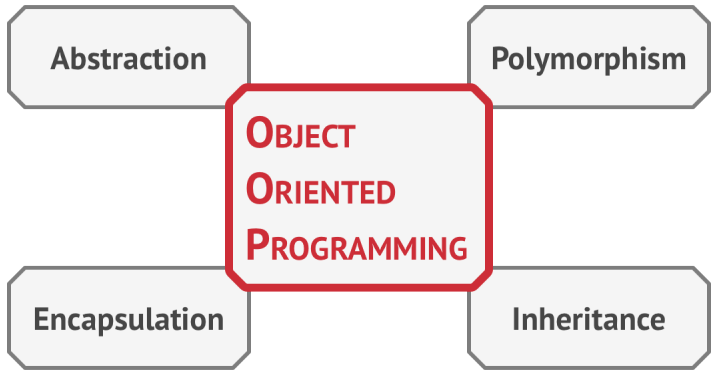


Classes in a UML diagram can be simplified if it's more important to show their relations than their contents.

Subclasses can override the behavior of methods that they inherit from parent classes. A subclass can either completely replace the default behavior or just enhance it with some extra stuff.

Pillars of OOP

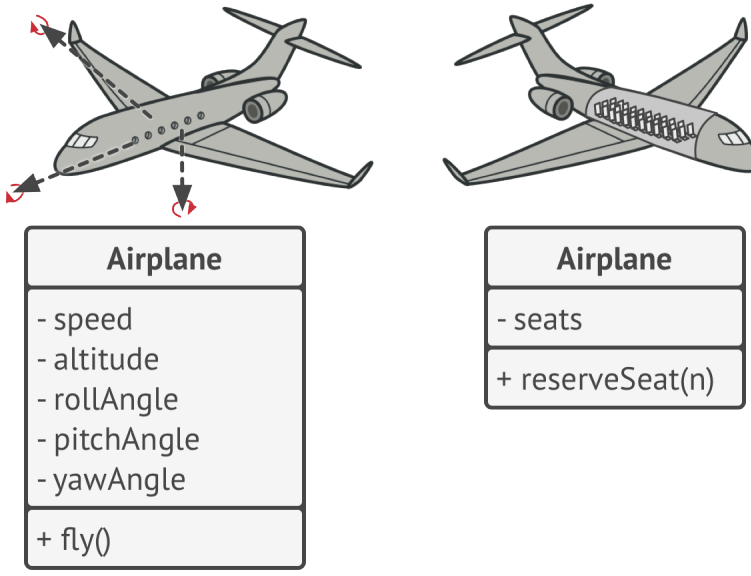
Object-oriented programming is based on four pillars, concepts that differentiate it from other programming paradigms.



Abstraction

Most of the time when you're creating a program with OOP, you shape objects of the program based on real-world objects. However, objects of the program don't represent the originals with 100% accuracy (and it's rarely required that they do). Instead, your objects only *model* attributes and behaviors of real objects in a specific context, ignoring the rest.

For example, an `Airplane` class could probably exist in both a flight simulator and a flight booking application. But in the former case, it would hold details related to the actual flight, whereas in the latter class you would care only about the seat map and which seats are available.



Different models of the same real-world object.

Abstraction is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant to this context with high accuracy and omits all the rest.

Encapsulation

To start a car engine, you only need to turn a key or press a button. You don't need to connect wires under the hood, rotate the crankshaft and cylinders, and initiate the power cycle of the engine. These details are hidden under the hood of the car. You have only a simple interface: a start switch, a steering wheel and some pedals. This illustrates how each object has an **interface**—a public part of an object, open to interactions with other objects.

Encapsulation is the ability of an object to hide parts of its state and behaviors from other objects, exposing only a limited interface to the rest of the program.

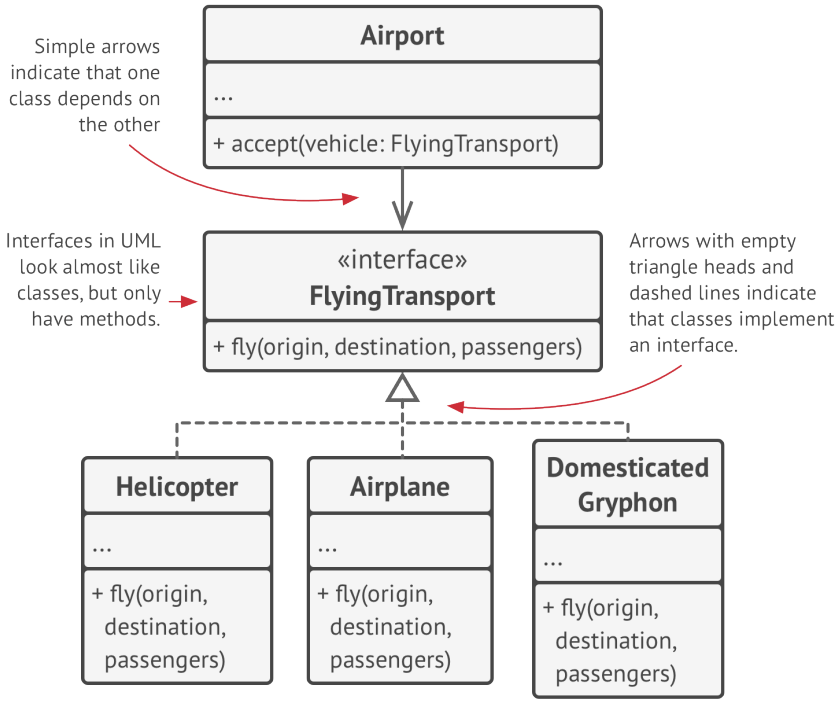
To encapsulate something means to make it `private`, and thus accessible only from within of the methods of its own class. There's a little bit less restrictive mode called `protected` that makes a member of a class available to subclasses as well.

Interfaces and abstract classes/methods of most programming languages are based on the concepts of abstraction and encapsulation. In modern object-oriented programming languages, the interface mechanism (usually declared with the `interface` or `protocol` keyword) lets you define contracts of interaction between objects. That's one of the reasons why the interfaces only care about behaviors of objects, and why you can't declare a field in an interface.

> The fact that the word *interface* stands for a public part of an object, while there's also the `interface` type in most programming languages, is very confusing. I'm with you on that.

Imagine that you have a `FlyingTransport` interface with a method `fly(origin, destination, passengers)`. When designing an air transportation simulator, you could restrict the `Airport` class to work only with objects that implement the `FlyingTransport` interface. After this, you can be sure

that any object passed to an airport object, whether it's an `Airplane`, a `Helicopter` or a freaking `DomesticatedGryphon` would be able to arrive or depart from this type of airport.



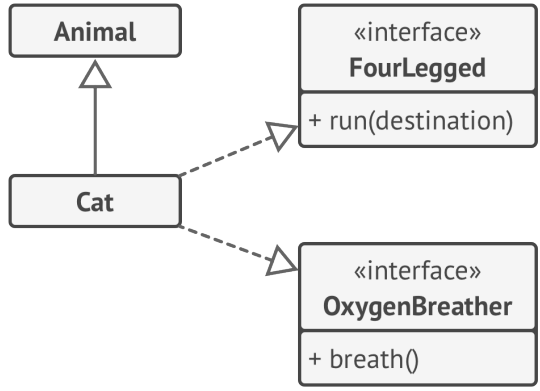
UML diagram of several classes implementing an interface.

You could change the implementation of the `fly` method in these classes in any way you want. As long as the signature of the method remains the same as declared in the interface, all instances of the `Airport` class can work with your flying objects just fine.

Inheritance

Inheritance is the ability to build new classes on top of existing ones. The main benefit of inheritance is code reuse. If you want to create a class that's slightly different from an existing one, there's no need to duplicate code. Instead, you extend the existing class and put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.

The consequence of using inheritance is that subclasses have the same interface as their parent class. You can't hide a method in a subclass if it was declared in the superclass. You must also implement all abstract methods, even if they don't make sense for your subclass.



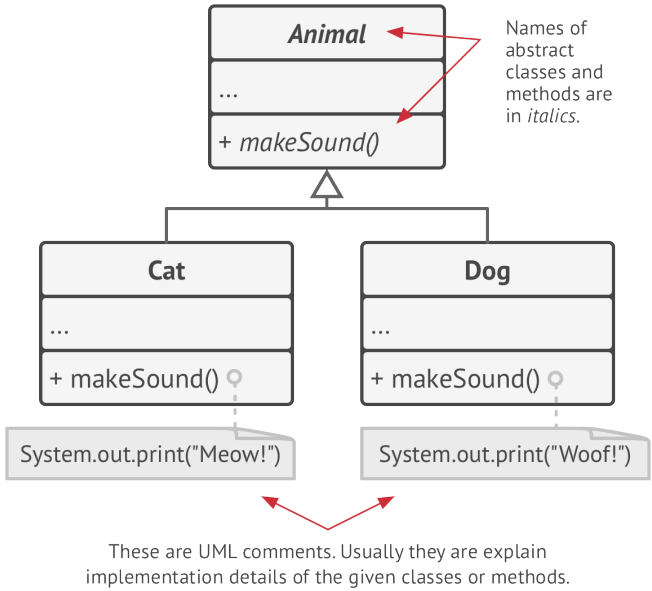
UML diagram of extending a single class versus implementing multiple interfaces at the same time.

In most programming languages a subclass can extend only one superclass. On the other hand, any class can implement several interfaces at the same time. But, as I mentioned before,

if a superclass implements an interface, all of its subclasses must also implement it.

Polymorphism

Let's look at some animal examples. Most `Animals` can make sounds. We can anticipate that all subclasses will need to override the base `makeSound` method so each subclass can emit the correct sound; therefore we can declare it *abstract* right away. This lets us omit any default implementation of the method in the superclass, but force all subclasses to come up with their own.



Imagine that we've put several cats and dogs into a large bag. Then, with closed eyes, we take the animals one-by-one out of

the bag. After taking an animal from the bag, we don't know for sure what it is. However, if we cuddle it hard enough, the animal will emit a specific sound of joy, depending on its concrete class.

```
bag = [new Cat(), new Dog()];

foreach (Animal a : bag)
  a.makeSound()

// Meow!
// Woof!
```

The program doesn't know the concrete type of the object contained inside the `a` variable; but, thanks to the special mechanism called *polymorphism*, the program can trace down the subclass of the object whose method is being executed and run the appropriate behavior.

Polymorphism is the ability of a program to detect the real class of an object and call its implementation even when its real type is unknown in the current context.

You can also think of polymorphism as the ability of an object to "pretend" to be something else, usually a class it extends or an interface it implements. In our example, the dogs and cats in the bag were pretending to be generic animals.

Relations Between Objects

In addition to *inheritance* and *implementation* that we've already seen, there are other types of relations between objects that we haven't talked about yet.

UML Association. Professor communicates with students.

Association is a type of relationship in which one object uses or interacts with another. In UML diagrams the association relationship is shown by a simple arrow drawn from an object and pointing to the object it uses. By the way, having a bi-directional association is a completely normal thing. In this case, the arrow has a point at each end.

In general, you use an association to represent something like a field in a class. The link is always there, in that you can always ask an order for its customer. It need not actually be a field, if you are modeling from a more interface perspective, it can just indicate the presence of a method that will return the order's customer.

UML Dependency. Professor depends on salary.

Dependency is a weaker variant of association that usually implies that there's no permanent link between objects. Dependency typically (but not always) implies that an object accepts another object as a method parameter, instantiates, or uses another object. Here's how you can spot a dependency between classes: a dependency exists between two classes if changes to the definition of one class result in modifications in another class.

UML Composition. University consists of departments.

Composition is a "whole-part" relationship between two objects, one of which is composed of one or more instances of the other. The distinction between this relation and others is that the component can only exist as a part of the container. In UML the composition relationship is shown by a line with a filled diamond at the container end and an arrow at the end pointing toward the component.

While we talk about relations between objects, keep in mind that UML represents relations between *classes*. It means that a university object might consist of multiple departments even though you see just one “block” for each entity in the diagram. UML notation can represent quantities on both sides of relationships, but it’s okay to omit them if the quantities are clear from the context.

UML Aggregation. Department contains professors.

Aggregation is a less strict variant of composition, where one object merely contains a reference to another. The container doesn’t control the life cycle of the component. The component can exist without the container and can be linked to several containers at the same time. In UML the aggregation relationship is drawn the same as for composition, but with an empty diamond at the arrow’s base.

INTRODUCTION TO PATTERNS

What's a Design Pattern?

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

What does the pattern consist of?

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

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

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

Classification of patterns

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

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

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

In addition, all patterns can be categorized by their *intent*, or purpose. This book covers three main groups of patterns:

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.

- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Who invented patterns?

That's a good, but not a very accurate, question. Design patterns aren't obscure, sophisticated concepts—quite the opposite. Patterns are typical solutions to common problems in object-oriented design. When a solution gets repeated over and over in various projects, someone eventually puts a name

to it and describes the solution in detail. That's basically how a pattern gets discovered.

The concept of patterns was first described by Christopher Alexander in *A Pattern Language: Towns, Buildings, Construction*[1]. The book describes a "language" for designing the urban environment. The units of this language are patterns. They may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. In 1995, they published *Design Patterns: Elements of Reusable Object-Oriented Software*[2], in which they applied the concept of design patterns to programming. The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly. Due to its lengthy name, people started to call it "the book by the gang of four" which was soon shortened to simply "the GOF book".

Since then, dozens of other object-oriented patterns have been discovered. The "pattern approach" became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

---

1. *A Pattern Language: Towns, Buildings, Construction*: **https://refactoring.guru/pattern-language-book**
2. *Design Patterns: Elements of Reusable Object-Oriented Software*: **https://refactoring.guru/gof-book**

Why Should I Learn Patterns?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

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

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

SOFTWARE DESIGN PRINCIPLES

Features of Good Design

Before we proceed to the actual patterns, let's discuss the process of designing software architecture: things to aim for and things you'd better avoid.

Code reuse

Cost and time are two of the most valuable metrics when developing any software product. Less time in development means entering the market earlier than competitors. Lower development costs mean more money is left for marketing and a broader reach to potential customers.

Code reuse is one of the most common ways to reduce development costs. The intent is pretty obvious: instead of developing something over and over from scratch, why don't we reuse existing code in new projects?

The idea looks great on paper, but it turns out that making existing code work in a new context usually takes extra effort. Tight coupling between components, dependencies on concrete classes instead of interfaces, hardcoded operations—all of this reduces flexibility of the code and makes it harder to reuse it.

Using design patterns is one way to increase flexibility of software components and make them easier to reuse. However,

this sometimes comes at the price of making the components more complicated.

Here's a piece of wisdom from Erich Gamma [1], one of the founding fathers of design patterns, about the role of design patterns in code reuse:

> I see three levels of reuse.
>
> At the lowest level, you reuse classes: class libraries, containers, maybe some class "teams" like container/iterator.
>
> Frameworks are at the highest level. They really try to distill your design decisions. They identify the key abstractions for solving a problem, represent them by classes and define relationships between them. JUnit is a small framework, for example. It is the "Hello, world" of frameworks. It has `Test`, `TestCase`, `TestSuite` and relationships defined.
>
> A framework is typically larger-grained than just a single class. Also, you hook into frameworks by subclassing somewhere. They use the so-called Hollywood principle of "don't call us, we'll call you." The framework lets you define your custom behavior, and it will call you when it's your turn to do something. Same with JUnit, right? It calls you when it wants to execute a test for you, but the rest happens in the framework.

1. Erich Gamma on Flexibility and Reuse: **https://refactoring.guru/gamma-interview**

> There also is a middle level. This is where I see patterns. Design patterns are both smaller and more abstract than frameworks. They're really a description about how a couple of classes can relate to and interact with each other. The level of reuse increases when you move from classes to patterns and finally frameworks.
>
> What is nice about this middle layer is that patterns offer reuse in a way that is less risky than frameworks. Building a framework is high-risk and a significant investment. Patterns let you reuse design ideas and concepts independently of concrete code.

Extensibility

Change is the only constant thing in a programmer's life.

- You released a video game for Windows, but now people ask for a macOS version.
- You created a GUI framework with square buttons, but several months later round buttons become a trend.
- You designed a brilliant e-commerce website architecture, but just a month later customers ask for a feature that would let them accept phone orders.

Each software developer has dozens of similar stories. There are several reasons why this happens.

First, we understand the problem better once we start to solve it. Often by the time you finish the first version of an app, you're ready to rewrite it from scratch because now you understand many aspects of the problem much better. You have also grown professionally, and your own code now looks like crap.

Something beyond your control has changed. This is why so many dev teams pivot from their original ideas into something new. Everyone who relied on Flash in an online application has been reworking or migrating their code as browser after browser drops support for Flash.

The third reason is that the goalposts move. Your client was delighted with the current version of the application, but now sees eleven "little" changes he'd like so it can do other things he never mentioned in the original planning sessions. These aren't frivolous changes: your excellent first version has shown him that even more is possible.

> There's a bright side: if someone asks you to change something in your app, that means someone still cares about it.

That's why all seasoned developers try to provide for possible future changes when designing an application's architecture.

Design Principles

What is good software design? How would you measure it? What practices would you need to follow to achieve it? How can you make your architecture flexible, stable and easy to understand?

These are the great questions; but, unfortunately, the answers are different depending on the type of application you're building. Nevertheless, there are several universal principles of software design that might help you answer these questions for your own project. Most of the design patterns listed in this book are based on these principles.

Encapsulate What Varies

> Identify the aspects of your application that vary and separate them from what stays the same.

The main goal of this principle is to minimize the effect caused by changes.

Imagine that your program is a ship, and changes are hideous mines that linger under water. Struck by the mine, the ship sinks.

Knowing this, you can divide the ship's hull into independent compartments that can be safely sealed to limit damage to a single compartment. Now, if the ship hits a mine, the ship as a whole remains afloat.

In the same way, you can isolate the parts of the program that vary in independent modules, protecting the rest of the code from adverse effects. As a result, you spend less time getting the program back into working shape, implementing and testing the changes. The less time you spend making changes, the more time you have for implementing features.

Encapsulation on a method level

Say you're making an e-commerce website. Somewhere in your code, there's a `getOrderTotal` method that calculates a grand total for the order, including taxes.

We can anticipate that tax-related code might need to change in the future. The tax rate depends on the country, state or even city where the customer resides, and the actual formula may change over time due to new laws or regulations. As a result, you'll need to change the `getOrderTotal` method quite often. But even the method's name suggests that it doesn't care about *how* the tax is calculated.

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // US sales tax
  else if (order.country == "EU"):
    total += total * 0.20 // European VAT

  return total
```

BEFORE: tax calculation code is mixed with the rest of the method's code.

You can extract the tax calculation logic into a separate method, hiding it from the original method.

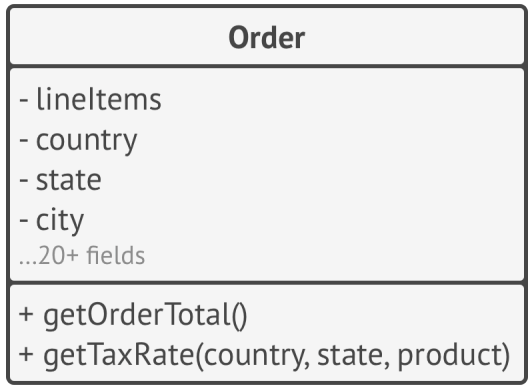
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    return 0.07 // US sales tax
  else if (country == "EU")
    return 0.20 // European VAT
  else
    return 0
```

AFTER: you can get the tax rate by calling a designated method.

Tax-related changes become isolated inside a single method. Moreover, if the tax calculation logic becomes too complicated, it's now easier to move it to a separate class.

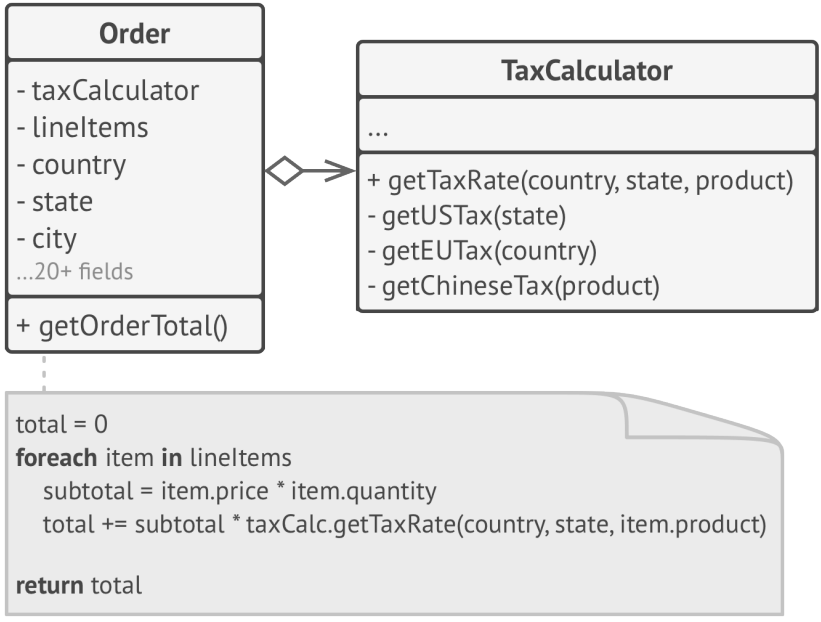
Encapsulation on a class level

Over time you might add more and more responsibilities to a method which used to do a simple thing. These added behaviors often come with their own helper fields and methods that eventually blur the primary responsibility of the containing class. Extracting everything to a new class might make things much more clear and simple.



BEFORE: calculating tax in `Order` class.

Objects of the `Order` class delegate all tax-related work to a special object that does just that.



AFTER: tax calculation is hidden from the order class.

Program to an Interface, not an Implementation

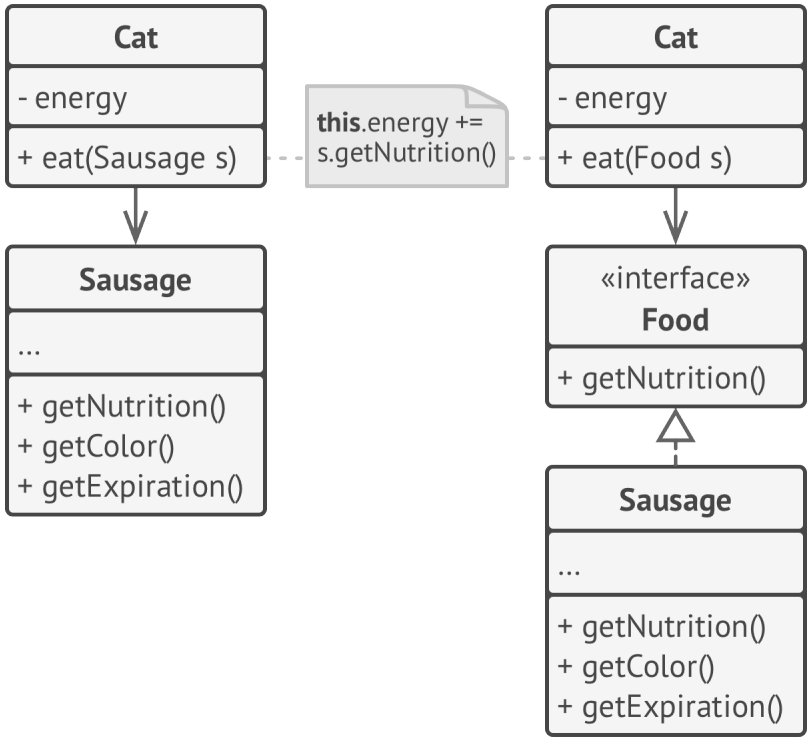
> Program to an interface, not an implementation. Depend on abstractions, not on concrete classes.

You can tell that the design is flexible enough if you can easily extend it without breaking any existing code. Let's make sure that this statement is correct by looking at another cat example. A `Cat` that can eat any food is more flexible than one that can eat just sausages. You can still feed the first cat with sausages because they are a subset of "any food"; however, you can extend that cat's menu with any other food.

When you want to make two classes collaborate, you can start by making one of them dependent on the other. Hell, I often start by doing that myself. However, there's another, more flexible way to set up collaboration between objects:

1. Determine what exactly one object needs from the other: which methods does it execute?
2. Describe these methods in a new interface or abstract class.
3. Make the class that is a dependency implement this interface.
4. Now make the second class dependent on this interface rather than on the concrete class. You still can make it work with

objects of the original class, but the connection is now much more flexible.

