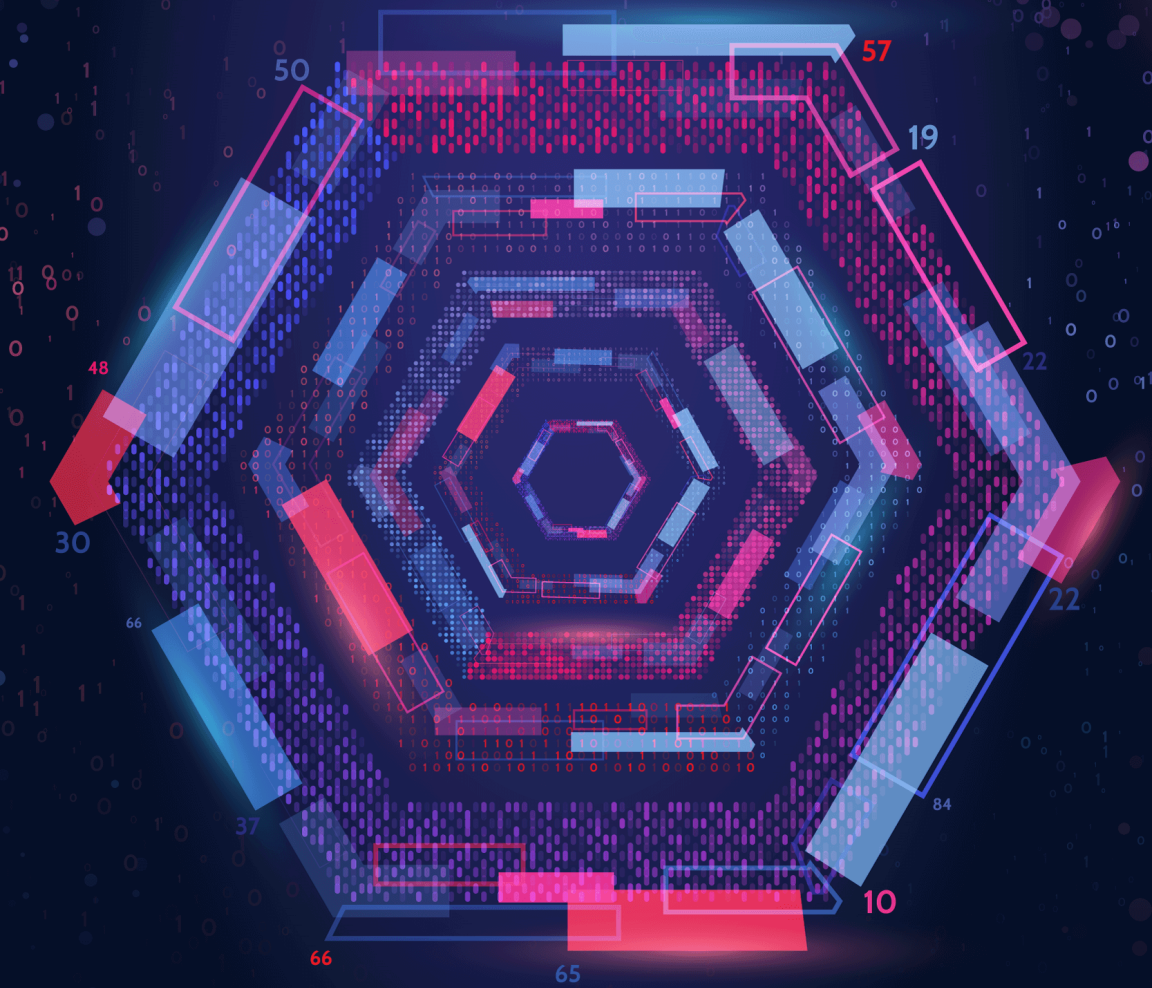
Dive Into DESIGN PATTERNS



Alexander Shvets

Dive Into
DESIGN PATTERNS

v2018-1.1

DEMO VERSION

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.