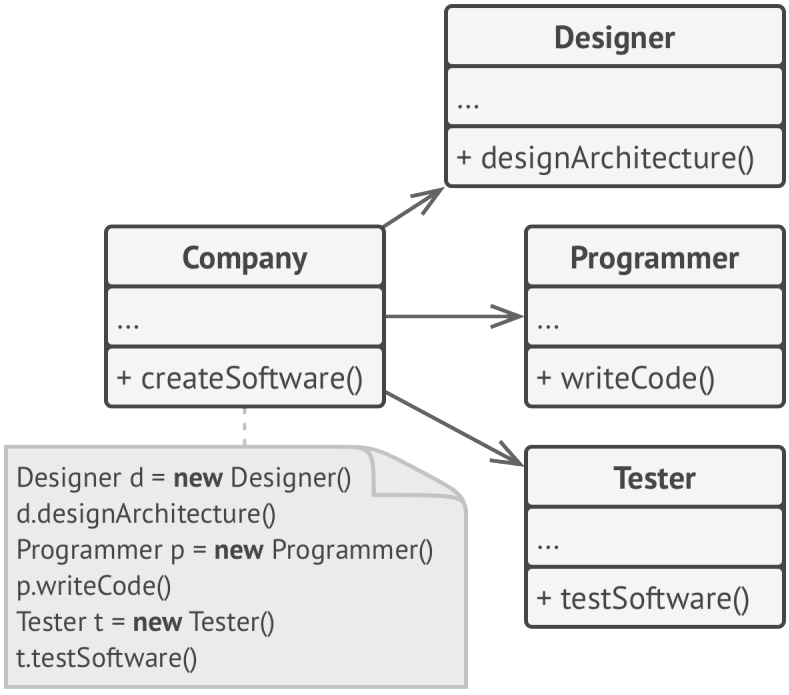


Before and after extracting the interface. The code on the right is more flexible than the code on the left, but it's also more complicated.

After making this change, you won't probably feel any immediate benefit. On the contrary, the code has become more complicated than it was before. However, if you feel that this might be a good extension point for some extra functionality, or that some other people who use your code might want to extend it here, then go for it.

Example

Let's look at another example which illustrates that working with objects through interfaces might be more beneficial than depending on their concrete classes. Imagine that you're creating a software development company simulator. You have different classes that represent various employee types.

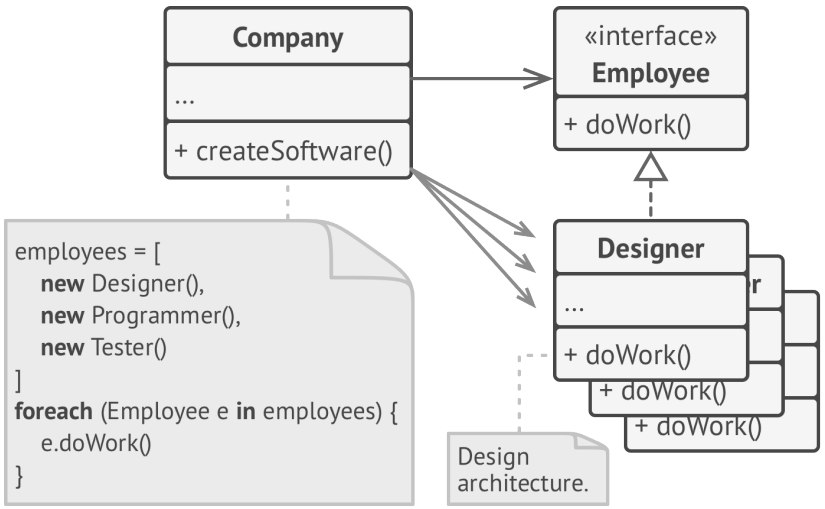


BEFORE: all classes are tightly coupled.

In the beginning, the `Company` class is tightly coupled to concrete classes of employees. However, despite the difference in their implementations, we can generalize various work-related

methods and then extract a common interface for all employee classes.

After doing that, we can apply polymorphism inside the `Company` class, treating various employee objects via the `Employee` interface.

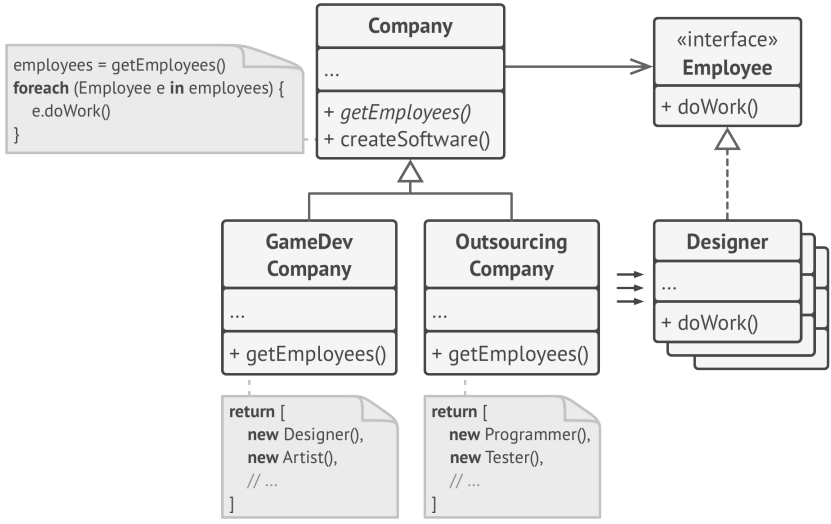


BETTER: polymorphism helped us simplify the code, but the rest of the `Company` class still depends on the concrete employee classes.

The `Company` class remains coupled to the employee classes. This is bad because if we introduce new types of companies that work with other types of employees, we'll need to override most of the `Company` class instead of reusing that code.

To solve this problem, we could declare the method for getting employees as *abstract*. Each concrete company will imple-

ment this method differently, creating only those employees that it needs.



AFTER: the primary method of the `Company` class is independent from concrete employee classes. Employee objects are created in concrete company subclasses.

After this change, the `Company` class has become independent from various employee classes. Now you can extend this class and introduce new types of companies and employees while still reusing a portion of the base company class. Extending the base company class doesn't break any existing code that already relies on it.

By the way, you've just seen applying a design pattern in action! That was an example of the *Factory Method* pattern. Don't worry: we'll discuss it later in detail.

Favor Composition Over Inheritance

Inheritance is probably the most obvious and easy way of reusing code between classes. You have two classes with the same code. Create a common base class for these two and move the similar code into it. Piece of cake!

Unfortunately, inheritance comes with caveats that often become apparent only after your program already has tons of classes and changing anything is pretty hard. Here's a list of those problems.

- **A subclass can't reduce the interface of the superclass**. You have to implement all abstract methods of the parent class even if you won't be using them.

- **When overriding methods you need to make sure that the new behavior is compatible with the base one**. It's important because objects of the subclass may be passed to any code that expects objects of the superclass and you don't want that code to break.

- **Inheritance breaks encapsulation of the superclass** because the internal details of the parent class become available to the subclass. There might be an opposite situation where a programmer makes a superclass aware of some details of subclasses for the sake of making further extension easier.

- **Subclasses are tightly coupled to superclasses**. Any change in a superclass may break the functionality of subclasses.

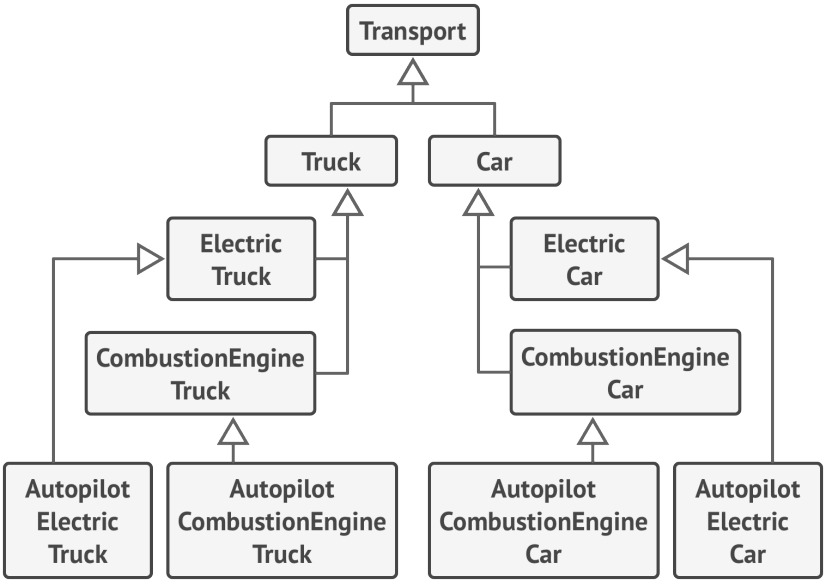
- **Trying to reuse code through inheritance can lead to creating parallel inheritance hierarchies**. Inheritance usually takes place in a single dimension. But whenever there are two or more dimensions, you have to create lots of class combinations, bloating the class hierarchy to a ridiculous size.

  There's an alternative to inheritance called *composition*. Whereas inheritance represents the "is a" relationship between classes (a car *is a* transport), composition represents the "has a" relationship (a car *has an* engine).

  I should mention that this principle also applies to aggregation—a more relaxed variant of composition where one object may have a reference to the other one but doesn't manage its lifecycle. Here's an example: a car *has a* driver, but he or she may use another car or just walk *without the car*.

Example

Imagine that you need to create a catalog app for a car manufacturer. The company makes both cars and trucks; they can be either electric or gas; all models have either manual controls or an autopilot.

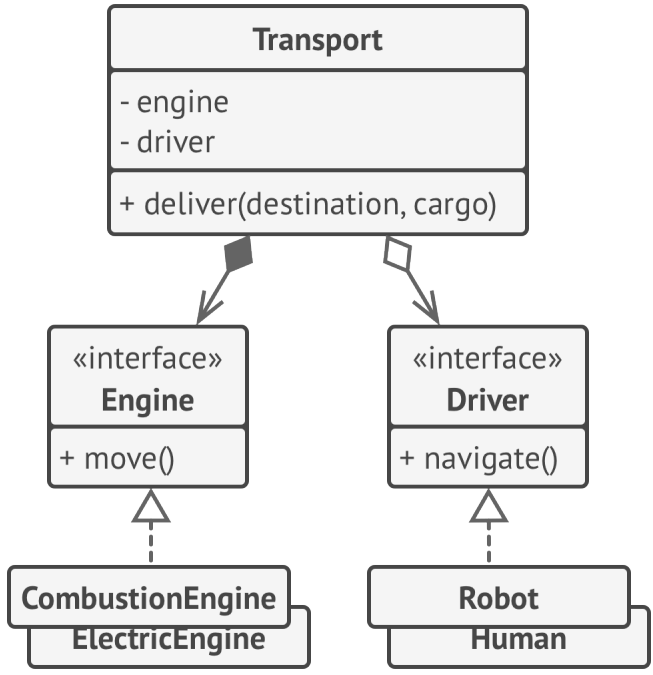


INHERITANCE: extending a class in several dimensions (cargo type × engine type × navigation type) may lead to a combinatorial explosion of subclasses.

As you see, each additional parameter results in multiplying the number of subclasses. There's a lot of duplicate code between subclasses because a subclass can't extend two classes at the same time.

You can solve this problem with composition. Instead of car objects implementing a behavior on their own, they can delegate it to other objects.

The added benefit is that you can replace a behavior at runtime. For instance, you can replace an engine object linked to a car object just by assigning a different engine object to the car.



COMPOSITION: different "dimensions" of functionality extracted to their own class hierarchies.

This structure of classes resembles the *Strategy* pattern, which we'll go over later in this book.

SOLID Principles

Now that you know the basic design principles, let's take a look at five that are commonly known as the SOLID principles. Robert Martin introduced them in the book *Agile Software Development, Principles, Patterns, and Practices*[1].

SOLID is a mnemonic for five design principles intended to make software designs more understandable, flexible and maintainable.

As with everything in life, using these principles mindlessly can cause more harm than good. The cost of applying these principles into a program's architecture might be making it more complicated than it should be. I doubt that there's a successful software product in which all of these principles are applied at the same time. Striving for these principles is good, but always try to be pragmatic and don't take everything written here as dogma.

1. *Agile Software Development, Principles, Patterns, and Practices*: **https://refactoring.guru/principles-book**

Single Responsibility Principle

> A class should have just one reason to change.

Try to make every class responsible for a single part of the functionality provided by the software, and make that responsibility entirely encapsulated by (you can also say *hidden within*) the class.

The main goal of this principle is reducing complexity. You don't need to invent a sophisticated design for a program that only has about 200 lines of code. Make a dozen methods pretty, and you'll be fine.

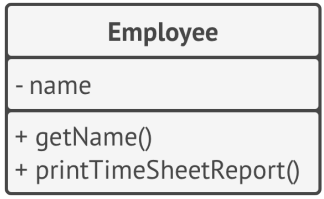
The real problems emerge when your program constantly grows and changes. At some point classes become so big that you can no longer remember their details. Code navigation slows down to a crawl, and you have to scan through whole classes or even an entire program to find specific things. The number of entities in program overflows your brain stack, and you feel that you're losing control over the code.

There's more: if a class does too many things, you have to change it every time one of these things changes. While doing that, you're risking breaking other parts of the class which you didn't even intend to change.

If you feel that it's becoming hard to focus on specific aspects of the program one at a time, remember the single responsibility principle and check whether it's time to divide some classes into parts.

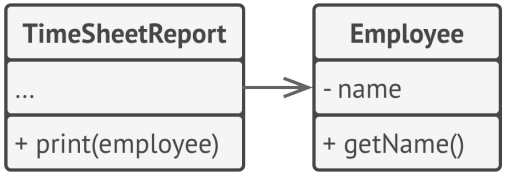
Example

The `Employee` class has several reasons to change. The first reason might be related to the main job of the class: managing employee data. However, there's another reason: the format of the timesheet report may change over time, requiring you to change the code within the class.



BEFORE: the class contains several different behaviors.

Solve the problem by moving the behavior related to printing timesheet reports into a separate class. This change lets you move other report-related stuff to the new class.



AFTER: the extra behavior is in its own class.

Open/Closed Principle

> Classes should be open for extension but closed for modification.

The main idea of this principle is to keep existing code from breaking when you implement new features.

A class is *open* if you can extend it, produce a subclass and do whatever you want with it—add new methods or fields, override base behavior, etc. Some programming languages let you restrict further extension of a class with special keywords, such as `final`. After this, the class is no longer open. At the same time, the class is *closed* (you can also say *complete*) if it's 100% ready to be used by other classes—its interface is clearly defined and won't be changed in the future.

When I first learned about this principle, I was confused because the words *open* & *closed* sound mutually exclusive. But in terms of this principle, a class can be both open (for extension) and closed (for modification) at the same time.

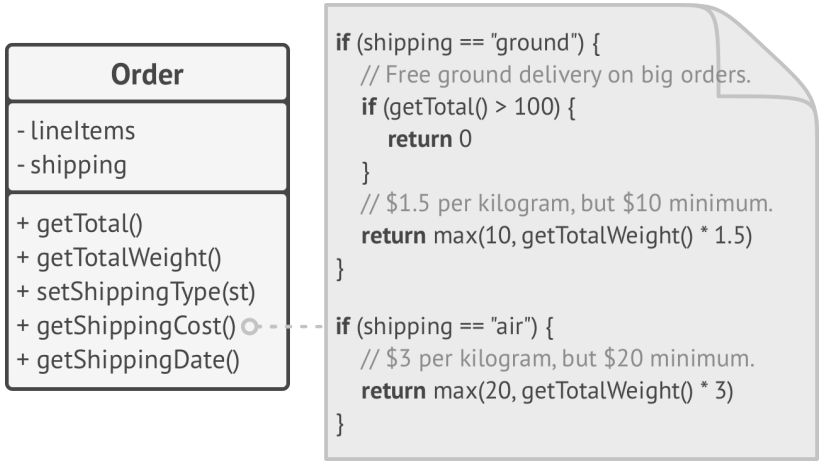
If a class is already developed, tested, reviewed, and included in some framework or otherwise used in an app, trying to mess with its code is risky. Instead of changing the code of the class directly, you can create a subclass and override parts of

the original class that you want to behave differently. You'll achieve your goal but also won't break any existing clients of the original class.

This principle isn't meant to be applied for all changes to a class. If you know that there's a bug in the class, just go on and fix it; don't create a subclass for it. A child class shouldn't be responsible for the parent's issues.

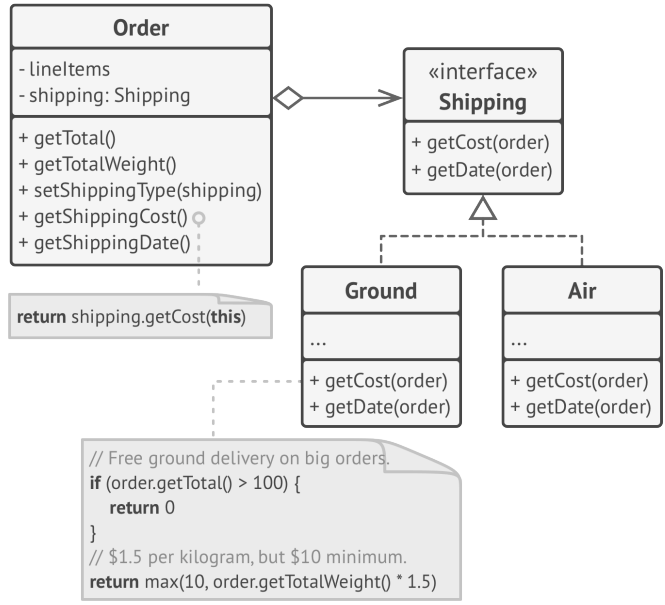
Example

You have an e-commerce application with an `Order` class that calculates shipping costs and all shipping methods are hard-coded inside the class. If you need to add a new shipping method, you have to change the code of the `Order` class and risk breaking it.



BEFORE: you have to change the `Order` class whenever you add a new shipping method to the app.

You can solve the problem by applying the *Strategy* pattern. Start by extracting shipping methods into separate classes with a common interface.



AFTER: adding a new shipping method doesn't require changing existing classes.

Now when you need to implement a new shipping method, you can derive a new class from the `Shipping` interface without touching any of the `Order` class' code. The client code of the `Order` class will link orders with a shipping object of the new class whenever the user selects this shipping methods in the UI.

As a bonus, this solution let you move the delivery time calculation to more relevant classes, according to the *single responsibility principle*.

Liskov Substitution Principle[1]

> When extending a class, remember that you should be able to pass objects of the subclass in place of objects of the parent class without breaking the client code.

This means that the subclass should remain compatible with the behavior of the superclass. When overriding a method, extend the base behavior rather than replacing it with something else entirely.

The substitution principle is a set of checks that help predict whether a subclass remains compatible with the code that was able to work with objects of the superclass. This concept is critical when developing libraries and frameworks because your classes are going to be used by other people whose code you can't directly access and change.

Unlike other design principles which are wide open for interpretation, the substitution principle has a set of formal requirements for subclasses, and specifically for their methods. Let's go over this checklist in detail.

1. This principle is named by Barbara Liskov, who defined it in 1987 in her work *Data abstraction and hierarchy*: **https://refactoring.guru/liskov/dah**

- **Parameter types in a method of a subclass should *match* or be *more abstract* than parameter types in the method of the superclass.** Sounds confusing? Let's have an example.

  - Say there's a class with a method that's supposed to feed cats: `feed(Cat c)`. Client code always passes cat objects into this method.

  - **Good:** Say you created a subclass that overrode the method so that it can feed any animal (a superclass of cats): `feed(Animal c)`. Now if you pass an object of this subclass instead of an object of the superclass to the client code, everything would still work fine. The method can feed all animals, so it can still feed any cat passed by the client.

  - **Bad:** You created another subclass and restricted the feeding method to only accept Bengal cats (a subclass of cats): `feed(BengalCat c)`. What will happen to the client code if you link it with an object like this instead of with the original class? Since the method can only feed a specific breed of cats, it won't serve generic cats passed by the client, breaking all related functionality.

- **The return type in a method of a subclass should *match* or be a *subtype* of the return type in the method of the superclass.** As you can see, requirements for a return type are inverse to requirements for parameter types.

  - Say you have a class with a method `buyCat(): Cat`. The client code expects to receive any cat as a result of executing this method.

  - **Good:** A subclass overrides the method as follows: `buyCat(): BengalCat`. The client gets a Bengal cat, which is still a cat, so everything is okay.

  - **Bad:** A subclass overrides the method as follows: `buyCat(): Animal`. Now the client code breaks since it receives an unknown generic animal (an alligator? a bear?) that doesn't fit a structure designed for a cat.

  Another anti-example comes from the world of programming languages with dynamic typing: the base method returns a string, but the overridden method returns a number.

- **A method in a subclass shouldn't throw types of exceptions which the base method isn't expected to throw.** In other words, types of exceptions should *match* or be *subtypes* of the ones that the base method is already able to throw. This rule comes from the fact that `try–catch` blocks in the client code target specific types of exceptions which the base method is likely to throw. Therefore, an unexpected exception might slip through the defensive lines of the client code and crash the entire application.

> In most modern programming languages, especially statically typed ones (Java, C#, and others), these rules are built into the language. You won't be able to compile a program that violates these rules.

- **A subclass shouldn't strengthen pre-conditions.** For example, the base method has a parameter with type `int`. If a subclass overrides this method and requires that the value of an argument passed to the method should be positive (by throwing an exception if the value is negative), this strengthens the pre-conditions. The client code, which used to work fine when passing negative numbers into the method, now breaks if it starts working with an object of this subclass.

- **A subclass shouldn't weaken post-conditions.** Say you have a class with a method that works with a database. A method of the class is supposed to always close all opened database connections upon returning a value.

  You created a subclass and changed it so that database connections remain open so you can reuse them. But the client might not know anything about your intentions. Because it expects the methods to close all the connections, it may simply terminate the program right after calling the method, polluting a system with ghost database connections.

- **Invariants of a superclass must be preserved.** This is probably the least formal rule of all. *Invariants* are conditions in which

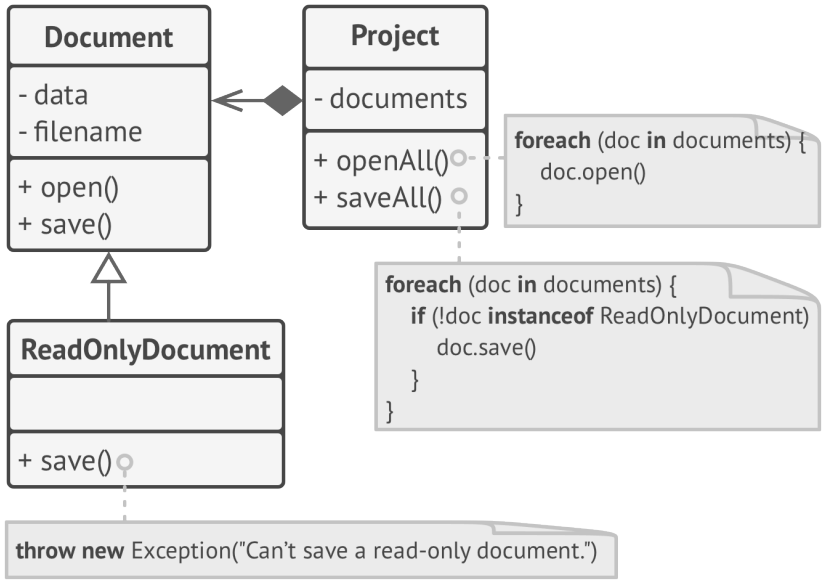
an object makes sense. For example, invariants of a cat are having four legs, a tail, ability to meow, etc. The confusing part about invariants is that while they can be defined explicitly in the form of interface contracts or a set of assertions within methods, they could also be implied by certain unit tests and expectations of the client code.

The rule on invariants is the easiest to violate because you might misunderstand or not realize all of the invariants of a complex class. Therefore, the safest way to extend a class is to introduce new fields and methods, and not mess with any existing members of the superclass. Of course, that's not always doable in real life.

- **A subclass shouldn't change values of private fields of the superclass.** *What? How's that even possible?* It turns out some programming languages let you access private members of a class via reflection mechanisms. Other languages (Python, JavaScript) don't have any protection for the private members at all.

Example

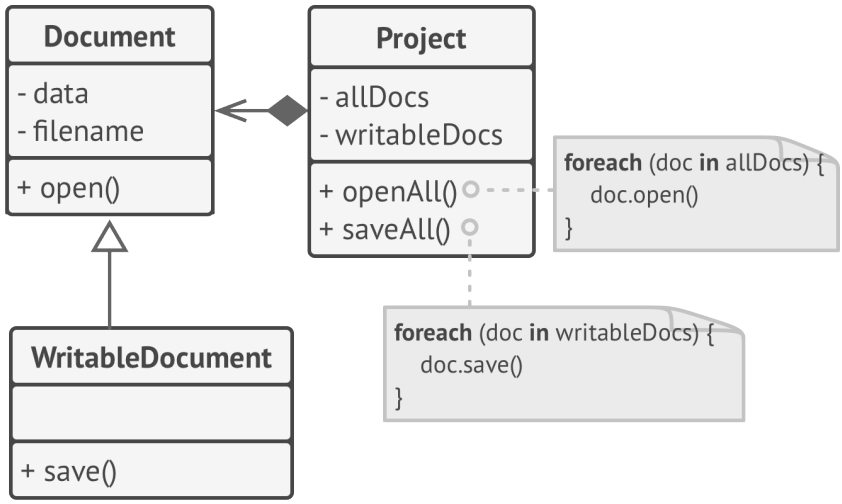
Let's look at an example of a hierarchy of document classes that violates the substitution principle.



BEFORE: saving doesn't make sense in a read-only document, so the subclass tries to solve it by resetting the base behavior in the overridden method.

The `save` method in the `ReadOnlyDocuments` subclass throws an exception if someone tries to call it. The base method doesn't have this restriction. This means that the client code will break if we don't check the document type before saving it.

The resulting code also violates the open/closed principle, since the client code becomes dependent on concrete classes of documents. If you introduce a new document subclass, you'll need to change the client code to support it.



AFTER: the problem is solved after making the read-only document class the base class of the hierarchy.

You can solve the problem by redesigning the class hierarchy: a subclass should extend the behavior of a superclass, therefore the read-only document becomes the base class of the hierarchy. The writable document is now a subclass which extends the base class and adds the saving behavior.

Interface Segregation Principle

> Clients shouldn't be forced to depend on methods they do not use.

Try to make your interfaces narrow enough that client classes don't have to implement behaviors they don't need.

According to the interface segregation principle, you should break down "fat" interfaces into more granular and specific ones. Clients should implement only those methods that they really need. Otherwise, a change to a "fat" interface would break even clients that don't use the changed methods.

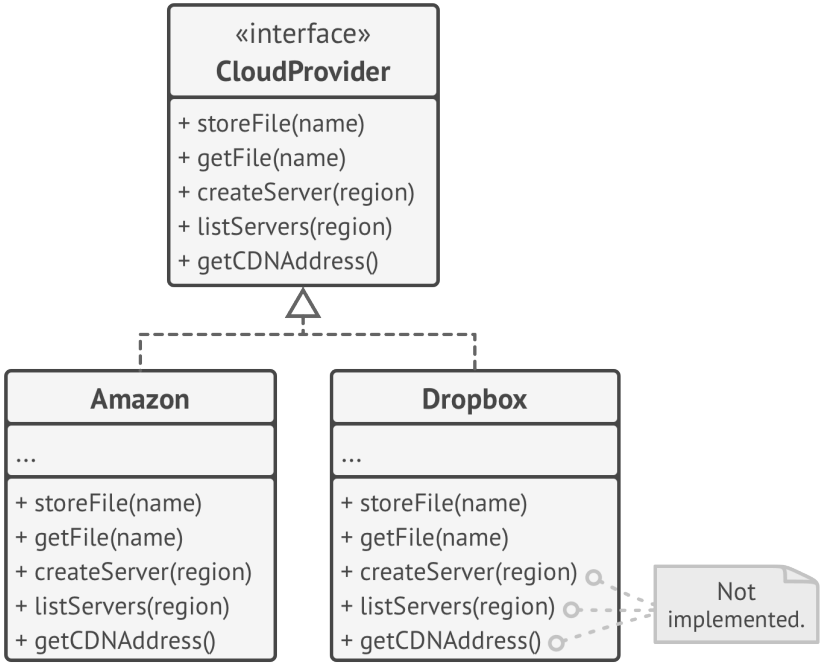
Class inheritance lets a class have just one superclass, but it doesn't limit the number of interfaces that the class can implement at the same time. Hence, there's no need to cram tons of unrelated methods to a single interface. Break it down into several more refined interfaces—you can implement them all in a single class if needed. However, some classes may be fine with implementing just one of them.

Example

Imagine that you created a library that makes it easy to integrate apps with various cloud computing providers. While in

the initial version it only supported Amazon Cloud, it covered the full set of cloud services and features.

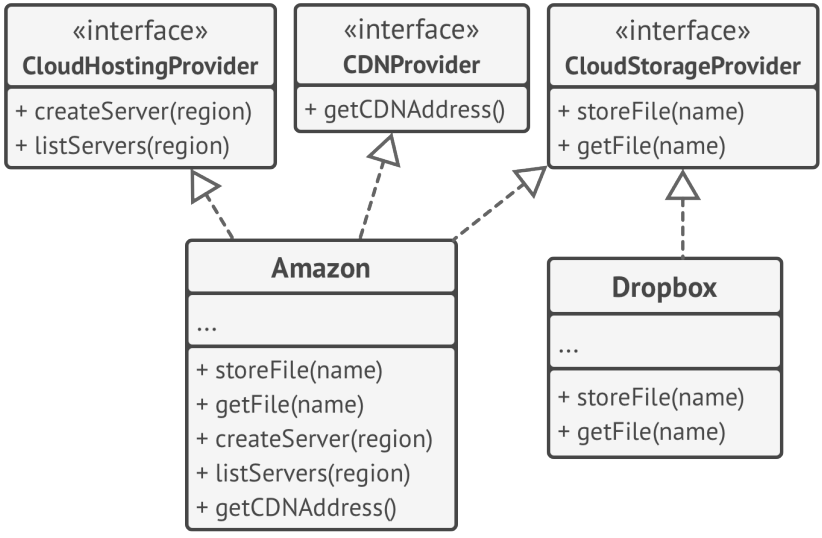
At the time you assumed that all cloud providers have the same broad spectrum of features as Amazon. But when it came to implementing support for another provider, it turned out that most of the interfaces of the library are too wide. Some methods describe features that other cloud providers just don't have.



BEFORE: not all clients can satisfy the requirements of the bloated interface.

While you can still implement these methods and put some stubs there, it wouldn't be a pretty solution. The better

approach is to break down the interface into parts. Classes that are able to implement the original interface can now just implement several refined interfaces. Other classes can implement only those interfaces which have methods that make sense for them.



AFTER: one bloated interface is broken down into a set of more granular interfaces.

As with the other principles, you can go too far with this one. Don't further divide an interface which is already quite specific. Remember that the more interfaces you create, the more complex your code becomes. Keep the balance.

Dependency Inversion Principle

> High-level classes shouldn't depend on low-level classes. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions.

Usually when designing software, you can make a distinction between two levels of classes.

- **Low-level classes** implement basic operations such as working with a disk, transferring data over a network, connecting to a database, etc.
- **High-level classes** contain complex business logic that directs low-level classes to do something.

Sometimes people design low-level classes first and only then start working on high-level ones. This is very common when you start developing a prototype on a new system, and you're not even sure what's possible at the higher level because low-level stuff isn't yet implemented or clear. With such an approach business logic classes tend to become dependent on primitive low-level classes.

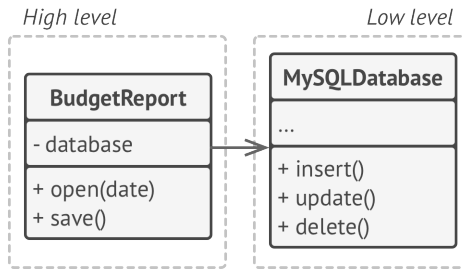
The dependency inversion principle suggests changing the direction of this dependency.

1. For starters, you need to describe interfaces for low-level operations that high-level classes rely on, preferably in business terms. For instance, business logic should call a method `openReport(file)` rather than a series of methods `openFile(x)`, `readBytes(n)`, `closeFile(x)`. These interfaces count as high-level ones.
2. Now you can make high-level classes dependent on those interfaces, instead of on concrete low-level classes. This dependency will be much softer than the original one.
3. Once low-level classes implement these interfaces, they become dependent on the business logic level, reversing the direction of the original dependency.

   The dependency inversion principle often goes along with the *open/closed principle*: you can extend low-level classes to use with different business logic classes without breaking existing classes.

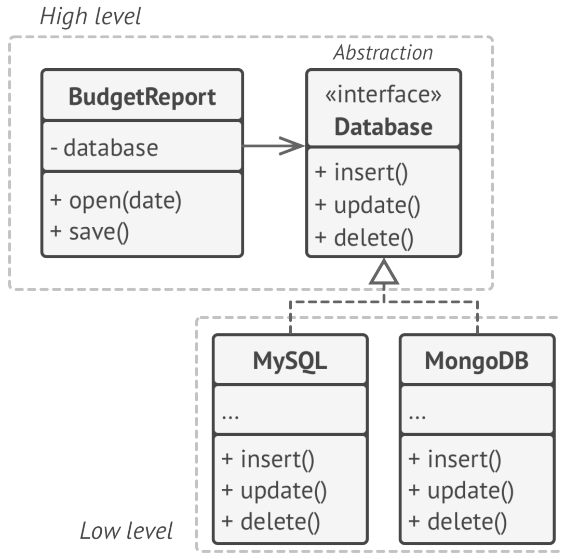
Example

In this example, the high-level budget reporting class uses a low-level database class for reading and persisting its data. This means that any change in the low-level class, such as when a new version of the database server gets released, may affect the high-level class, which isn't supposed to care about the data storage details.



BEFORE: a high-level class depends on a low-level class.

You can fix this problem by creating a high-level interface that describes read/write operations and making the reporting class use that interface instead of the low-level class. Then you can change or extend the original low-level class to implement the new read/write interface declared by the business logic.



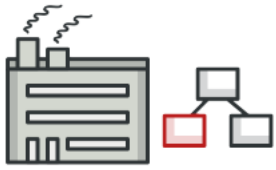
AFTER: low-level classes depend on a high-level abstraction.

As a result, the direction of the original dependency has been inverted: low-level classes are now dependent on high-level abstractions.

CATALOG OF DESIGN PATTERNS

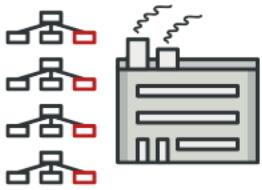
Creational Design Patterns

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

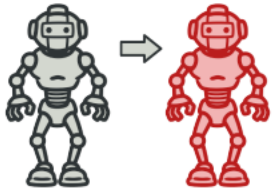


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

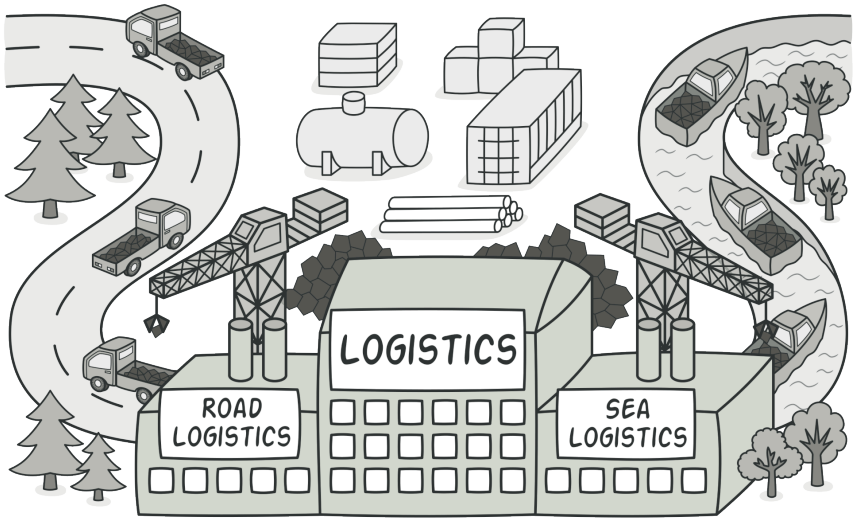


Prototype

Lets you copy existing objects without making your code dependent on their classes.

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.



FACTORY METHOD

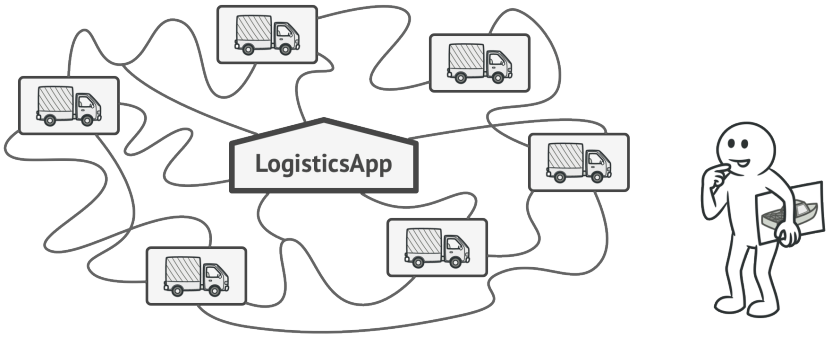
Also known as: Virtual Constructor

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

☹ Problem

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the `Truck` class.

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.



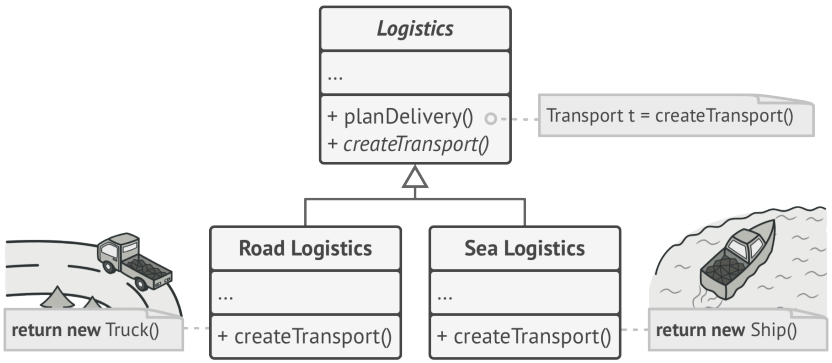
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Great news, right? But how about the code? At present, most of your code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

😊 Solution

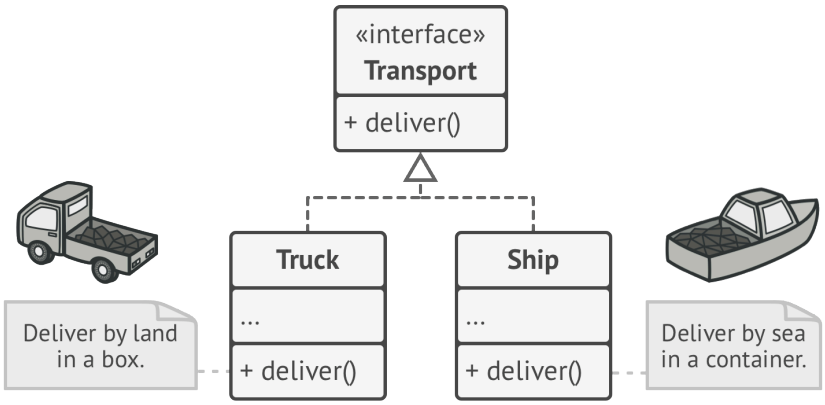
The Factory Method pattern suggests that you replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Don't worry: the objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as "products."



Subclasses can alter the class of objects being returned by the factory method.

At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

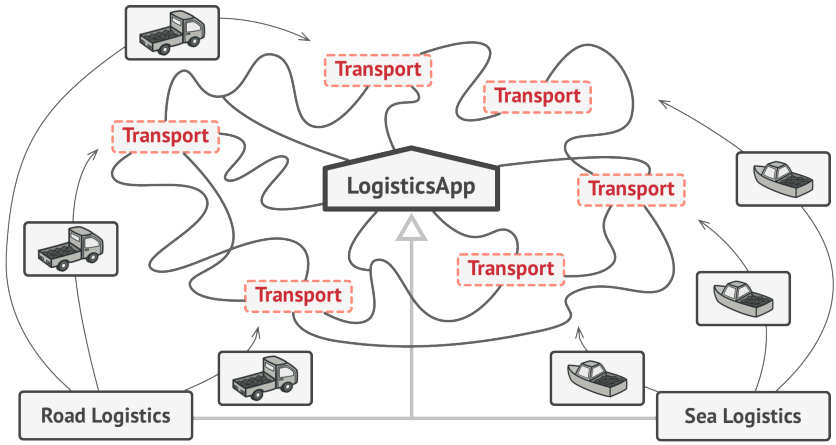
There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.



All products must follow the same interface.

For example, both `Truck` and `Ship` classes should implement the `Transport` interface, which declares a method called `deliver`. Each class implements this method differently: trucks deliver cargo by land, ships deliver cargo by sea. The factory method in the `RoadLogistics` class returns truck objects, whereas the factory method in the `SeaLogistics` class returns ships.

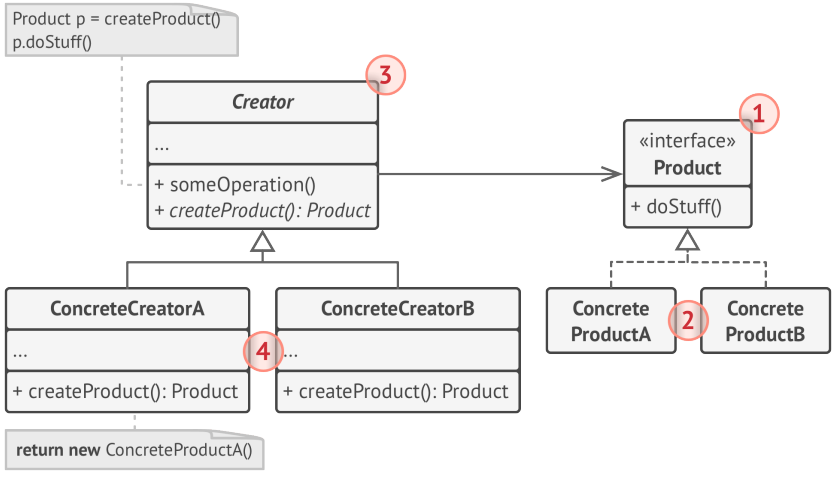
The code that uses the factory method (often called the *client* code) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as abstract `Transport`.



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

The client knows that all transport objects are supposed to have the `deliver` method, but exactly how it works isn't important to the client.

Structure



1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2. **Concrete Products** are different implementations of the product interface.

3. The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

   You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

   Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.

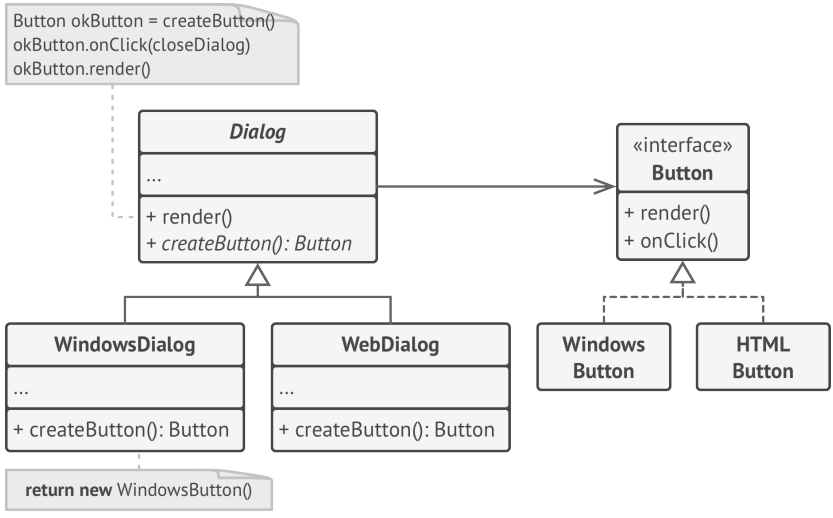
4. **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Pseudocode

This example illustrates how the **Factory Method** can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes.

The base dialog class uses different UI elements to render its window. Under various operating systems, these elements may look a little bit different, but they should still behave consistently. A button in Windows is still a button in Linux.



The cross-platform dialog example.

When the factory method comes into play, you don't need to rewrite the logic of the dialog for each operating system. If we declare a factory method that produces buttons inside the base dialog class, we can later create a dialog subclass that returns Windows-styled buttons from the factory method. The subclass then inherits most of the dialog's code from the base class, but, thanks to the factory method, can render Windows-looking buttons on the screen.

For this pattern to work, the base dialog class must work with abstract buttons: a base class or an interface that all concrete buttons follow. This way the dialog's code remains functional, whichever type of buttons it works with.

Of course, you can apply this approach to other UI elements as well. However, with each new factory method you add to the dialog, you get closer to the **Abstract Factory** pattern. Fear not, we'll talk about this pattern later.

```
// The creator class declares the factory method that must
// return an object of a product class. The creator's subclasses
// usually provide the implementation of this method.
class Dialog is
  // The creator may also provide some default implementation
  // of the factory method.
  abstract method createButton()

  // Note that, despite its name, the creator's primary
  // responsibility isn't creating products. It usually
```

```
    // contains some core business logic that relies on product
    // objects returned by the factory method. Subclasses can
    // indirectly change that business logic by overriding the
    // factory method and returning a different type of product
    // from it.
    method render() is
      // Call the factory method to create a product object.
      Button okButton = createButton()
      // Now use the product.
      okButton.onClick(closeDialog)
      okButton.render()


// Concrete creators override the factory method to change the
// resulting product's type.
class WindowsDialog extends Dialog is
  method createButton() is
    return new WindowsButton()

class WebDialog extends Dialog is
  method createButton() is
    return new HTMLButton()


// The product interface declares the operations that all
// concrete products must implement.
interface Button is
  method render()
  method onClick(f)

// Concrete products provide various implementations of the
// product interface.
```

```
class WindowsButton implements Button is
  method render(a, b) is
    // Render a button in Windows style.
  method onClick(f) is
    // Bind a native OS click event.

class HTMLButton implements Button is
  method render(a, b) is
    // Return an HTML representation of a button.
  method onClick(f) is
    // Bind a web browser click event.


class Application is
  field dialog: Dialog

  // The application picks a creator's type depending on the
  // current configuration or environment settings.
  method initialize() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      dialog = new WindowsDialog()
    else if (config.OS == "Web") then
      dialog = new WebDialog()
    else
      throw new Exception("Error! Unknown operating system.")

  // The client code works with an instance of a concrete
  // creator, albeit through its base interface. As long as
  // the client keeps working with the creator via the base
  // interface, you can pass it any creator's subclass.
```

```
75    method main() is
76      this.initialize()
77      dialog.render()
```

Applicability

Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.

The Factory Method separates product construction code from the code that actually uses the product. Therefore it's easier to extend the product construction code independently from the rest of the code.

For example, to add a new product type to the app, you'll only need to create a new creator subclass and override the factory method in it.

Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.

Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component?

The solution is to reduce the code that constructs components across the framework into a single factory method and let anyone override this method in addition to extending the component itself.

Let's see how that would work. Imagine that you write an app using an open source UI framework. Your app should have round buttons, but the framework only provides square ones. You extend the standard `Button` class with a glorious `RoundButton` subclass. But now you need to tell the main `UIFramework` class to use the new button subclass instead of a default one.

To achieve this, you create a subclass `UIWithRoundButtons` from a base framework class and override its `createButton` method. While this method returns `Button` objects in the base class, you make your subclass return `RoundButton` objects. Now use the `UIWithRoundButtons` class instead of `UIFramework`. And that's about it!

Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

You often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.

Let's think about what has to be done to reuse an existing object:

1. First, you need to create some storage to keep track of all of the created objects.
2. When someone requests an object, the program should look for a free object inside that pool.
3. … and then return it to the client code.
4. If there are no free objects, the program should create a new one (and add it to the pool).

That's a lot of code! And it must all be put into a single place so that you don't pollute the program with duplicate code.

Probably the most obvious and convenient place where this code could be placed is the constructor of the class whose objects we're trying to reuse. However, a constructor must always return **new objects** by definition. It can't return existing instances.

Therefore, you need to have a regular method capable of creating new objects as well as reusing existing ones. That sounds very much like a factory method.