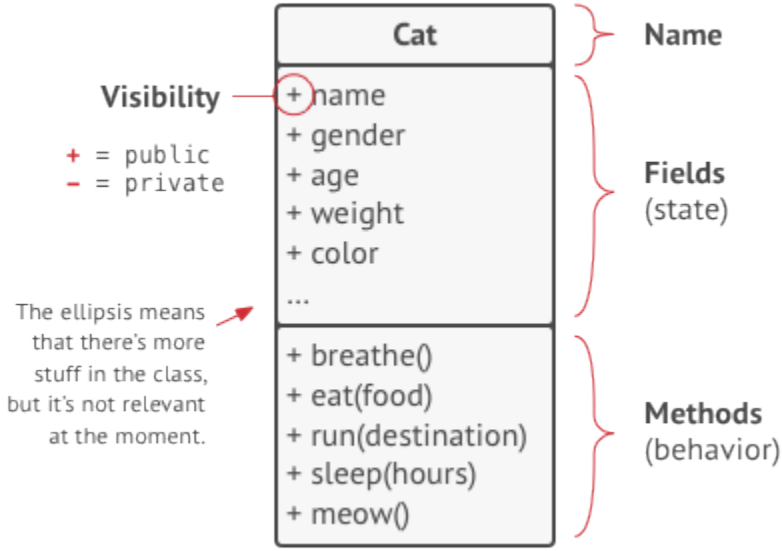
INTRODUCTION TO OOP

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.

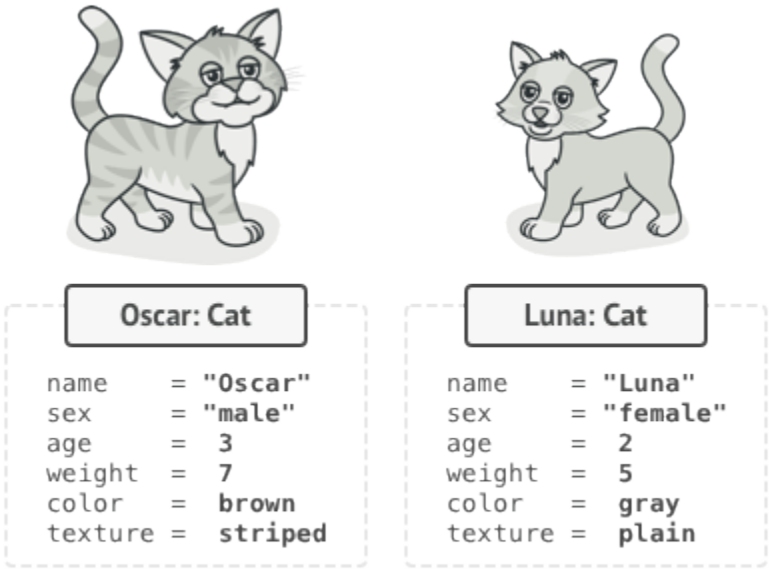


This is a UML class diagram. You'll see a lot of such diagrams in the book.

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*.



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.

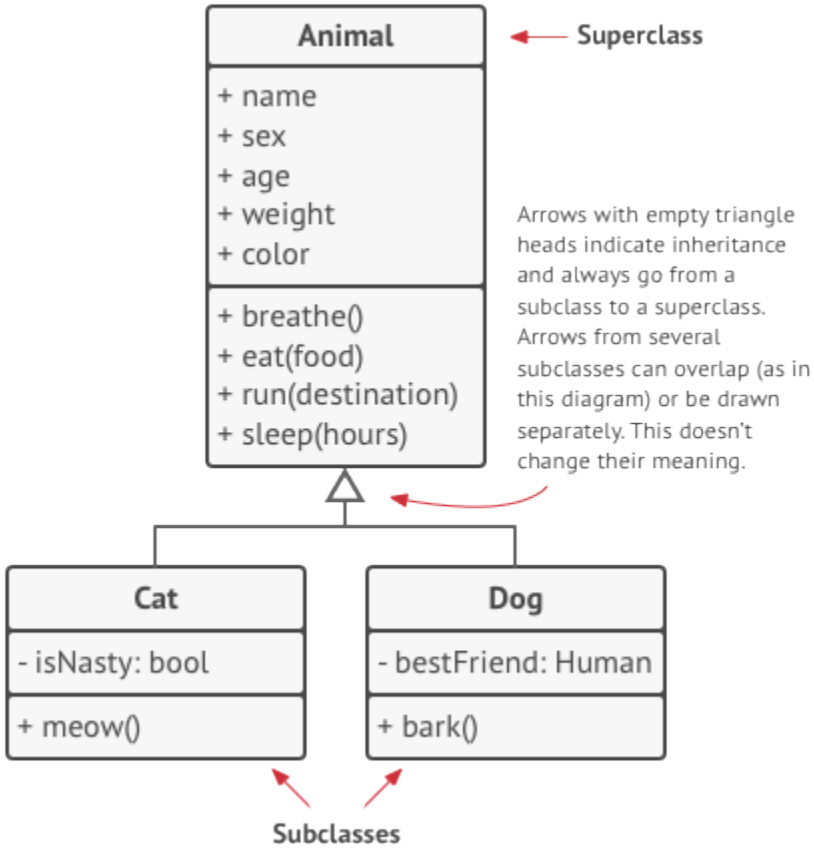
So a *class* is like a blueprint that defines the structure for *objects*, which are concrete instances of that class.