How to Implement

1. Make all products follow the same interface. This interface should declare methods that make sense in every product.

2. Add an empty factory method inside the creator class. The return type of the method should match the common product interface.

3. In the creator's code find all references to product constructors. One by one, replace them with calls to the factory method, while extracting the product creation code into the factory method.

   You might need to add a temporary parameter to the factory method to control the type of returned product.

   At this point, the code of the factory method may look pretty ugly. It may have a large `switch` operator that picks which product class to instantiate. But don't worry, we'll fix it soon enough.

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

5. If there are too many product types and it doesn't make sense to create subclasses for all of them, you can reuse the control parameter from the base class in subclasses.

   For instance, imagine that you have the following hierarchy of classes: the base `Mail` class with a couple of subclasses: `AirMail` and `GroundMail`; the `Transport` classes are `Plane`,

`Truck` and `Train`. While the `AirMail` class only uses `Plane` objects, `GroundMail` may work with both `Truck` and `Train` objects. You can create a new subclass (say `TrainMail`) to handle both cases, but there's another option. The client code can pass an argument to the factory method of the `GroundMail` class to control which product it wants to receive.

6. If, after all of the extractions, the base factory method has become empty, you can make it abstract. If there's something left, you can make it a default behavior of the method.

⚖ Pros and Cons

✓ You avoid tight coupling between the creator and the concrete products.

✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.

✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

✕ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

⇄ Relations with Other Patterns

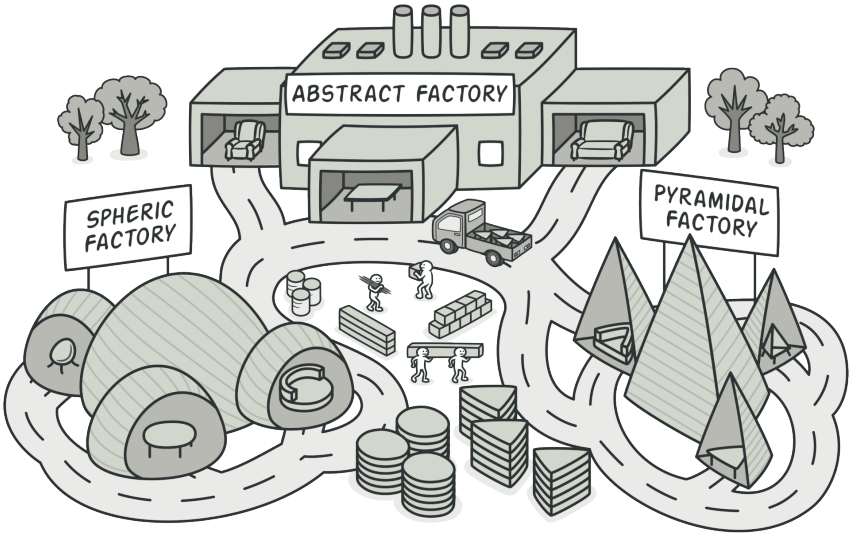
- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.

- You can use **Factory Method** along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections.

- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.

- **Factory Method** is a specialization of **Template Method**. At the same time, a *Factory Method* may serve as a step in a large *Template Method*.



ABSTRACT FACTORY

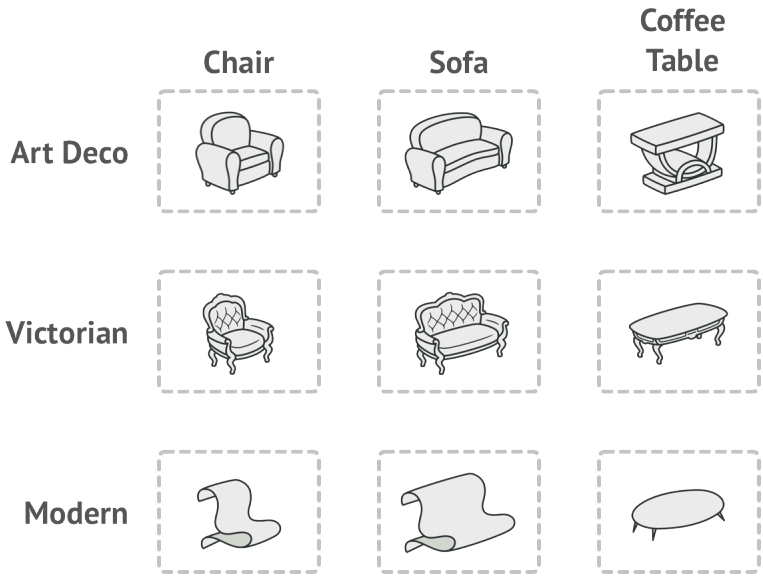
Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

☹ Problem

Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say: `Chair` + `Sofa` + `CoffeeTable`.

2. Several variants of this family. For example, products `Chair` + `Sofa` + `CoffeeTable` are available in these variants: `Modern`, `Victorian`, `ArtDeco`.



Product families and their variants.

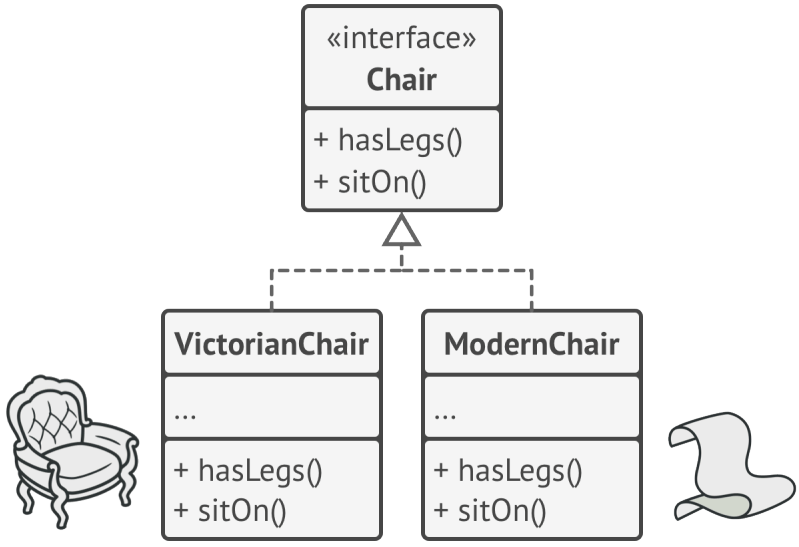
You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

A Modern-style sofa doesn't match Victorian-style chairs.

Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.

😊 Solution

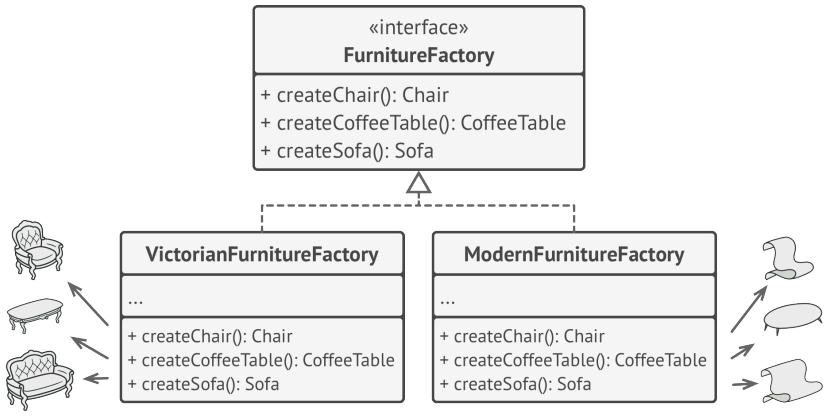
The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the `Chair` interface; all coffee table variants can implement the `CoffeeTable` interface, and so on.



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

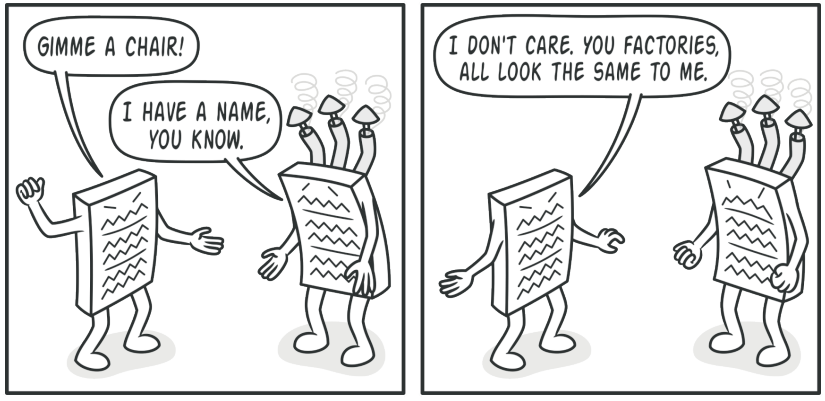
The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (for example, `createChair`, `createSofa` and `createCoffeeTable`). These methods must return **abstract** product types represented by the interfaces we extracted previously: `Chair`, `Sofa`, `CoffeeTable` and so on.

Now, how about the product variants? For each variant of a product family, we create a separate factory class based on the `AbstractFactory` interface. A factory is a class that returns products of a particular kind. For example, the `ModernFactory` can only create `ModernChair`, `ModernSofa` and `ModernCoffeeTable` objects.



Each concrete factory corresponds to a specific product variant.

The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

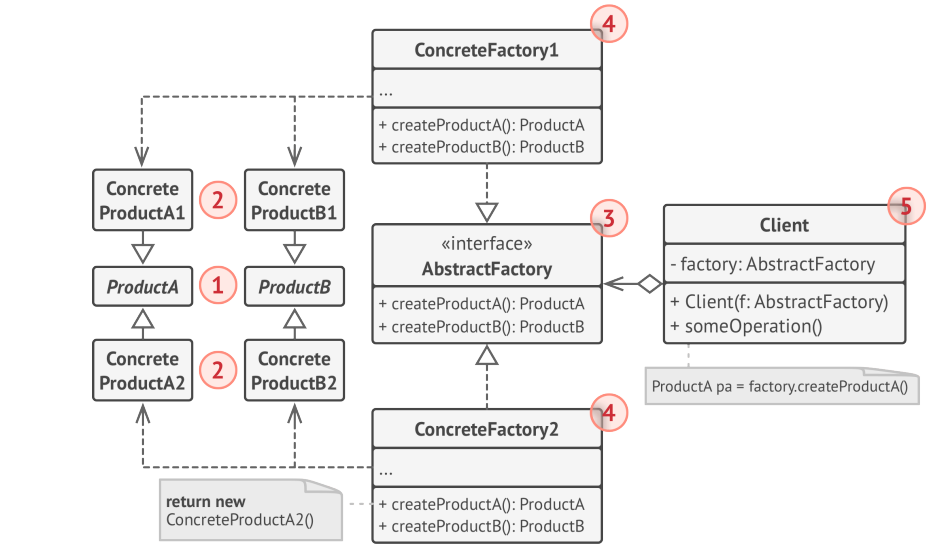


The client shouldn't care about the concrete class of the factory it works with.

Say the client wants a factory to produce a chair. The client doesn't have to be aware of the factory's class, nor does it matter what kind of chair it gets. Whether it's a Modern model or a Victorian-style chair, the client must treat all chairs in the same manner, using the abstract `Chair` interface. With this approach, the only thing that the client knows about the chair is that it implements the `sitOn` method in some way. Also, whichever variant of the chair is returned, it'll always match the type of sofa or coffee table produced by the same factory object.

One more thing left to clarify: if the client is only exposed to the abstract interfaces, what creates the actual factory objects? Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.

Structure



1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.

2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).

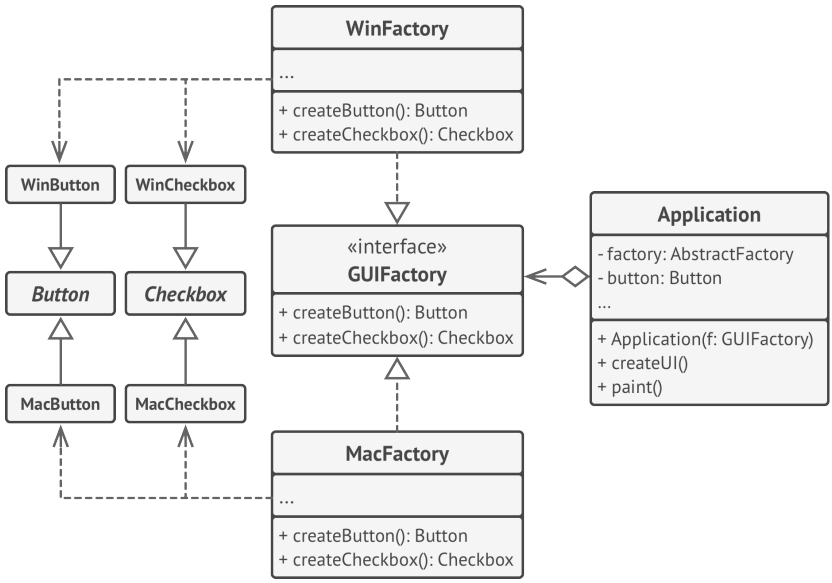
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.

4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

5. Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding *abstract* products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

Pseudocode

This example illustrates how the **Abstract Factory** pattern can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes, while keeping all created elements consistent with a selected operating system.



The cross-platform UI classes example.

The same UI elements in a cross-platform application are expected to behave similarly, but look a little bit different under different operating systems. Moreover, it's your job to make sure that the UI elements match the style of the current operating system. You wouldn't want your program to render macOS controls when it's executed in Windows.

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

It works like this: when an application launches, it checks the type of the current operating system. The app uses this infor-

mation to create a factory object from a class that matches the operating system. The rest of the code uses this factory to create UI elements. This prevents the wrong elements from being created.

With this approach, the client code doesn't depend on concrete classes of factories and UI elements as long as it works with these objects via their abstract interfaces. This also lets the client code support other factories or UI elements that you might add in the future.

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

```
// The abstract factory interface declares a set of methods that
// return different abstract products. These products are called
// a family and are related by a high-level theme or concept.
// Products of one family are usually able to collaborate among
// themselves. A family of products may have several variants,
// but the products of one variant are incompatible with the
// products of another variant.
interface GUIFactory is
  method createButton():Button
  method createCheckbox():Checkbox


// Concrete factories produce a family of products that belong
// to a single variant. The factory guarantees that the
// resulting products are compatible. Signatures of the concrete
// factory's methods return an abstract product, while inside
// the method a concrete product is instantiated.
class WinFactory implements GUIFactory is
  method createButton():Button is
    return new WinButton()
  method createCheckbox():Checkbox is
    return new WinCheckbox()

// Each concrete factory has a corresponding product variant.
class MacFactory implements GUIFactory is
  method createButton():Button is
    return new MacButton()
  method createCheckbox():Checkbox is
    return new MacCheckbox()



```

```
// Each distinct product of a product family should have a base
// interface. All variants of the product must implement this
// interface.
interface Button is
  method paint()

// Concrete products are created by corresponding concrete
// factories.
class WinButton implements Button is
  method paint() is
    // Render a button in Windows style.

class MacButton implements Button is
  method paint() is
    // Render a button in macOS style.

// Here's the base interface of another product. All products
// can interact with each other, but proper interaction is
// possible only between products of the same concrete variant.
interface Checkbox is
  method paint()

class WinCheckbox implements Checkbox is
  method paint() is
    // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
  method paint() is
    // Render a checkbox in macOS style.
```

```
// The client code works with factories and products only
// through abstract types: GUIFactory, Button and Checkbox. This
// lets you pass any factory or product subclass to the client
// code without breaking it.
class Application is
  private field button: Button
  constructor Application(factory: GUIFactory) is
    this.factory = factory
  method createUI() is
    this.button = factory.createButton()
  method paint() is
    button.paint()


// The application picks the factory type depending on the
// current configuration or environment settings and creates it
// at runtime (usually at the initialization stage).
class ApplicationConfigurator is
  method main() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      factory = new WinFactory()
    else if (config.OS == "Mac") then
      factory = new MacFactory()
    else
      throw new Exception("Error! Unknown operating system.")

    Application app = new Application(factory)
```

Applicability

Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

The Abstract Factory provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

- Consider implementing the Abstract Factory when you have a class with a set of **Factory Methods** that blur its primary responsibility.

- In a well-designed program *each class is responsible only for one thing*. When a class deals with multiple product types, it may be worth extracting its factory methods into a standalone factory class or a full-blown Abstract Factory implementation.

How to Implement

1. Map out a matrix of distinct product types versus variants of these products.

2. Declare abstract product interfaces for all product types. Then make all concrete product classes implement these interfaces.

3. Declare the abstract factory interface with a set of creation methods for all abstract products.

4. Implement a set of concrete factory classes, one for each product variant.

5. Create factory initialization code somewhere in the app. It should instantiate one of the concrete factory classes, depending on the application configuration or the current environment. Pass this factory object to all classes that construct products.

6. Scan through the code and find all direct calls to product constructors. Replace them with calls to the appropriate creation method on the factory object.

Pros and Cons

- ✓ You can be sure that the products you're getting from a factory are compatible with each other.
- ✓ You avoid tight coupling between concrete products and client code.
- ✓ *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.

✓ *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

✕ The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

⇄ Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

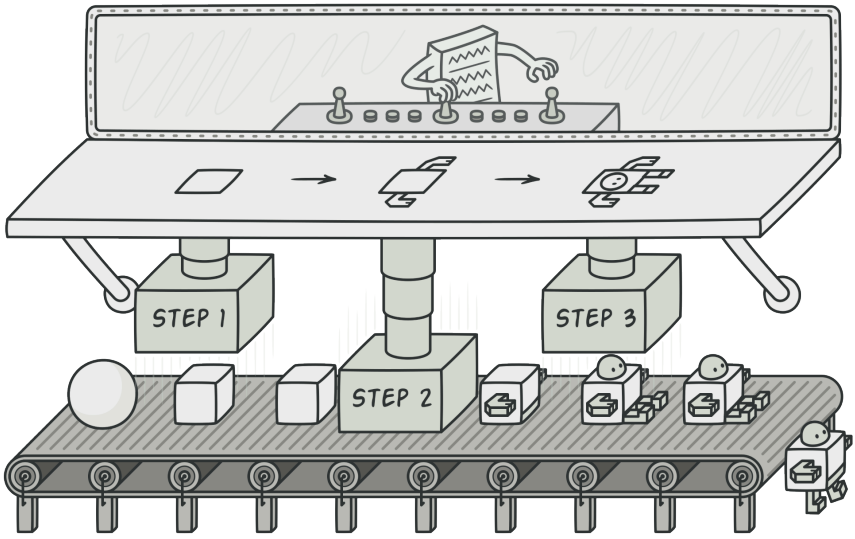
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.

- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.

- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.

- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by *Bridge* can only work with specific implementations. In this case, *Abstract Factory* can encapsulate these relations and hide the complexity from the client code.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

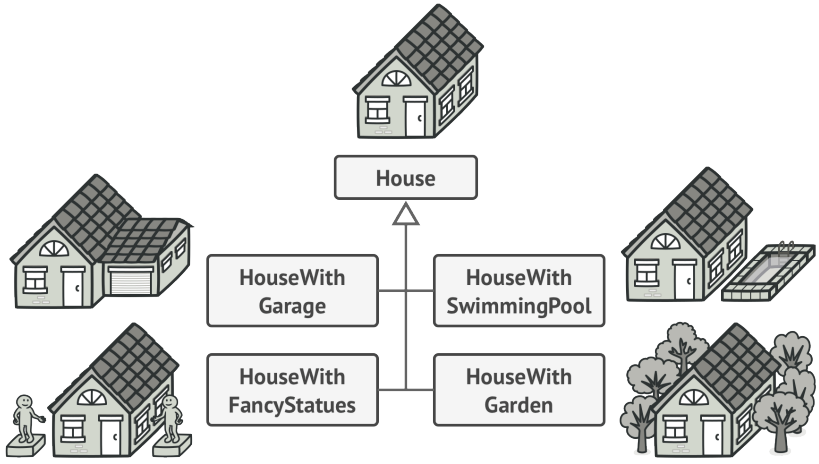


BUILDER

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

☹ Problem

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

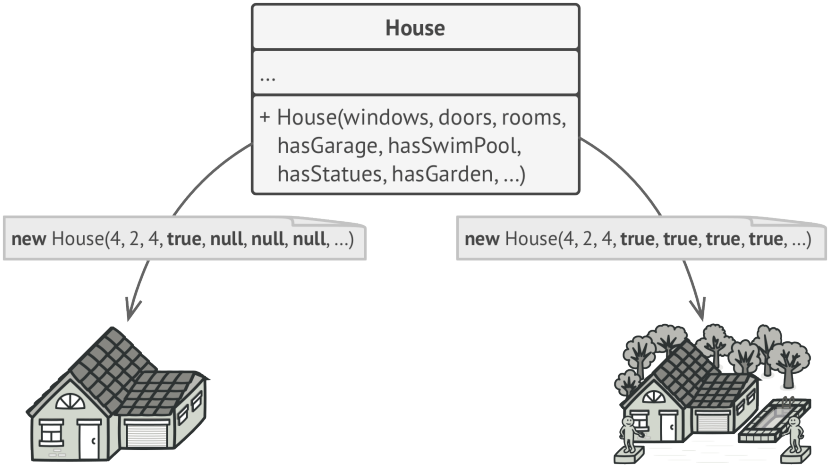


You might make the program too complex by creating a subclass for every possible configuration of an object.

For example, let's think about how to create a `House` object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

The simplest solution is to extend the base `House` class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base `House` class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

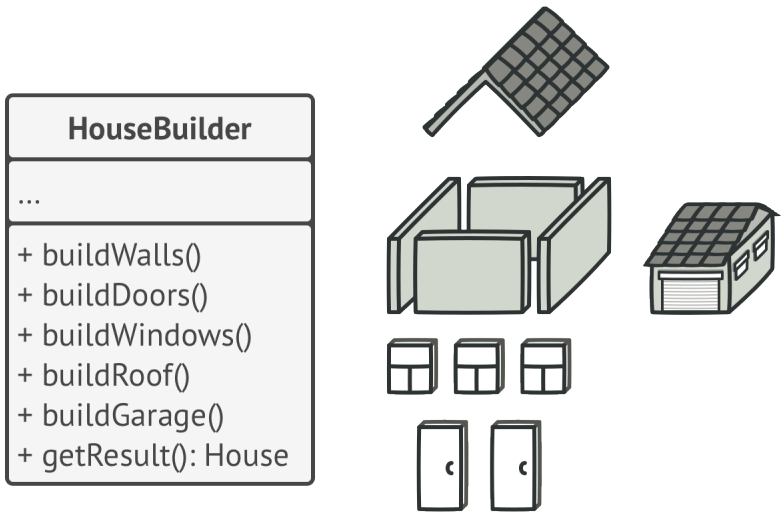


The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

In most cases most of the parameters will be unused, making **the constructor calls pretty ugly**. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.

😊 Solution

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

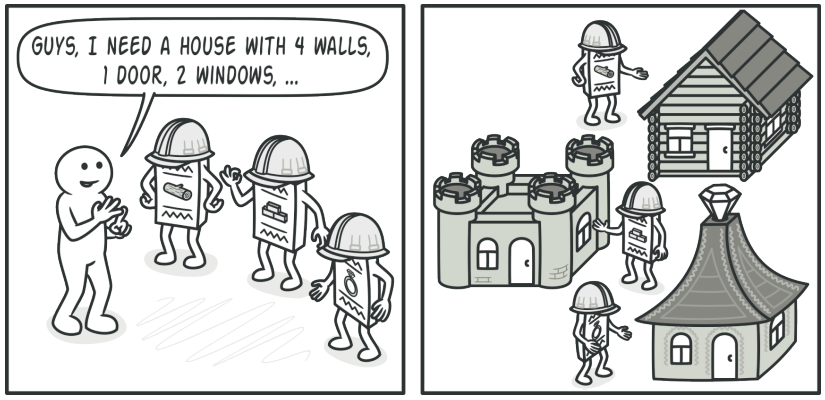


The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

The pattern organizes object construction into a set of steps (`buildWalls`, `buildDoor`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.

In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.



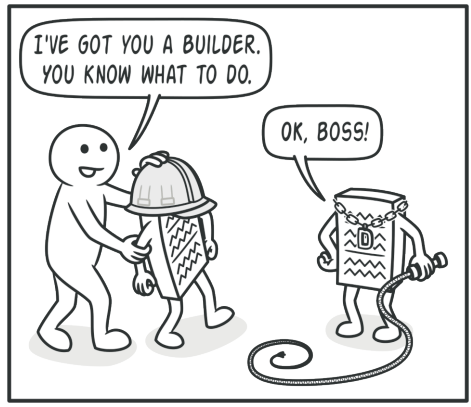
Different builders execute the same task in various ways.

For example, imagine a builder that builds everything from wood and glass, a second one that builds everything with stone and iron and a third one that uses gold and diamonds. By calling the same set of steps, you get a regular house from the first builder, a small castle from the second and a palace from the third. However, this would only work if the client code that

calls the building steps is able to interact with builders using a common interface.

Director

You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called *director*. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

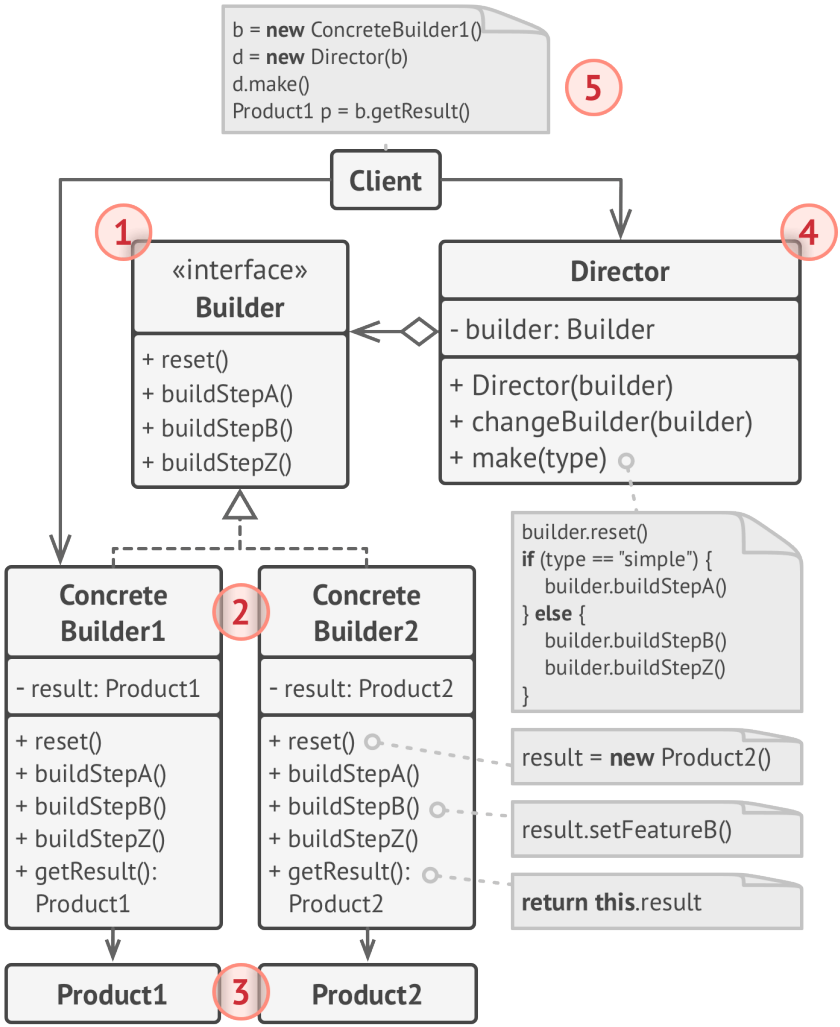


The director knows which building steps to execute to get a working product.

Having a director class in your program isn't strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

Structure



1. The **Builder** interface declares product construction steps that are common to all types of builders.

2. **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

3. **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.

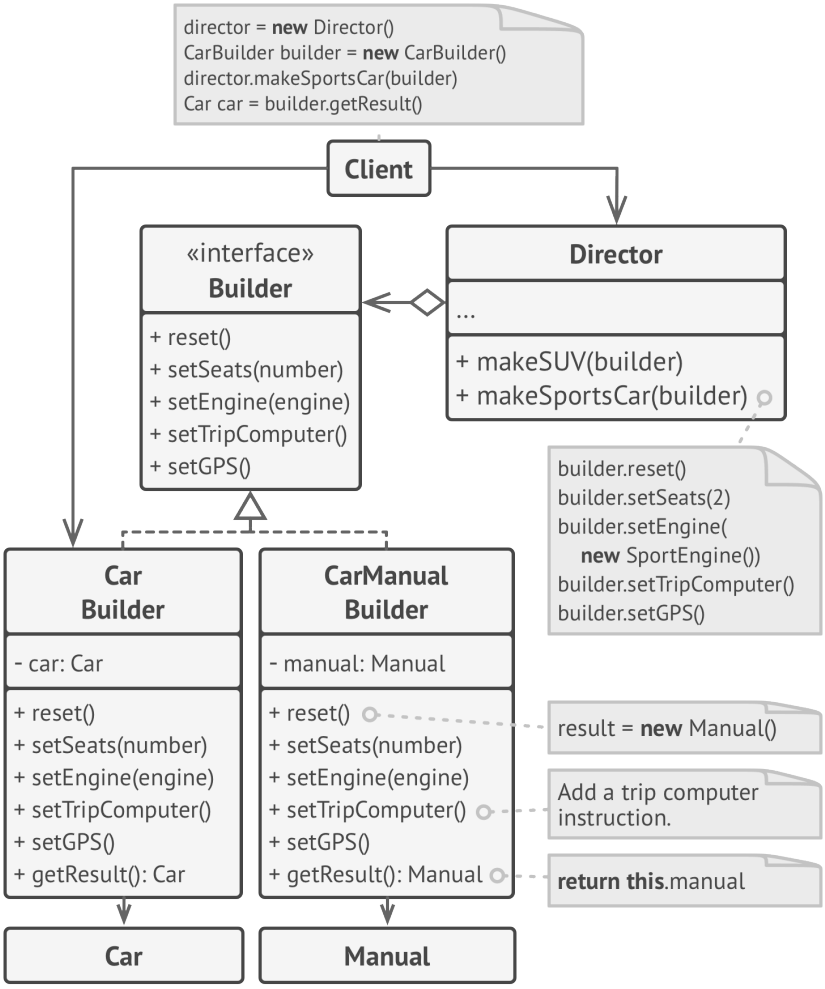
4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

5. The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

Pseudocode

This example of the **Builder** pattern illustrates how you can reuse the same object construction code when building differ-

ent types of products, such as cars, and create the corresponding manuals for them.



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

A car is a complex object that can be constructed in a hundred different ways. Instead of bloating the `Car` class with a huge

constructor, we extracted the car assembly code into a separate car builder class. This class has a set of methods for configuring various parts of a car.

If the client code needs to assemble a special, fine-tuned model of a car, it can work with the builder directly. On the other hand, the client can delegate the assembly to the director class, which knows how to use a builder to construct several of the most popular models of cars.

You might be shocked, but every car needs a manual (seriously, who reads them?). The manual describes every feature of the car, so the details in the manuals vary across the different models. That's why it makes sense to reuse an existing construction process for both real cars and their respective manuals. Of course, building a manual isn't the same as building a car, and that's why we must provide another builder class that specializes in composing manuals. This class implements the same building methods as its car-building sibling, but instead of crafting car parts, it describes them. By passing these builders to the same director object, we can construct either a car or a manual.

The final part is fetching the resulting object. A metal car and a paper manual, although related, are still very different things. We can't place a method for fetching results in the director without coupling the director to concrete product classes. Hence, we obtain the result of the construction from the builder which performed the job.

```
// Using the Builder pattern makes sense only when your products
// are quite complex and require extensive configuration. The
// following two products are related, although they don't have
// a common interface.
class Car is
  // A car can have a GPS, trip computer and some number of
  // seats. Different models of cars (sports car, SUV,
  // cabriolet) might have different features installed or
  // enabled.

class Manual is
  // Each car should have a user manual that corresponds to
  // the car's configuration and describes all its features.


// The builder interface specifies methods for creating the
// different parts of the product objects.
interface Builder is
  method reset()
  method setSeats(...)
  method setEngine(...)
  method setTripComputer(...)
  method setGPS(...)

// The concrete builder classes follow the builder interface and
// provide specific implementations of the building steps. Your
// program may have several variations of builders, each
// implemented differently.
class CarBuilder implements Builder is
  private field car:Car


```

```
33    // A fresh builder instance should contain a blank product
34    // object which it uses in further assembly.
35    constructor CarBuilder() is
36      this.reset()
37
38    // The reset method clears the object being built.
39    method reset() is
40      this.car = new Car()
41
42    // All production steps work with the same product instance.
43    method setSeats(...) is
44      // Set the number of seats in the car.
45
46    method setEngine(...) is
47      // Install a given engine.
48
49    method setTripComputer(...) is
50      // Install a trip computer.
51
52    method setGPS(...) is
53      // Install a global positioning system.
54
55    // Concrete builders are supposed to provide their own
56    // methods for retrieving results. That's because various
57    // types of builders may create entirely different products
58    // that don't all follow the same interface. Therefore such
59    // methods can't be declared in the builder interface (at
60    // least not in a statically-typed programming language).
61    //
62    // Usually, after returning the end result to the client, a
63    // builder instance is expected to be ready to start
64    // producing another product. That's why it's a usual
```

```
    // practice to call the reset method at the end of the
    // `getProduct` method body. However, this behavior isn't
    // mandatory, and you can make your builder wait for an
    // explicit reset call from the client code before disposing
    // of the previous result.
    method getProduct():Car is
      product = this.car
      this.reset()
      return product

// Unlike other creational patterns, builder lets you construct
// unrelated products that don't follow a common interface.
class CarManualBuilder implements Builder is
    private field manual:Manual

    constructor CarManualBuilder() is
      this.reset()

    method reset() is
      this.manual = new Manual()

    method setSeats(...) is
      // Document car seat features.

    method setEngine(...) is
      // Add engine instructions.

    method setTripComputer(...) is
      // Add trip computer instructions.

    method setGPS(...) is
      // Add GPS instructions.
```

```
    method getProduct():Manual is
      // Return the manual and reset the builder.


// The director is only responsible for executing the building
// steps in a particular sequence. It's helpful when producing
// products according to a specific order or configuration.
// Strictly speaking, the director class is optional, since the
// client can control builders directly.
class Director is
  private field builder:Builder

  // The director works with any builder instance that the
  // client code passes to it. This way, the client code may
  // alter the final type of the newly assembled product.
  method setBuilder(builder:Builder)
    this.builder = builder

  // The director can construct several product variations
  // using the same building steps.
  method constructSportsCar(builder: Builder) is
    builder.reset()
    builder.setSeats(2)
    builder.setEngine(new SportEngine())
    builder.setTripComputer(true)
    builder.setGPS(true)

  method constructSUV(builder: Builder) is
    // ...
```

```
// The client code creates a builder object, passes it to the
// director and then initiates the construction process. The end
// result is retrieved from the builder object.
class Application is

  method makeCar() is
    director = new Director()

    CarBuilder builder = new CarBuilder()
    director.constructSportsCar(builder)
    Car car = builder.getProduct()

    CarManualBuilder builder = new CarManualBuilder()
    director.constructSportsCar(builder)

    // The final product is often retrieved from a builder
    // object since the director isn't aware of and not
    // dependent on concrete builders and products.
    Manual manual = builder.getProduct()
```

Applicability

Use the Builder pattern to get rid of a “telescopic constructor”.

Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  Pizza(int size, boolean cheese, boolean pepperoni) { ... }
  // ...
```

Creating such a monster is only possible in languages that support method overloading, such as C# or Java.

The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don't have to cram dozens of parameters into your constructors anymore.

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.

Use the Builder to construct Composite trees or other complex objects.

The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

A builder doesn't expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

How to Implement

1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.

2. Declare these steps in the base builder interface.

3. Create a concrete builder class for each of the product representations and implement their construction steps.

   Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may construct products that don't have a common interface. Therefore, you don't know what would be the return type for such a method. However, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.

4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.

5. The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director. Usually, the client does this only once, via parameters of the director's constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed directly to the construction method of the director.

6. The construction result can be obtained directly from the director only if all products follow the same interface. Otherwise, the client should fetch the result from the builder.

Pros and Cons

✓ You can construct objects step-by-step, defer construction steps or run steps recursively.

✓ You can reuse the same construction code when building various representations of products.

✓ *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

✕ The overall complexity of the code increases since the pattern requires creating multiple new classes.

⇄ Relations with Other Patterns

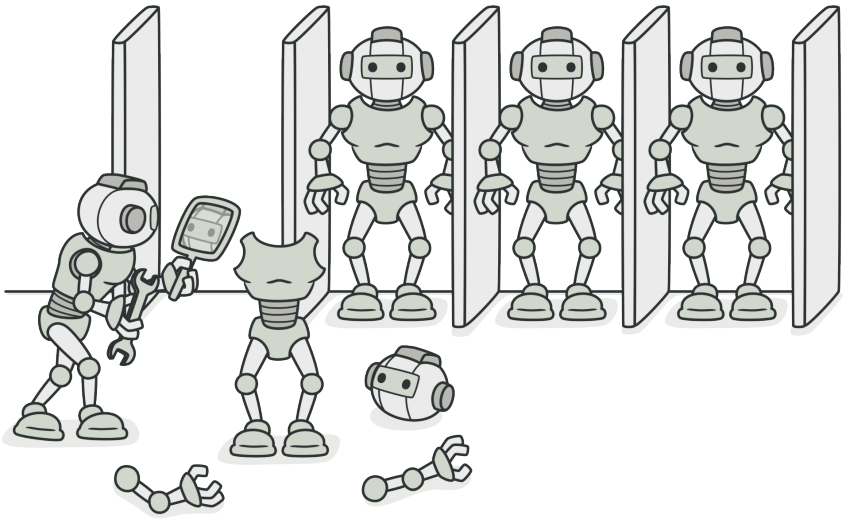
- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.

- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.

- You can combine **Builder** with **Bridge**: the *director* class plays the role of the abstraction, while different *builders* act as *implementations*.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.



PROTOTYPE

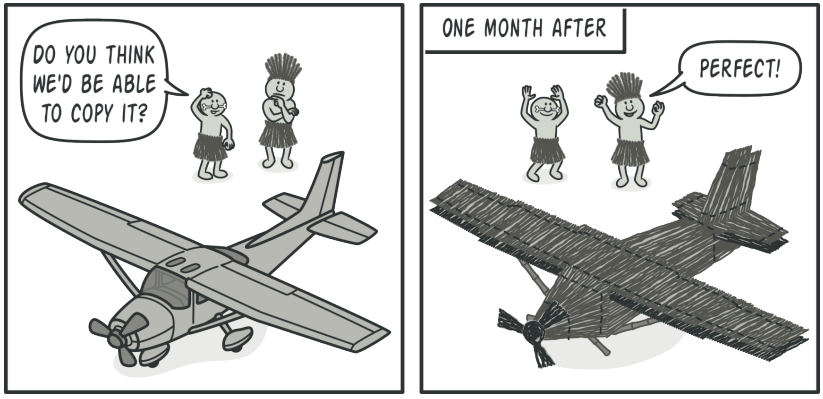
Also known as: Clone

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

☹ Problem

Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.



*Copying an object "from the outside" **isn't** always possible.*

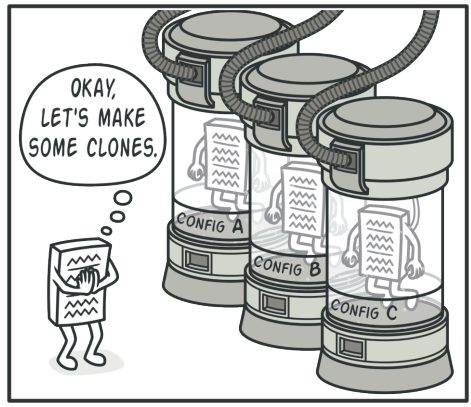
There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete

class, when, for example, a parameter in a method accepts any objects that follow some interface.

😊 Solution

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single `clone` method.

The implementation of the `clone` method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.



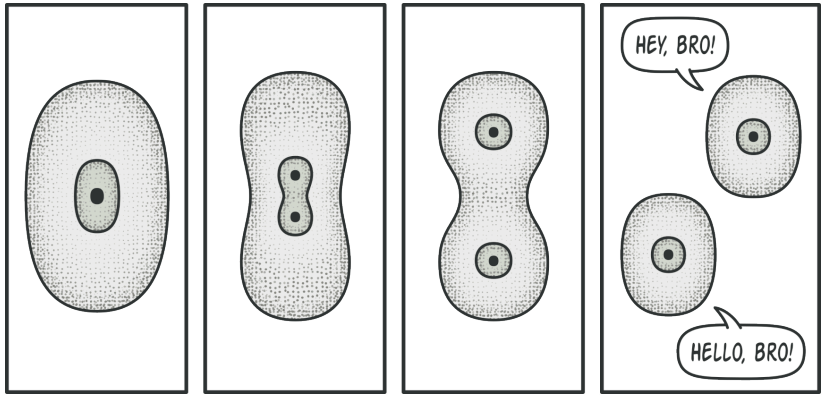
Pre-built prototypes can be an alternative to subclassing.

An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.

Real-World Analogy

In real life, prototypes are used for performing various tests before starting mass production of a product. However, in this case, prototypes don't participate in any actual production, playing a passive role instead.

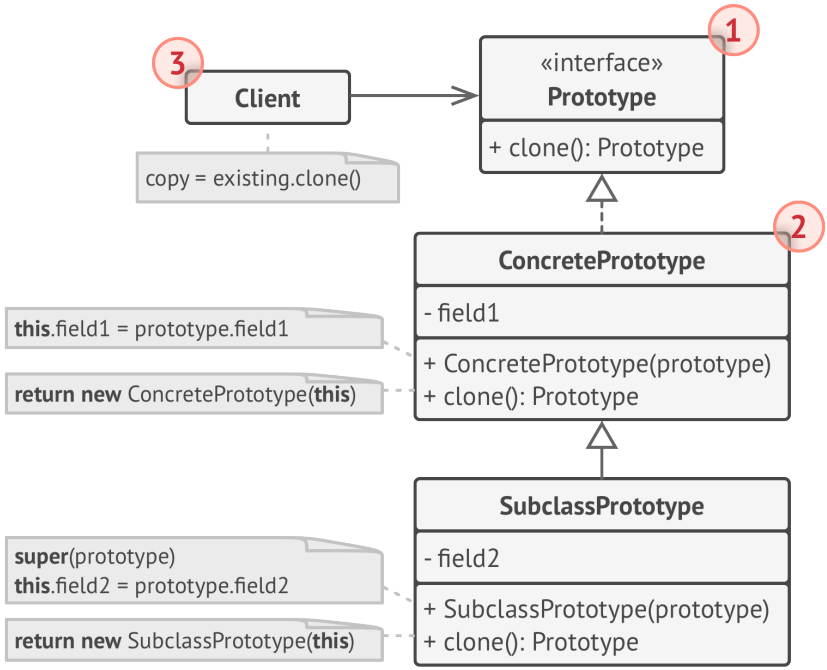


The division of a cell.

Since industrial prototypes don't really copy themselves, a much closer analogy to the pattern is the process of mitotic cell division (biology, remember?). After mitotic division, a pair of identical cells is formed. The original cell acts as a prototype and takes an active role in creating the copy.

Structure

Basic implementation

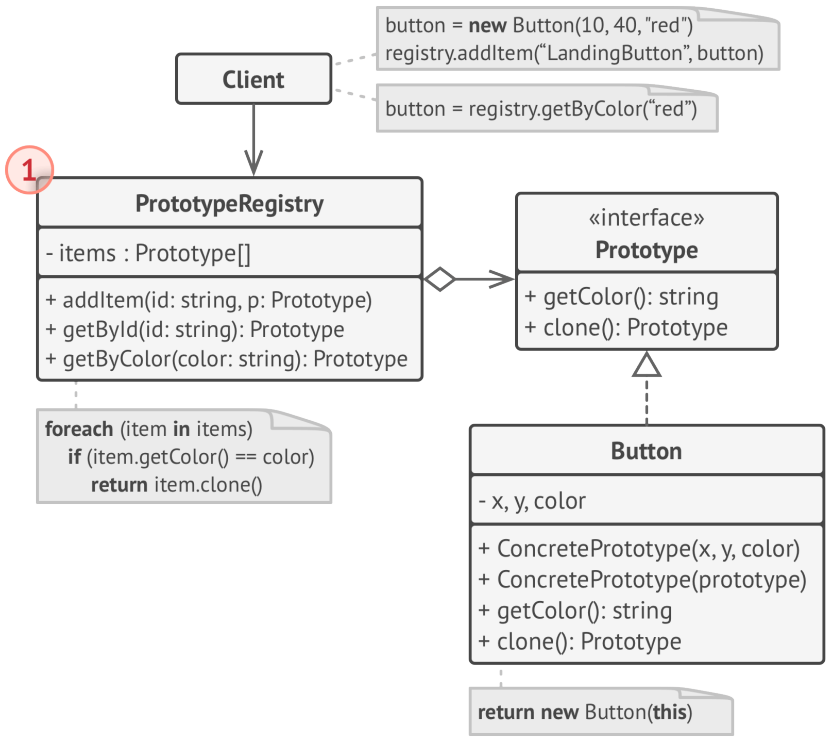


1. The **Prototype** interface declares the cloning methods. In most cases, it's a single `clone` method.

2. The **Concrete Prototype** class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

3. The **Client** can produce a copy of any object that follows the prototype interface.

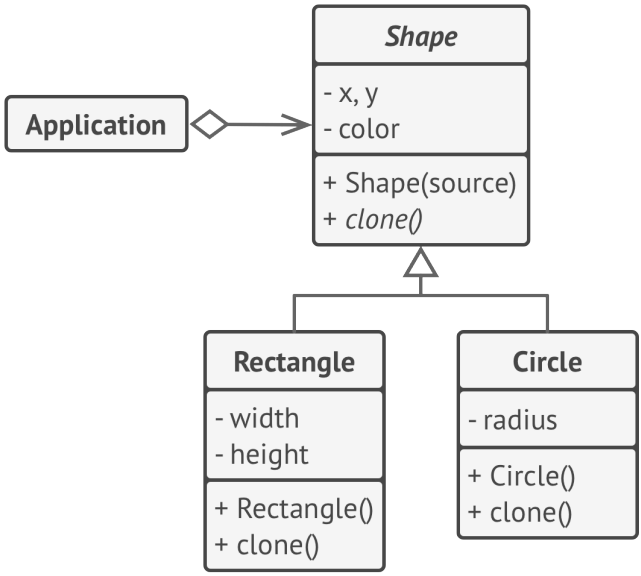
Prototype registry implementation



1. The **Prototype Registry** provides an easy way to access frequently-used prototypes. It stores a set of pre-built objects that are ready to be copied. The simplest prototype registry is a `name → prototype` hash map. However, if you need better search criteria than a simple name, you can build a much more robust version of the registry.

Pseudocode

In this example, the **Prototype** pattern lets you produce exact copies of geometric objects, without coupling the code to their classes.



Cloning a set of objects that belong to a class hierarchy.

All shape classes follow the same interface, which provides a cloning method. A subclass may call the parent's cloning

method before copying its own field values to the resulting object.

```
// Base prototype.
abstract class Shape is
  field X: int
  field Y: int
  field color: string

  // A regular constructor.
  constructor Shape() is
    // ...

  // The prototype constructor. A fresh object is initialized
  // with values from the existing object.
  constructor Shape(source: Shape) is
    this()
    this.X = source.X
    this.Y = source.Y
    this.color = source.color

  // The clone operation returns one of the Shape subclasses.
  abstract method clone():Shape


// Concrete prototype. The cloning method creates a new object
// and passes it to the constructor. Until the constructor is
// finished, it has a reference to a fresh clone. Therefore,
// nobody has access to a partly-built clone. This keeps the
// cloning result consistent.
class Rectangle extends Shape is
```

```
  field width: int
  field height: int

  constructor Rectangle(source: Rectangle) is
    // A parent constructor call is needed to copy private
    // fields defined in the parent class.
    super(source)
    this.width = source.width
    this.height = source.height

  method clone():Shape is
    return new Rectangle(this)


class Circle extends Shape is
  field radius: int

  constructor Circle(source: Circle) is
    super(source)
    this.radius = source.radius

  method clone():Shape is
    return new Circle(this)


// Somewhere in the client code.
class Application is
  field shapes: array of Shape

  constructor Application() is
    Circle circle = new Circle()
    circle.X = 10
```

```
    circle.Y = 10
    circle.radius = 20
    shapes.add(circle)

    Circle anotherCircle = circle.clone()
    shapes.add(anotherCircle)
    // The `anotherCircle` variable contains an exact copy
    // of the `circle` object.

    Rectangle rectangle = new Rectangle()
    rectangle.width = 10
    rectangle.height = 20
    shapes.add(rectangle)

  method businessLogic() is
    // Prototype rocks because it lets you produce a copy of
    // an object without knowing anything about its type.
    Array shapesCopy = new Array of Shapes.

    // For instance, we don't know the exact elements in the
    // shapes array. All we know is that they are all
    // shapes. But thanks to polymorphism, when we call the
    // `clone` method on a shape the program checks its real
    // class and runs the appropriate clone method defined
    // in that class. That's why we get proper clones
    // instead of a set of simple Shape objects.
    foreach (s in shapes) do
      shapesCopy.add(s.clone())

    // The `shapesCopy` array contains exact copies of the
    // `shape` array's children.
```

Applicability

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

This happens a lot when your code works with objects passed to you from 3rd-party code via some interface. The concrete classes of these objects are unknown, and you couldn't depend on them even if you wanted to.

The Prototype pattern provides the client code with a general interface for working with all objects that support cloning. This interface makes the client code independent from the concrete classes of objects that it clones.

Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.

The Prototype pattern lets you use a set of pre-built objects, configured in various ways, as prototypes.

Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

How to Implement

1. Create the prototype interface and declare the `clone` method in it. Or just add the method to all classes of an existing class hierarchy, if you have one.

2. A prototype class must define the alternative constructor that accepts an object of that class as an argument. The constructor must copy the values of all fields defined in the class from the passed object into the newly created instance. If you're changing a subclass, you must call the parent constructor to let the superclass handle the cloning of its private fields.

   If your programming language doesn't support method overloading, you may define a special method for copying the object data. The constructor is a more convenient place to do this because it delivers the resulting object right after you call the `new` operator.

3. The cloning method usually consists of just one line: running a `new` operator with the prototypical version of the constructor. Note, that every class must explicitly override the cloning method and use its own class name along with the `new` operator. Otherwise, the cloning method may produce an object of a parent class.

4. Optionally, create a centralized prototype registry to store a catalog of frequently used prototypes.

You can implement the registry as a new factory class or put it in the base prototype class with a static method for fetching the prototype. This method should search for a prototype based on search criteria that the client code passes to the method. The criteria might either be a simple string tag or a complex set of search parameters. After the appropriate prototype is found, the registry should clone it and return the copy to the client.

Finally, replace the direct calls to the subclasses' constructors with calls to the factory method of the prototype registry.

⚖ Pros and Cons

- ✓ You can clone objects without coupling to their concrete classes.
- ✓ You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- ✓ You can produce complex objects more conveniently.
- ✓ You get an alternative to inheritance when dealing with configuration presets for complex objects.

- ✕ Cloning complex objects that have circular references might be very tricky.

⇄ Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.

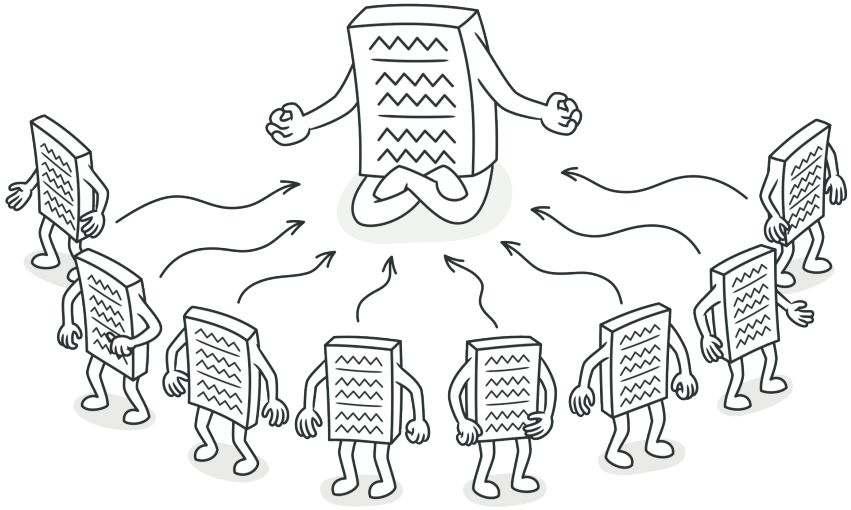
- **Prototype** can help when you need to save copies of **Commands** into history.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.

- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.

- Sometimes **Prototype** can be a simpler alternative to **Memento**. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.



SINGLETON

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

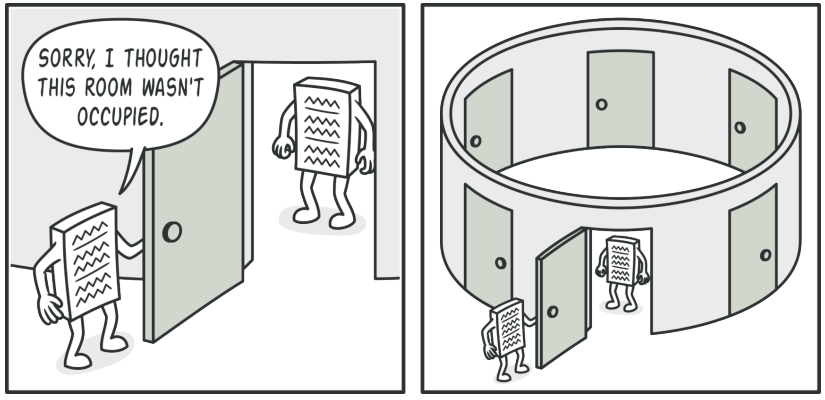
☹ Problem

The Singleton pattern solves two problems at the same time, violating the *Single Responsibility Principle*:

1. **Ensure that a class has just a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

   Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

   Note that this behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.



Clients may not even realize that they're working with the same object all the time.

2. **Provide a global access point to that instance**. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

   Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

   There's another side to this problem: you don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it.

   Nowadays, the Singleton pattern has become so popular that people may call something a *singleton* even if it solves just one of the listed problems.

😊 Solution

All implementations of the Singleton have these two steps in common:

- Make the default constructor private, to prevent other objects from using the `new` operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to

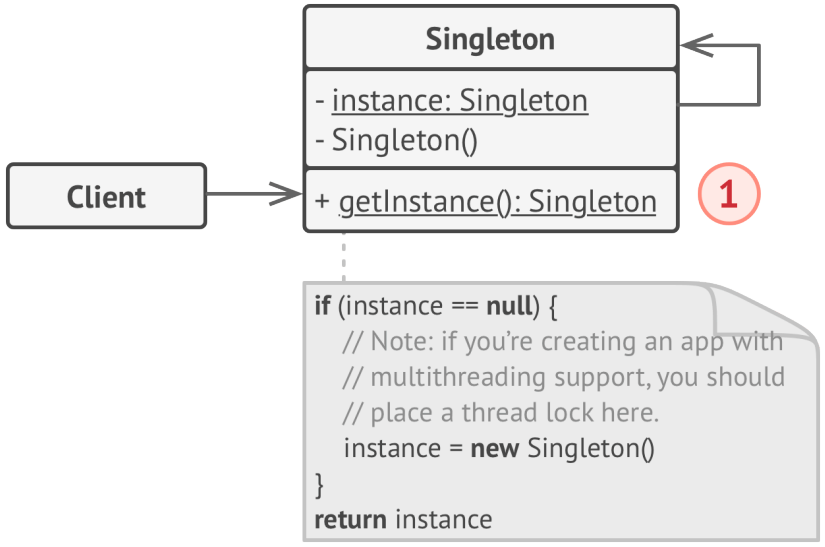
create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Real-World Analogy

The government is an excellent example of the Singleton pattern. A country can have only one official government. Regardless of the personal identities of the individuals who form governments, the title, "The Government of X", is a global point of access that identifies the group of people in charge.

Structure



1. The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

   The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

Pseudocode

In this example, the database connection class acts as a **Singleton**.

This class doesn't have a public constructor, so the only way to get its object is to call the `getInstance` method. This method caches the first created object and returns it in all subsequent calls.

```
// The Database class defines the `getInstance` method that lets
// clients access the same instance of a database connection
// throughout the program.
class Database is
  // The field for storing the singleton instance should be
  // declared static.
  private static field instance: Database

  // The singleton's constructor should always be private to
  // prevent direct construction calls with the `new`
  // operator.
  private constructor Database() is
```

```
    // Some initialization code, such as the actual
    // connection to a database server.
    // ...

  // The static method that controls access to the singleton
  // instance.
  public static method getInstance() is
    if (this.instance == null) then
      acquireThreadLock() and then
        // Ensure that the instance hasn't yet been
        // initialized by another thread while this one
        // has been waiting for the lock's release.
        if (this.instance == null) then
          this.instance = new Database()
    return this.instance

  // Finally, any singleton should define some business logic
  // which can be executed on its instance.
  public method query(sql) is
    // For instance, all database queries of an app go
    // through this method. Therefore, you can place
    // throttling or caching logic here.
    // ...

class Application is
  method main() is
    Database foo = Database.getInstance()
    foo.query("SELECT ...")
    // ...
    Database bar = Database.getInstance()
    bar.query("SELECT ...")
    // The variable `bar` will contain the same object as
```

```
45      // the variable `foo`.
```

Applicability

Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

The Singleton pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.

Use the Singleton pattern when you need stricter control over global variables.

Unlike global variables, the Singleton pattern guarantees that there's just one instance of a class. Nothing, except for the Singleton class itself, can replace the cached instance.

Note that you can always adjust this limitation and allow creating any number of Singleton instances. The only piece of code that needs changing is the body of the `getInstance()` method.

How to Implement

1. Add a private static field to the class for storing the singleton instance.

2. Declare a public static creation method for getting the singleton instance.

3. Implement "lazy initialization" inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.

4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.

5. Go over the client code and replace all direct calls to the singleton's constructor with calls to its static creation method.

Pros and Cons

✓ You can be sure that a class has only a single instance.

✓ You gain a global access point to that instance.

✓ The singleton object is initialized only when it's requested for the first time.

✕ Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.

- ✕ The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- ✕ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- ✕ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

⇄ Relations with Other Patterns

- A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:

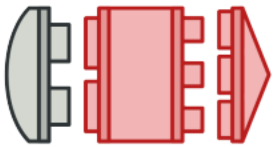
  1. There should be only one Singleton instance, whereas a *Flyweight* class can have multiple instances with different intrinsic states.

2. The *Singleton* object can be mutable. Flyweight objects are immutable.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Structural Design Patterns

Structural patterns explain how to assemble objects and classes into larger structures, while keeping this structures flexible and efficient.

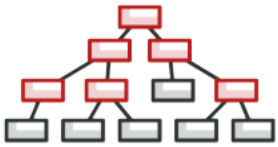


Adapter

Provides a unified interface that allows objects with incompatible interfaces to collaborate.

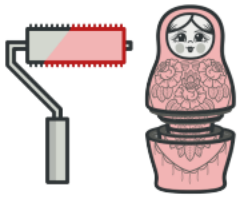
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

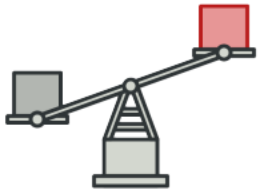


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

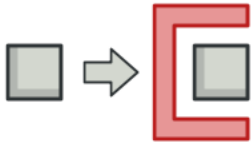
Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



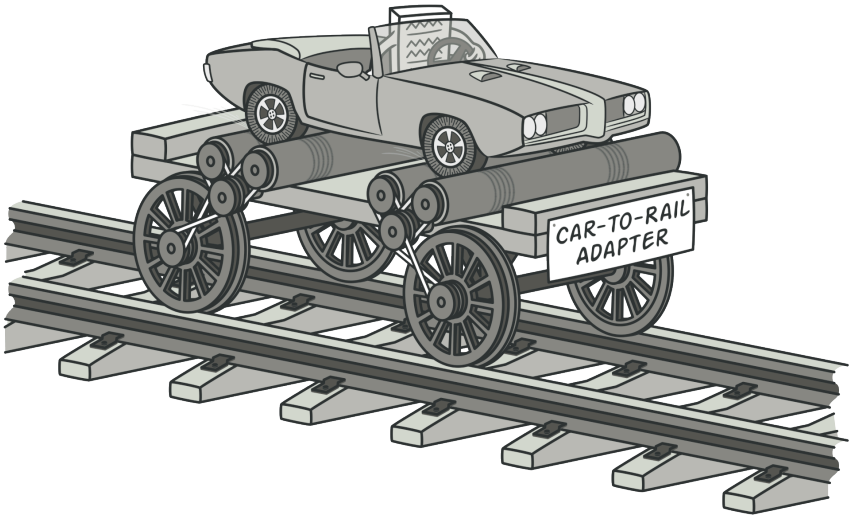
Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects, instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



ADAPTER

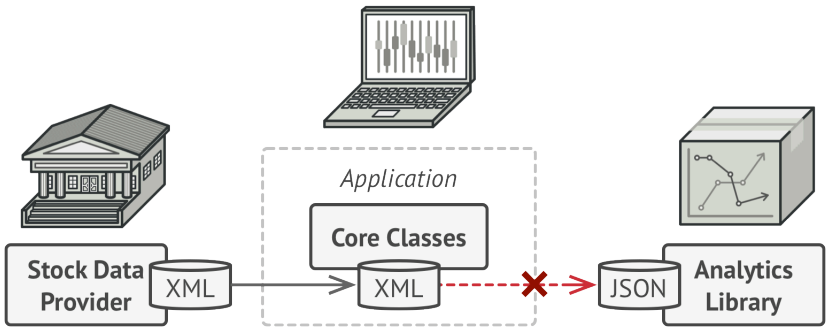
Also known as: Wrapper

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Problem

Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



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

You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.

😊 Solution

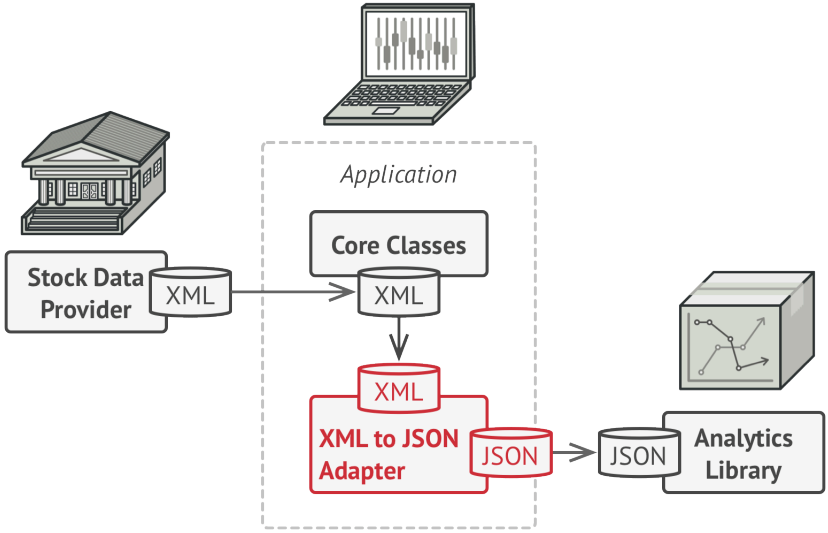
You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

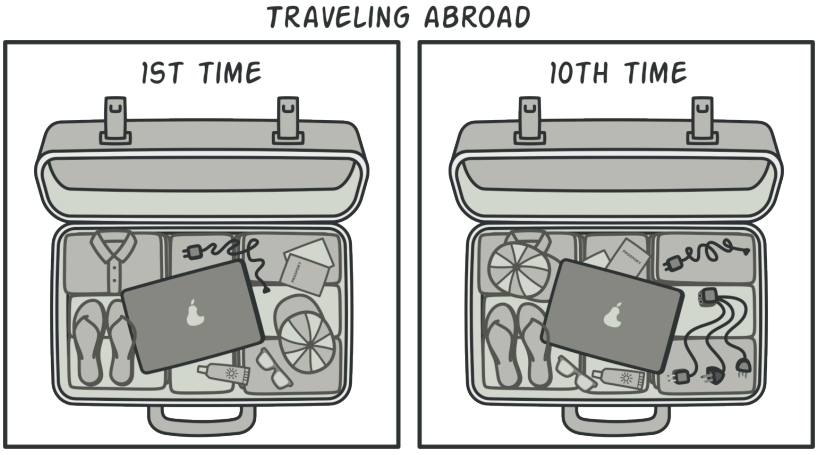
Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.



Let's get back to our stock market app. To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly. Then you adjust your code to communicate with the library only via these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

Real-World Analogy

When you travel from the US to Europe for the first time, you may get a surprise when trying to charge your laptop. The power plug and sockets standards are different in different countries.



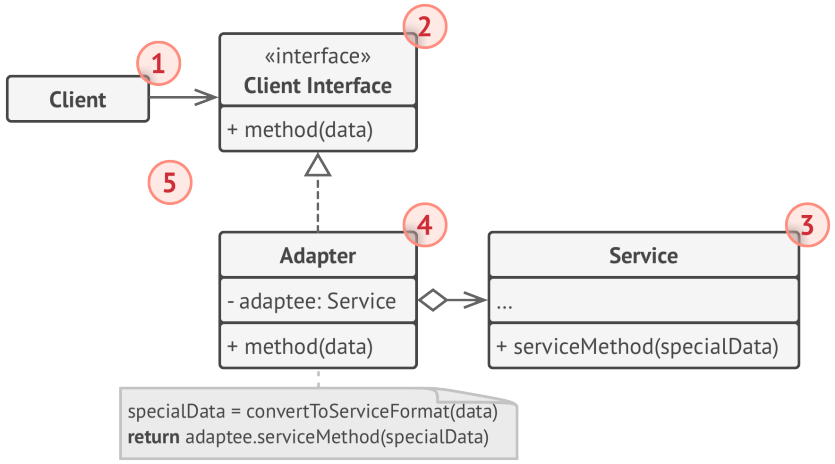
A suitcase before and after a trip abroad.

That's why your US plug won't fit a German socket. The problem can be solved by using a power plug adapter that has the American-style socket and the European-style plug.

Structure

Object adapter

This implementation uses the composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.



1. The **Client** is a class that contains the existing business logic of the program.

2. The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3. The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

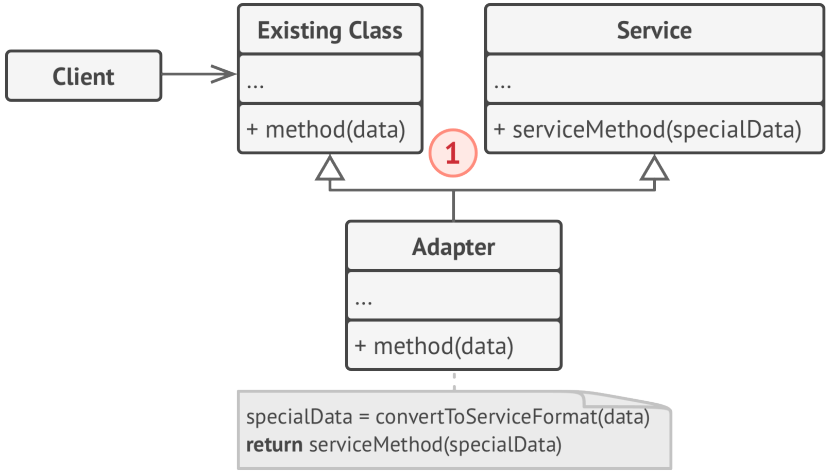
4. The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

5. The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client inter-

face. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

Class adapter

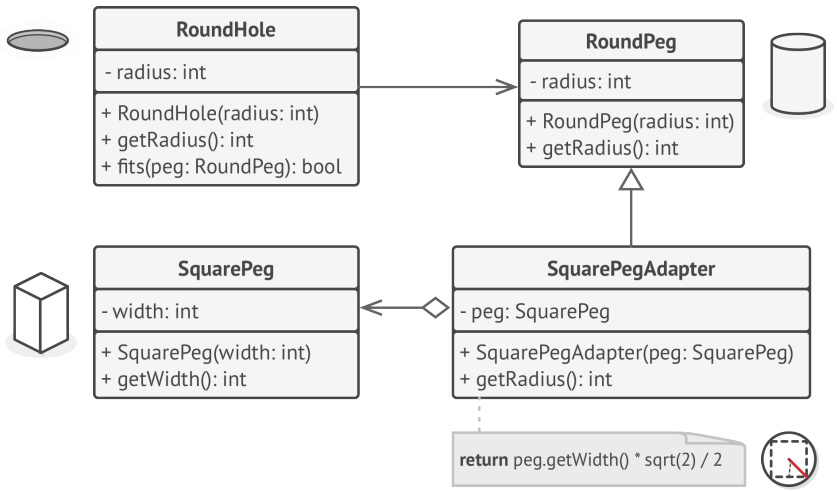
This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.



1. The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Pseudocode

This example of the **Adapter** pattern is based on the classic conflict between square pegs and round holes.



Adapting square pegs to round holes.

The Adapter pretends to be a round peg, with a radius equal to a half of the square's diameter (in other words, the radius of the smallest circle that can accommodate the square peg).

```
// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
  constructor RoundHole(radius) { ... }

  method getRadius() is
    // Return the radius of the hole.
```

```
8
9     method fits(peg: RoundPeg) is
10      return this.getRadius() >= peg.radius()
11
12  class RoundPeg is
13    constructor RoundPeg(radius) { ... }
14
15    method getRadius() is
16      // Return the radius of the peg.
17
18
19  // But there's an incompatible class: SquarePeg.
20  class SquarePeg is
21    constructor SquarePeg(width) { ... }
22
23    method getWidth() is
24      // Return the square peg width.
25
26
27  // An adapter class lets you fit square pegs into round holes.
28  // It extends the RoundPeg class to let the adapter objects act
29  // as round pegs.
30  class SquarePegAdapter extends RoundPeg is
31    // In reality, the adapter contains an instance of the
32    // SquarePeg class.
33    private field peg: SquarePeg
34
35    constructor SquarePegAdapter(peg: SquarePeg) is
36      this.peg = peg
37
38    method getRadius() is
39      // The adapter pretends that it's a round peg with a
```

```
    // radius that could fit the square peg that the adapter
    // actually wraps.
    return peg.getWidth() * Math.sqrt(2) / 2


// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

Applicability

Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.

The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.

Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes, which **smells really bad**.

The much more elegant solution would be to put the missing functionality into an adapter class. Then you would wrap objects with missing features inside the adapter, gaining needed features dynamically. For this to work, the target classes must have a common interface, and the adapter's field should follow that interface. This approach looks very similar to the **Decorator** pattern.

How to Implement

1. Make sure that you have at least two classes with incompatible interfaces:
    - A useful *service* class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
    - One or several *client* classes that would benefit from using the service class.
2. Declare the client interface and describe how clients communicate with the service.

3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

4. Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.

5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.

6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Pros and Cons

✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.

✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

✕ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's

simpler just to change the service class so that it matches the rest of your code.

⇄ Relations with Other Patterns

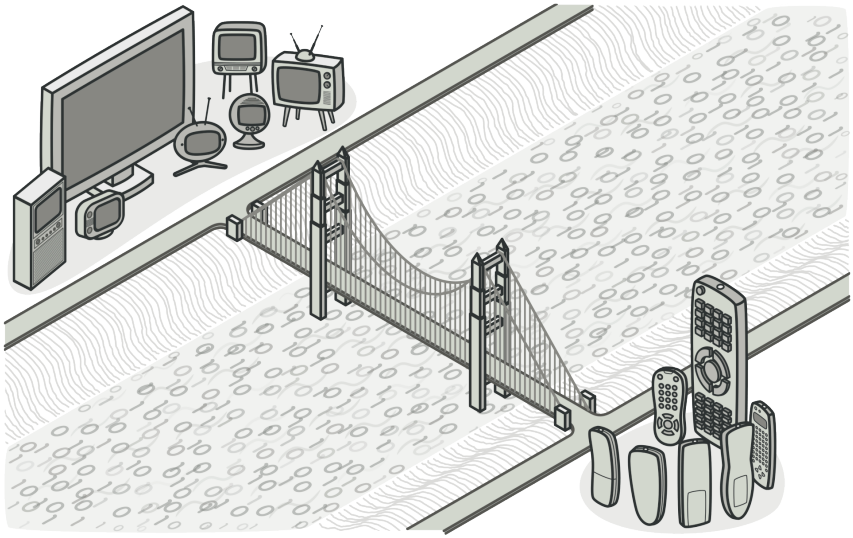
- **Bridge** is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, **Adapter** is commonly used with an existing app to make some otherwise-incompatible classes work together nicely.

- **Adapter** changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*.

- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.

- **Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. *Adapter* usually wraps just one object, while *Facade* works with an entire subsystem of objects.

- **Bridge**, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.



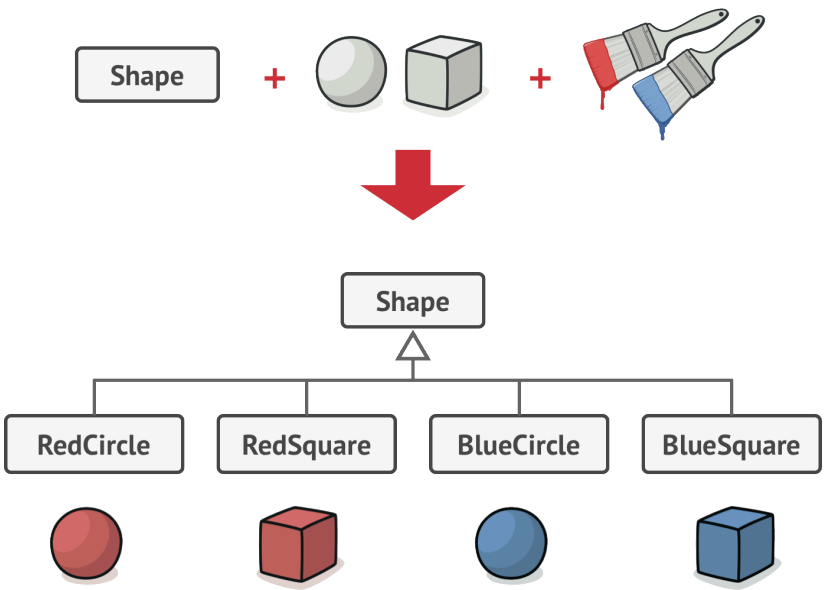
BRIDGE

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

☹ Problem

Abstraction? Implementation? Sound scary? Stay calm and let's consider a simple example.

Say you have a geometric `Shape` class with a pair of subclasses: `Circle` and `Square`. You want to extend this class hierarchy to incorporate colors, so you plan to create `Red` and `Blue` shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as `BlueCircle` and `RedSquare`.



Number of class combinations grows in geometric progression.

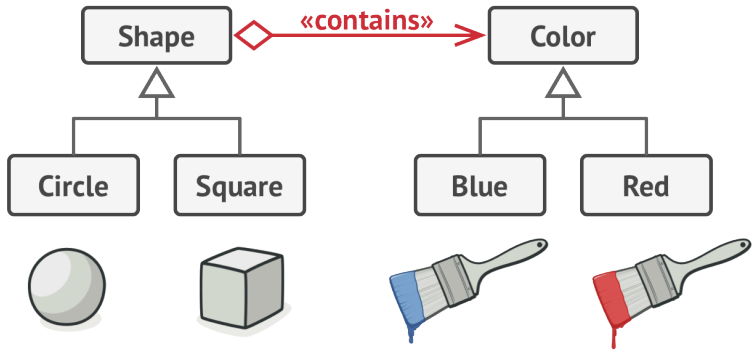
Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape you'd

need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

😊 Solution

This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.

The Bridge pattern attempts to solve this problem by switching from inheritance to composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.



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

Following this approach, we can extract the color-related code into its own class with two subclasses: `Red` and `Blue`. The `Shape` class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the `Shape` and `Color` classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.

Abstraction and Implementation

The GoF book[1] introduces the terms *Abstraction* and *Implementation* as part of the Bridge definition. In my opinion, the terms sound too academic and make the pattern seem more complicated than it really is. Having read the simple example with shapes and colors, let's decipher the meaning behind the GoF book's scary words.

Abstraction (also called *interface*) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own. It should delegate the work to the *implementation* layer (also called *platform*).

Note that we're not talking about *interfaces* or *abstract classes* from your programming language. These aren't the same things.

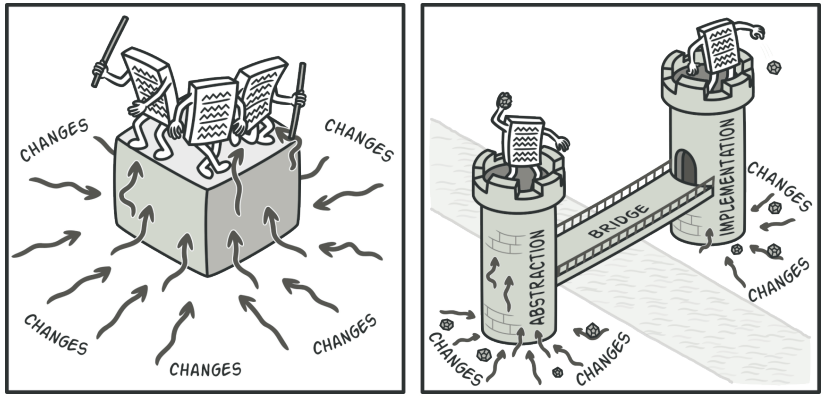
1. "Gang of Four" is a nickname given to the four authors of the original book about design patterns: *Design Patterns: Elements of Reusable Object-Oriented Software* **https://refactoring.guru/gof-book**.

When talking about real applications, the abstraction can be represented by a graphical user interface (GUI), and the implementation could be the underlying operating system code (API) which the GUI layer calls in response to user interactions.

Generally speaking, you can extend such an app in two independent directions:

- Have several different GUIs (for instance, tailored for regular customers or admins).
- Support several different APIs (for example, to be able to launch the app under Windows, Linux, and MacOS).

In a worst-case scenario, this app might look like a giant spaghetti bowl, where hundreds of conditionals connect different types of GUI with various APIs all over the code.

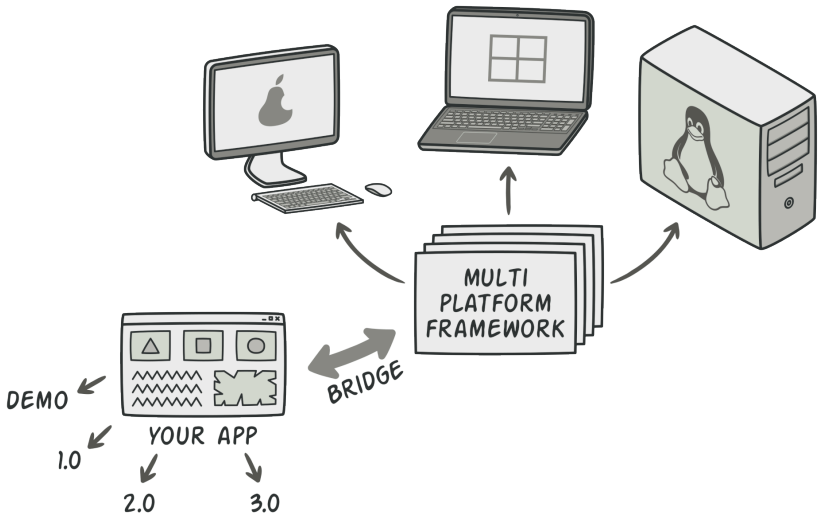


Making even a simple change to a monolithic codebase is pretty hard because you must understand the entire thing very well. Making changes to smaller, well-defined modules is much easier.

You can bring order to this chaos by extracting the code related to specific interface-platform combinations into separate classes. However, soon you'll discover that there are *lots* of these classes. The class hierarchy will grow exponentially because adding a new GUI or supporting a different API would require creating more and more classes.

Let's try to solve this issue with the Bridge pattern. It suggests that we divide the classes into two hierarchies:

- Abstraction: the GUI layer of the app.
- Implementation: the operating systems' APIs.



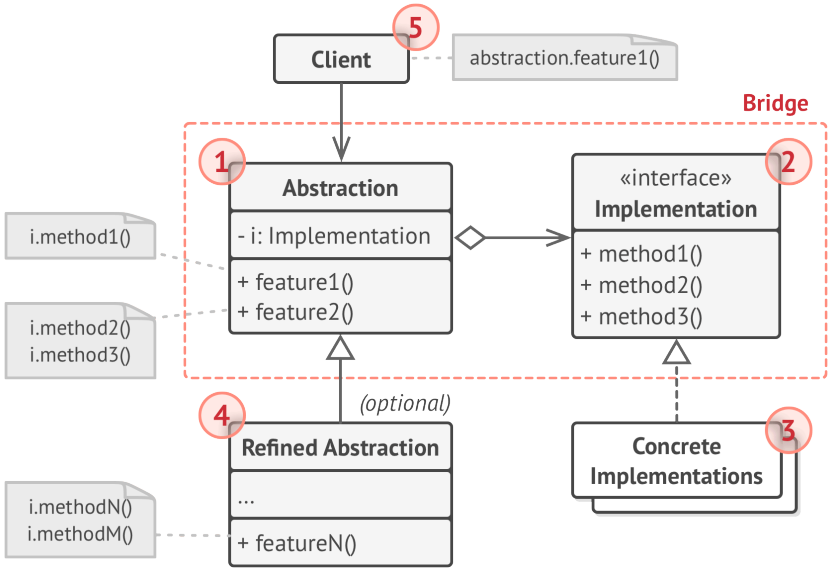
One of the ways to structure a cross-platform application.

The abstraction object controls the appearance of the app, delegating the actual work to the linked implementation object. Different implementations are interchangeable as long as they

follow a common interface, enabling the same GUI to work under Windows and Linux.

As a result, you can change the GUI classes without touching the API-related classes. Moreover, adding support for another operating system only requires creating a subclass in the implementation hierarchy.

Structure



1. The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

2. The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only com-

municate with an implementation object via methods that are declared here.

The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

3. **Concrete Implementations** contain platform-specific code.

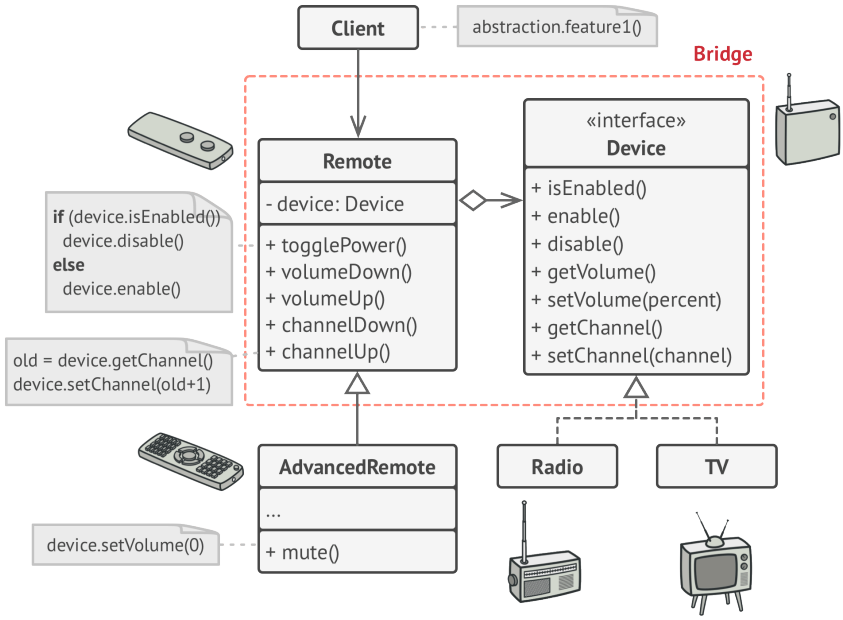
4. **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

5. Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

Pseudocode

This example illustrates how the **Bridge** pattern can help divide the monolithic code of an app that manages devices and their remote controls. The `Device` classes act as the implementation, whereas the `Remote`s act as the abstraction.

The base remote control class declares a reference field that links it with a device object. All remotes work with the devices via the general device interface, which lets the same remote support multiple device types.



The original class hierarchy is divided into two parts: devices and remote controls.

You can develop the remote control classes independently from the device classes. All that's needed is to create a new remote subclass. For example, a basic remote control might only have two buttons, but you could extend it with additional features, such as an extra battery or a touchscreen.

The client code links the desired type of remote control with a specific device object via the remote's constructor.

```
// The "abstraction" defines the interface for the "control"
// part of the two class hierarchies. It maintains a reference
// to an object of the "implementation" hierarchy and delegates
// all of the real work to this object.
class RemoteControl is
  protected field device: Device
  constructor RemoteControl(device: Device) is
    this.device = device
  method togglePower() is
    if (device.isEnabled()) then
      device.disable()
    else
      device.enable()
  method volumeDown() is
    device.setVolume(device.getVolume() - 10)
  method volumeUp() is
    device.setVolume(device.getVolume() + 10)
  method channelDown() is
    device.setChannel(device.getChannel() - 1)
  method channelUp() is
    device.setChannel(device.getChannel() + 1)


// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
  method mute() is
    device.setVolume(0)


// The "implementation" interface declares methods common to all
// concrete implementation classes. It doesn't have to match the
```

```
// abstraction's interface. In fact, the two interfaces can be
// entirely different. Typically the implementation interface
// provides only primitive operations, while the abstraction
// defines higher-level operations based on those primitives.
interface Device is
  method isEnabled()
  method enable()
  method disable()
  method getVolume()
  method setVolume(percent)
  method getChannel()
  method setChannel(channel)


// All devices follow the same interface.
class Tv implements Device is
  // ...

class Radio implements Device is
  // ...


// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

Applicability

Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).

The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change. The changes made to one of the variations of functionality may require making changes across the whole class, which often results in making errors or not addressing some critical side effects.

The Bridge pattern lets you split the monolithic class into several class hierarchies. After this, you can change the classes in each hierarchy independently of the classes in the others. This approach simplifies code maintenance and minimizes the risk of breaking existing code.

Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.

The Bridge suggests that you extract a separate class hierarchy for each of the dimensions. The original class delegates the related work to the objects belonging to those hierarchies instead of doing everything on its own.

Use the Bridge if you need to be able to switch implementations at runtime.

Although it's optional, the Bridge pattern lets you replace the implementation object inside the abstraction. It's as easy as assigning a new value to a field.

*By the way, this last item is the main reason why so many people confuse the Bridge with the **Strategy** pattern. Remember that a pattern is more than just a certain way to structure your classes. It may also communicate intent and a problem being addressed.*

How to Implement

1. Identify the orthogonal dimensions in your classes. These independent concepts could be: abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation.

2. See what operations the client needs and define them in the base abstraction class.

3. Determine the operations available on all platforms. Declare the ones that the abstraction needs in the general implementation interface.

4. For all platforms in your domain create concrete implementation classes, but make sure they all follow the implementation interface.

5. Inside the abstraction class, add a reference field for the implementation type. The abstraction delegates most of the

work to the implementation object that's referenced in that field.

6. If you have several variants of high-level logic, create refined abstractions for each variant by extending the base abstraction class.

7. The client code should pass an implementation object to the abstraction's constructor to associate one with the other. After that, the client can forget about the implementation and work only with the abstraction object.

⚖ Pros and Cons

✓ You can create platform-independent classes and apps.

✓ The client code works with high-level abstractions. It isn't exposed to the platform details.

✓ *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.

✓ *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

✕ You might make the code more complicated by applying the pattern to a highly cohesive class.

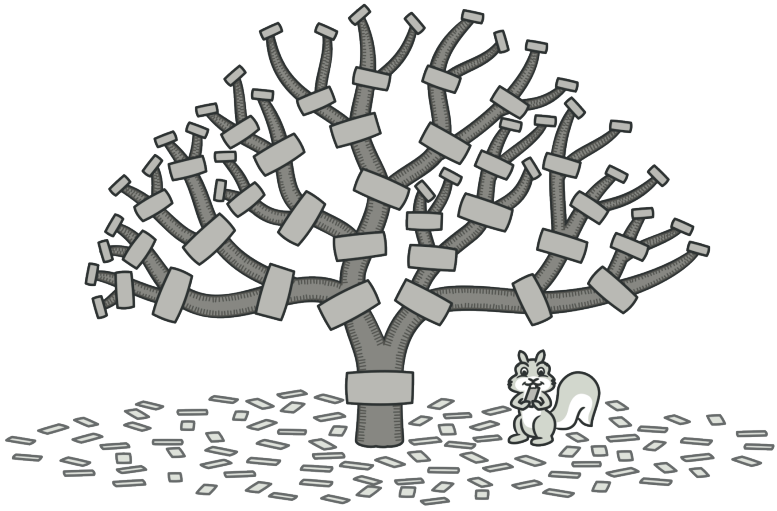
⇄ Relations with Other Patterns

- **Bridge** is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, **Adapter** is commonly used with an existing app to make some otherwise-incompatible classes work together nicely.

- **Bridge**, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.

- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by *Bridge* can only work with specific implementations. In this case, *Abstract Factory* can encapsulate these relations and hide the complexity from the client code.

- You can combine **Builder** with **Bridge**: the *director* class plays the role of the abstraction, while different *builders* act as *implementations*.



COMPOSITE

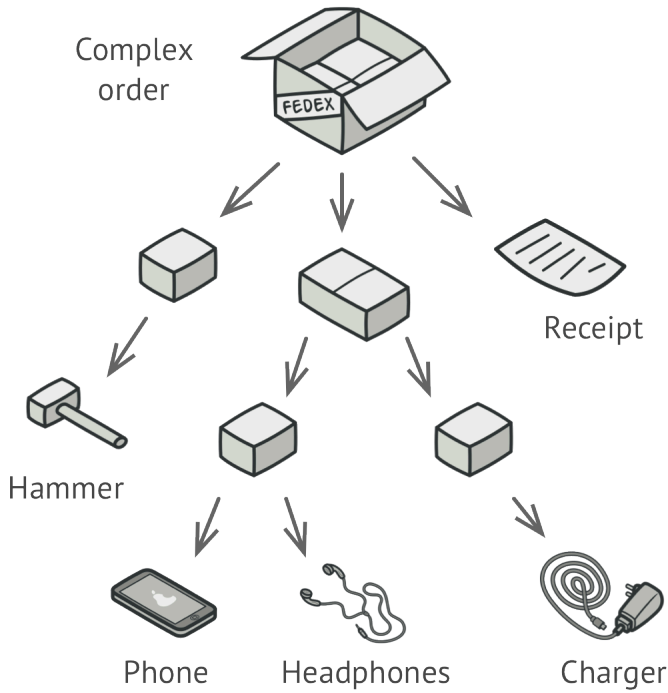
Also known as: Object Tree

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

☹ Problem

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

For example, imagine that you have two types of objects: `Products` and `Boxes`. A `Box` can contain several `Products` as well as a number of smaller `Boxes`. These little `Boxes` can also hold some `Products` or even smaller `Boxes`, and so on.



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

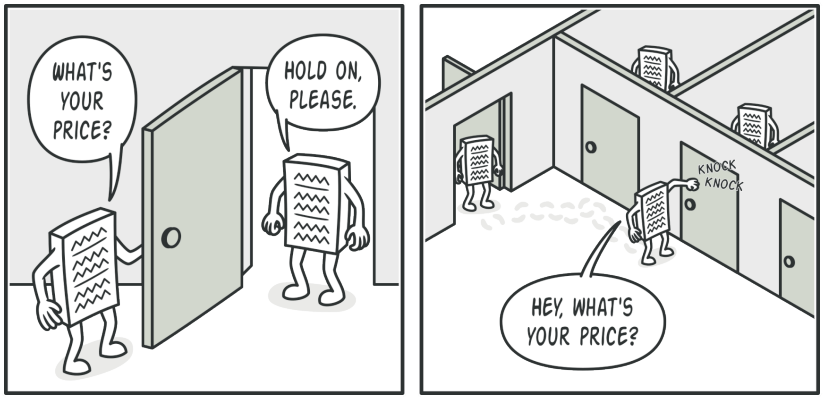
Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?

You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop. You have to know the classes of `Products` and `Boxes` you're going through, the nesting level of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

😊 Solution

The Composite pattern suggests that you work with `Products` and `Boxes` through a common interface which declares a method for calculating the total price.

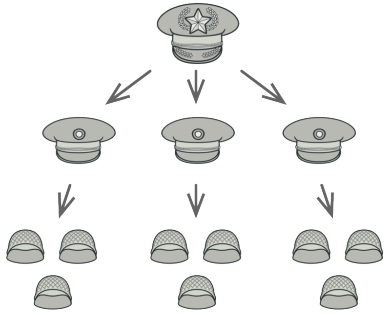
How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.



The Composite pattern lets you run a behavior recursively over all components of an object tree.

The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.

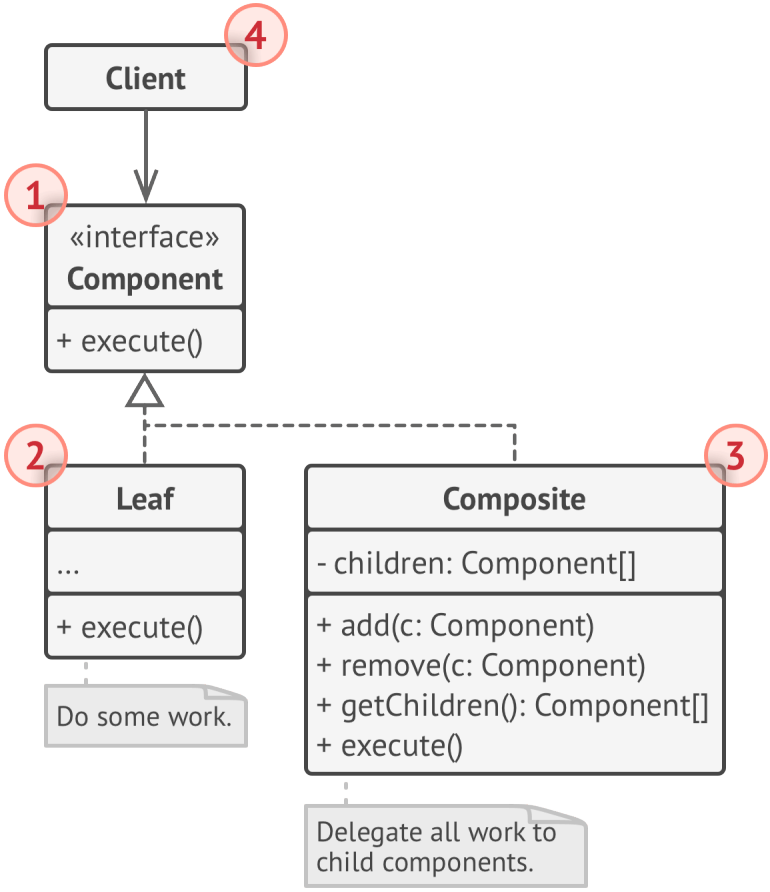
Real-World Analogy



An example of a military structure.

Armies of most countries are structured as hierarchies. An army consists of several divisions; a division is a set of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done.

Structure



1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.

2. The **Leaf** is a basic element of a tree that doesn't have sub-elements.

   Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.

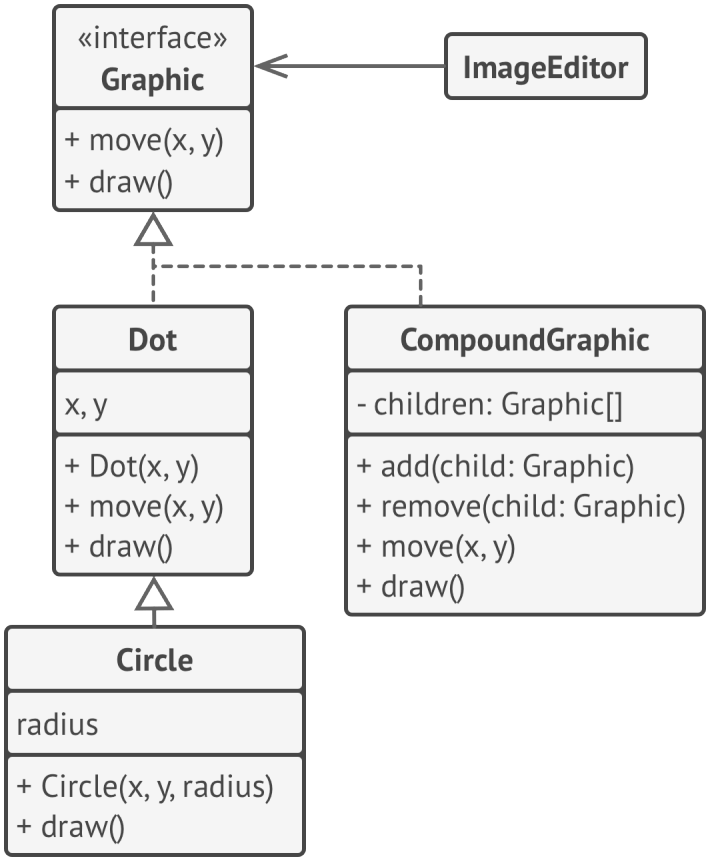
3. The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

   Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

4. The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

Pseudocode

In this example, the **Composite** pattern lets you implement stacking of geometric shapes in a graphical editor.



The geometric shapes editor example.

The `CompoundGraphic` class is a container that can comprise any number of sub-shapes, including other compound shapes. A compound shape has the same methods as a simple shape. However, instead of doing something on its own, a compound shape passes the request recursively to all its children and "sums up" the result.

The client code works with all shapes through the single interface common to all shape classes. Thus, the client doesn't

know whether it's working with a simple shape or a compound one. The client can work with very complex object structures without being coupled to concrete classes that form that structure.

```
// The component interface declares common operations for both
// simple and complex objects of a composition.
interface Graphic is
  method move(x, y)
  method draw()

// The leaf class represents end objects of a composition. A
// leaf object can't have any sub-objects. Usually, it's leaf
// objects that do the actual work, while composite objects only
// delegate to their sub-components.
class Dot implements Graphic is
  field x, y

  constructor Dot(x, y) { ... }

  method move(x, y) is
    this.x += x, this.y += y

  method draw() is
    // Draw a dot at X and Y.

// All component classes can extend other components.
class Circle extends Dot is
  field radius


```

```
    constructor Circle(x, y, radius) { ... }

    method draw() is
      // Draw a circle at X and Y with radius R.

// The composite class represents complex components that may
// have children. Composite objects usually delegate the actual
// work to their children and then "sum up" the result.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // A composite object can add or remove other components
    // (both simple or complex) to or from its child list.
    method add(child: Graphic) is
      // Add a child to the array of children.

    method remove(child: Graphic) is
      // Remove a child from the array of children.

    method move(x, y) is
      foreach (child in children) do
        child.move(x, y)

    // A composite executes its primary logic in a particular
    // way. It traverses recursively through all its children,
    // collecting and summing up their results. Since the
    // composite's children pass these calls to their own
    // children and so forth, the whole object tree is traversed
    // as a result.
    method draw() is
      // 1. For each child component:
      //     - Draw the component.
```

```
    //     - Update the bounding rectangle.
    // 2. Draw a dashed rectangle using the bounding
    // coordinates.


// The client code works with all the components via their base
// interface. This way the client code can support simple leaf
// components as well as complex composites.
class ImageEditor is
  method load() is
    all = new CompoundGraphic()
    all.add(new Dot(1, 2))
    all.add(new Circle(5, 3, 10))
    // ...

  // Combine selected components into one complex composite
  // component.
  method groupSelected(components: array of Graphic) is
    group = new CompoundGraphic()
    group.add(components)
    all.remove(components)
    all.add(group)
    // All components will be drawn.
    all.draw()
```

Applicability

Use the Composite pattern when you have to implement a tree-like object structure.

The Composite pattern provides you with two basic element types that share a common interface: simple leaves and complex containers. A container can be composed of both leaves and other containers. This lets you construct a nested recursive object structure that resembles a tree.

Use the pattern when you want the client code to treat both simple and complex elements uniformly.

All elements defined by the Composite pattern share a common interface. Using this interface, the client doesn't have to worry about the concrete class of the objects it works with.

How to Implement

1. Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.

2. Declare the component interface with a list of methods that make sense for both simple and complex components.

3. Create a leaf class to represent simple elements. A program may have multiple different leaf classes.

4. Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and

containers, so make sure it's declared with the component interface type.

While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.

5. Finally, define the methods for adding and removal of child elements in the container.

   Keep in mind that these operations can be declared in the component interface. This would violate the *Interface Segregation Principle* because the methods will be empty in the leaf class. However, the client will be able to treat all the elements equally, even when composing the tree.

Pros and Cons

✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.

✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

✕ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

⇄ Relations with Other Patterns

- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.

- **Chain of Responsibility** is often used in conjunction with **Composite**. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.

- You can use **Iterators** to traverse **Composite** trees.

- You can use **Visitor** to execute an operation over an entire **Composite** tree.

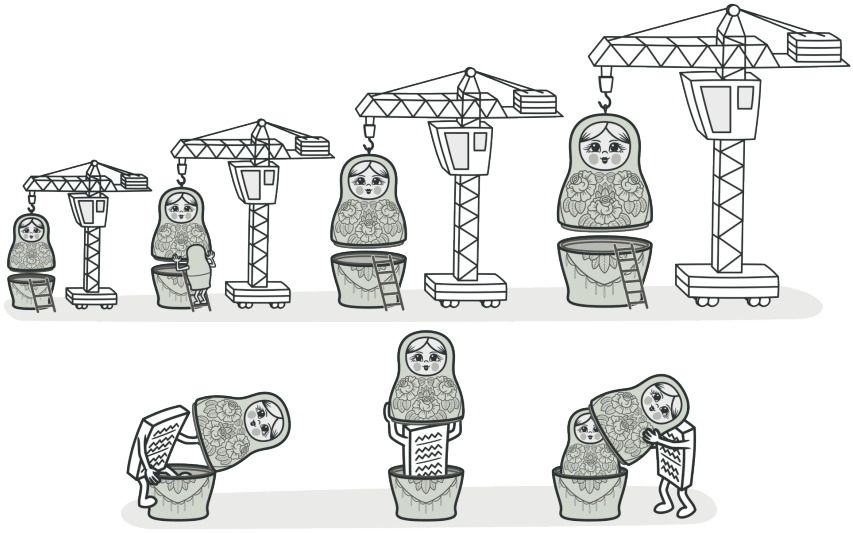
- You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM.

- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

  A *Decorator* is like a *Composite* but only has one child component. There's another significant difference: *Decorator* adds additional responsibilities to the wrapped object, while *Composite* just "sums up" its children's results.

However, the patterns can also cooperate: you can use *Decorator* to extend the behavior of a specific object in the *Composite* tree.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.



DECORATOR

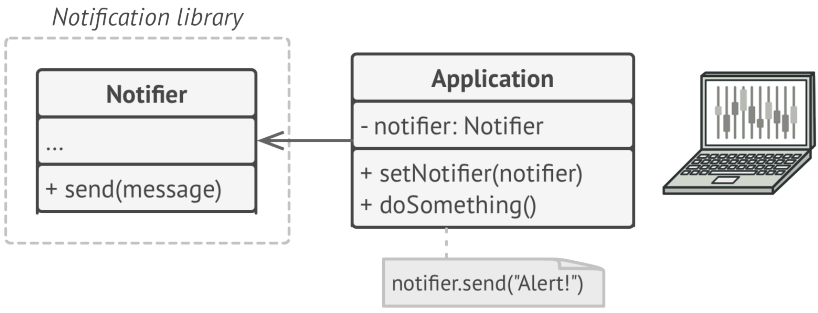
Also known as: Wrapper

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

☹ Problem

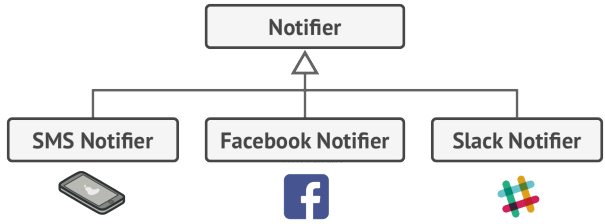
Imagine that you're working on a notification library which lets other programs notify their users about important events.

The initial version of the library was based on the `Notifier` class that had only a few fields, a constructor and a single `send` method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notificator via its constructor. A third-party app which acted as a client was supposed to create and configure the notificator object once, and then use it each time something important happened.



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

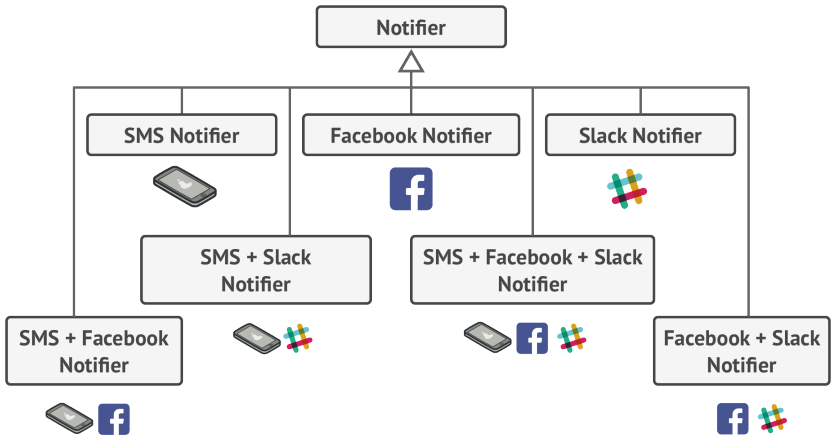
At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.



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

How hard can that be? You extended the `Notifier` class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

But then someone reasonably asked you, "Why can't you use several notification types at once? If your house is on fire, you'd probably want to be informed through every channel."



Combinatorial explosion of subclasses.

You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.

You have to find some other way to structure notifications classes so that their number won't accidentally break some Guinness record.

😊 Solution

Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.

- Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.

- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

One of the ways to overcome these caveats is by using *Composition* instead of *Inheritance*. With composition one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.

With composition, you can easily substitute the linked "helper" object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work.

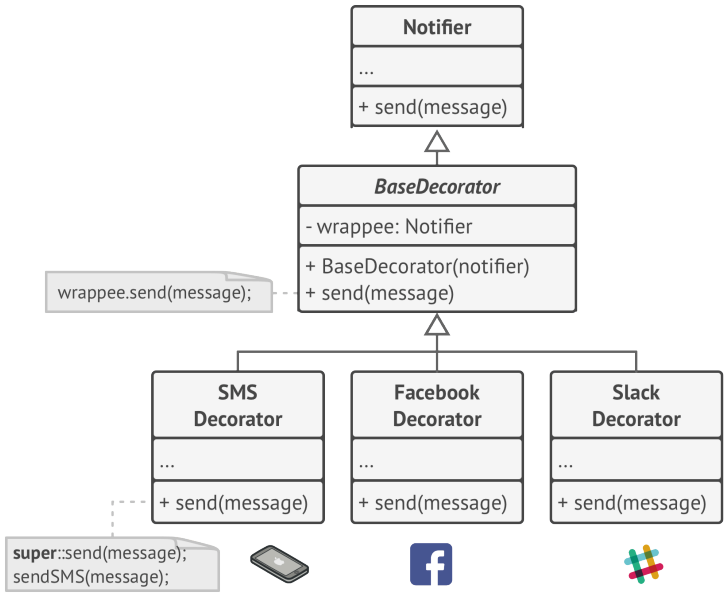
Composition is the key principle behind many design patterns, including the Decorator. On that note, let's return to the pattern discussion.

Inheritance vs. Composition

Wrapper is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern. A "wrapper" is an object that can be linked with some "target" object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

When does a simple wrapper become the real decorator? As I mentioned, the wrapper implements the same interface as the wrapped object. That's why from the client's perspective these objects are identical. Make the wrapper's reference field accept

any object that follows that interface. This will let you cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

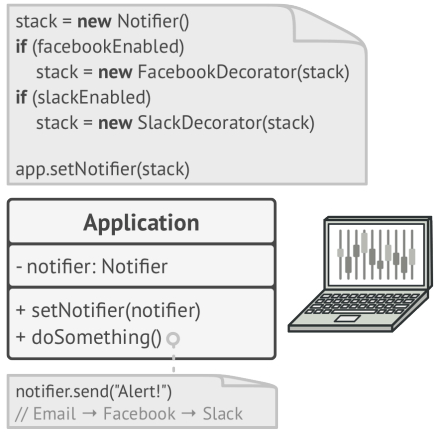


Various notification methods become decorators.

In our notifications example, let's leave the simple email notification behavior inside the base `Notifier` class, but turn all other notification methods into decorators. The client code would need to wrap a basic notifier object into a set of decorators that match the client's preferences. The resulting objects will be structured as a stack.

The last decorator in the stack would be the object that the client actually works with. Since all decorators implement the same interface as the base notifier, the rest of the client code

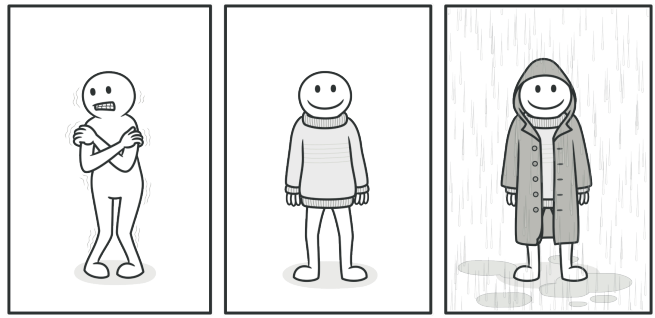
won't care whether it works with the "pure" notificator object or the decorated one.



Apps might configure complex stacks of notification decorators.

We could apply the same approach to other behaviors such as formatting messages or composing the recipient list. The client can decorate the object with any custom decorators, as long as they follow the same interface as the others.

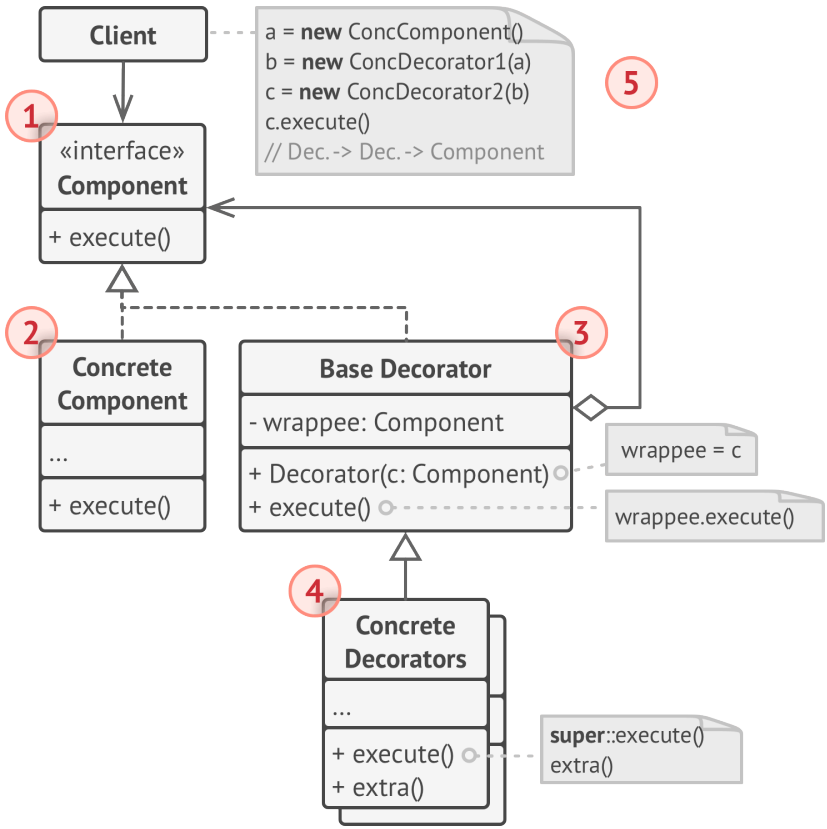
Real-World Analogy



You get a combined effect from wearing multiple pieces of clothing.

Wearing clothes is an example of using decorators. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

Structure



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

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

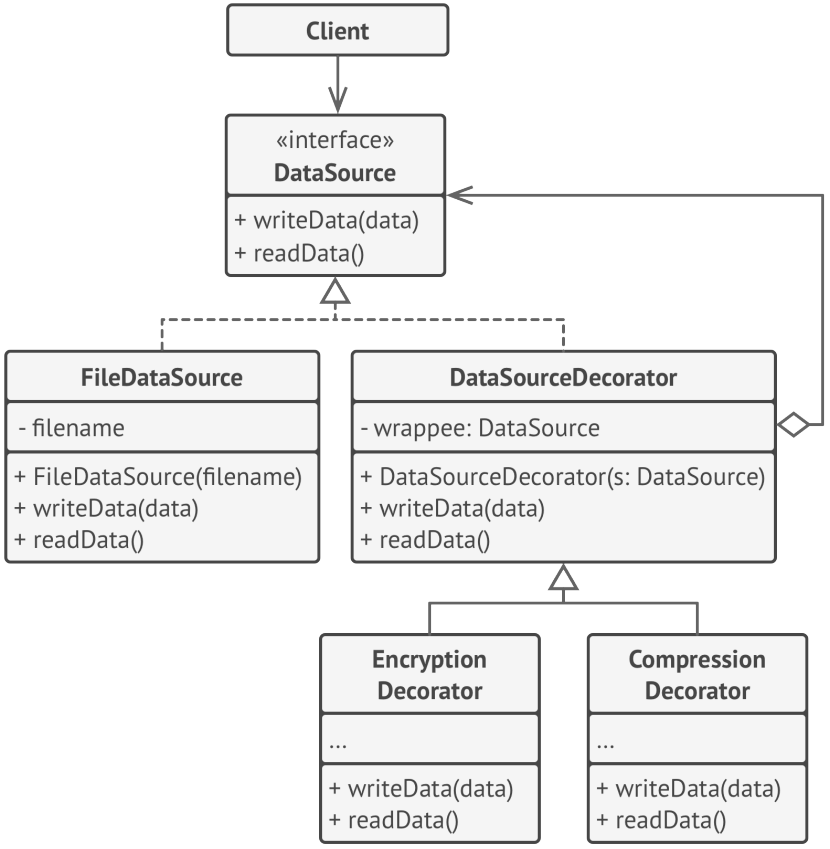
3. The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

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

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

Pseudocode

In this example, the **Decorator** pattern lets you compress and encrypt sensitive data independently from the code that actually uses this data.



The encryption and compression decorators example.

The application wraps the data source object with a pair of decorators. Both wrappers change the way the data is written to and read from the disk:

- Just before the data is **written to disk**, the decorators encrypt and compress it. The original class writes the encrypted and protected data to the file without knowing about the change.

- Right after the data is **read from disk**, it goes through the same decorators, which decompress and decode it.

  The decorators and the data source class implement the same interface, which makes them all interchangeable in the client code.

```
// The component interface defines operations that can be
// altered by decorators.
interface DataSource is
  method writeData(data)
  method readData():data

// Concrete components provide default implementations for the
// operations. There might be several variations of these
// classes in a program.
class FileDataSource implements DataSource is
  constructor FileDataSource(filename) { ... }

  method writeData(data) is
    // Write data to file.

  method readData():data is
    // Read data from file.

// The base decorator class follows the same interface as the
// other components. The primary purpose of this class is to
// define the wrapping interface for all concrete decorators.
// The default implementation of the wrapping code might include
// a field for storing a wrapped component and the means to
// initialize it.
```

```
class DataSourceDecorator implements DataSource is
  protected field wrappee: DataSource

  constructor DataSourceDecorator(source: DataSource) is
    wrappee = source

  // The base decorator simply delegates all work to the
  // wrapped component. Extra behaviors can be added in
  // concrete decorators.
  method writeData(data) is
    wrappee.writeData(data)

  // Concrete decorators may call the parent implementation of
  // the operation instead of calling the wrapped object
  // directly. This approach simplifies extension of decorator
  // classes.
  method readData():data is
    return wrappee.readData()

// Concrete decorators must call methods on the wrapped object,
// but may add something of their own to the result. Decorators
// can execute the added behavior either before or after the
// call to a wrapped object.
class EncryptionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. Encrypt passed data.
    // 2. Pass encrypted data to the wrappee's writeData
    // method.
  method readData():data is
    // 1. Get data from the wrappee's readData method.
    // 2. Try to decrypt it if it's encrypted.
    // 3. Return the result.
```

```
// You can wrap objects in several layers of decorators.
class CompressionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. Compress passed data.
    // 2. Pass compressed data to the wrappee's writeData
    // method.

  method readData():data is
    // 1. Get data from the wrappee's readData method.
    // 2. Try to decompress it if it's compressed.
    // 3. Return the result.


// Option 1. A simple example of a decorator assembly.
class Application is
  method dumbUsageExample() is
    source = new FileDataSource("somefile.dat")
    source.writeData(salaryRecords)
    // The target file has been written with plain data.

    source = new CompressionDecorator(source)
    source.writeData(salaryRecords)
    // The target file has been written with compressed
    // data.

    source = new EncryptionDecorator(source)
    // The source variable now contains this:
    // Encryption > Compression > FileDataSource
    source.writeData(salaryRecords)
    // The file has been written with compressed and
    // encrypted data.

```

```
// Option 2. Client code that uses an external data source.
// SalaryManager objects neither know nor care about data
// storage specifics. They work with a pre-configured data
// source received from the app configurator.
class SalaryManager is
  field source: DataSource

  constructor SalaryManager(source: DataSource) { ... }

  method load() is
    return source.readData()

  method save() is
    source.writeData(salaryRecords)
  // ...Other useful methods...


// The app can assemble different stacks of decorators at
// runtime, depending on the configuration or environment.
class ApplicationConfigurator is
  method configurationExample() is
    source = new FileDataSource("salary.dat")
    if (enabledEncryption)
      source = new EncryptionDecorator(source)
    if (enabledCompression)
      source = new CompressionDecorator(source)

    logger = new SalaryManager(source)
    salary = logger.load()
  // ...
```

Applicability

Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

The Decorator lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface.

Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

Many programming languages have the `final` keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

How to Implement

1. Make sure your business domain can be represented as a primary component with multiple optional layers over it.

2. Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.

3. Create a concrete component class and define the base behavior in it.

4. Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.

5. Make sure all classes implement the component interface.

6. Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behavior before or after the call to the parent method (which always delegates to the wrapped object).

7. The client code must be responsible for creating decorators and composing them in the way the client needs.

Pros and Cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.

✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

✕ It's hard to remove a specific wrapper from the wrappers stack.

✕ It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.

✕ The initial configuration code of layers might look pretty ugly.

⇄ Relations with Other Patterns

- **Adapter** changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*.

- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.

- **Chain of Responsibility** and **Decorator** have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

  The *CoR* handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various *Decorators* can extend the object's behavior while keeping it consistent with

the base interface. In addition, decorators aren't allowed to break the flow of the request.

- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

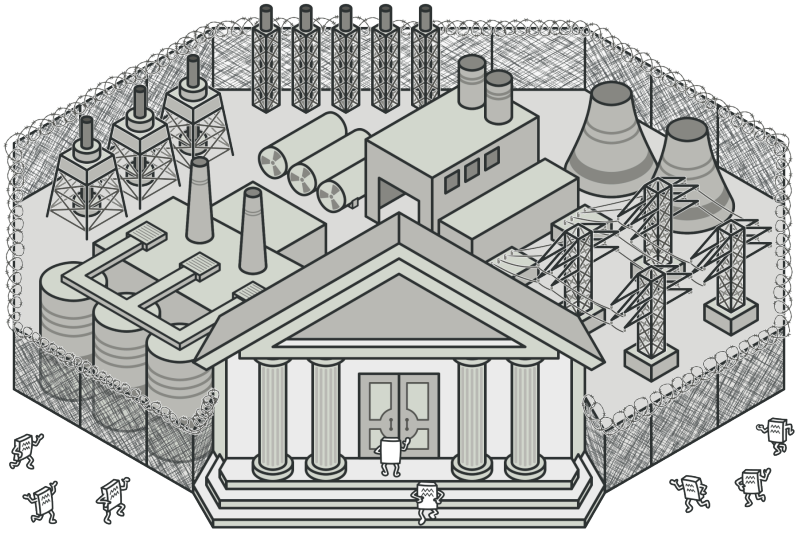
  A *Decorator* is like a *Composite* but only has one child component. There's another significant difference: *Decorator* adds additional responsibilities to the wrapped object, while *Composite* just "sums up" its children's results.

  However, the patterns can also cooperate: you can use *Decorator* to extend the behavior of a specific object in the *Composite* tree.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.

- **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts.

- **Decorator** and **Proxy** have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.



FACADE

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