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.

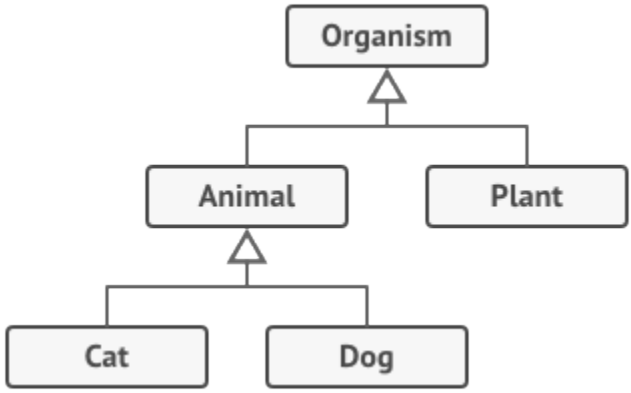
Say your neighbor has a dog called Fido. It turns out, dogs and cats have a lot in common: name, sex, age, and color are attributes of both dogs and cats. Dogs can breathe, sleep and run the same way cats do. So it seems that we can define the base `Animal` class that would list the common attributes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

16 pages

from the full book are omitted in the demo version

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.

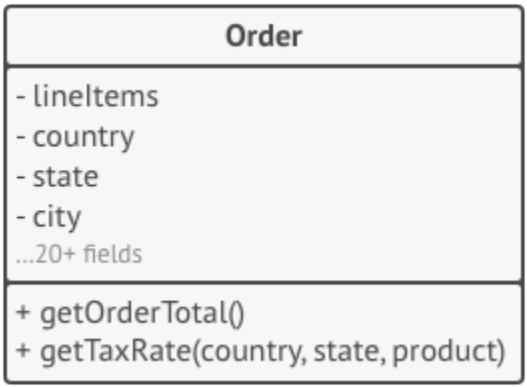
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  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.

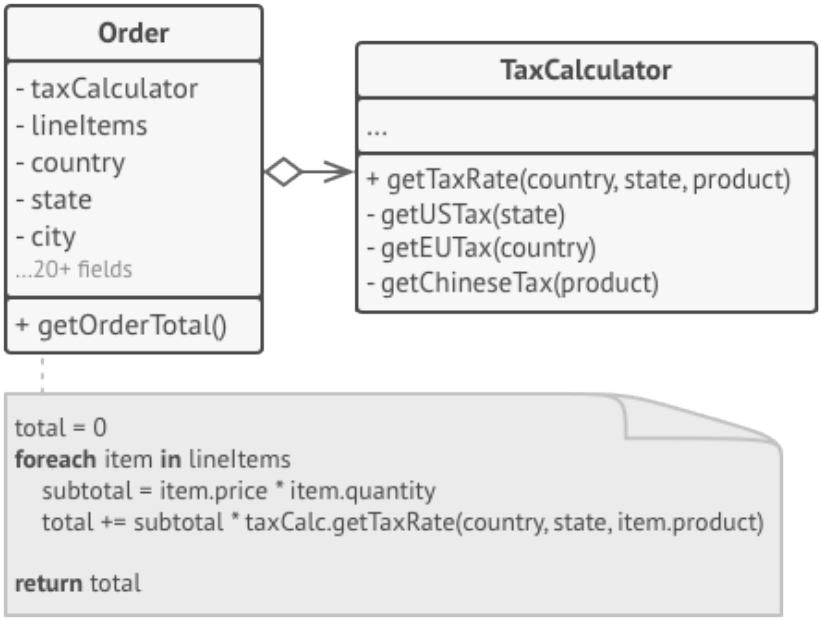
Encapsulation on a class level

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.

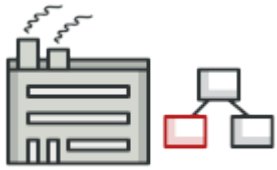


AFTER: tax calculation is hidden from the order class.

CATALOG OF DESIGN PATTERNS

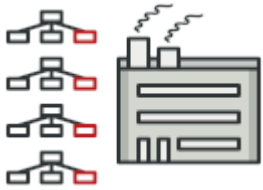
Creational Design Patterns

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.

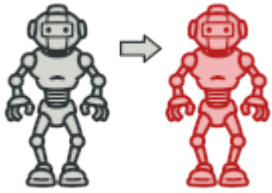


Abstract Factory

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

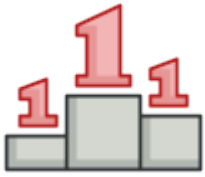
Builder

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.



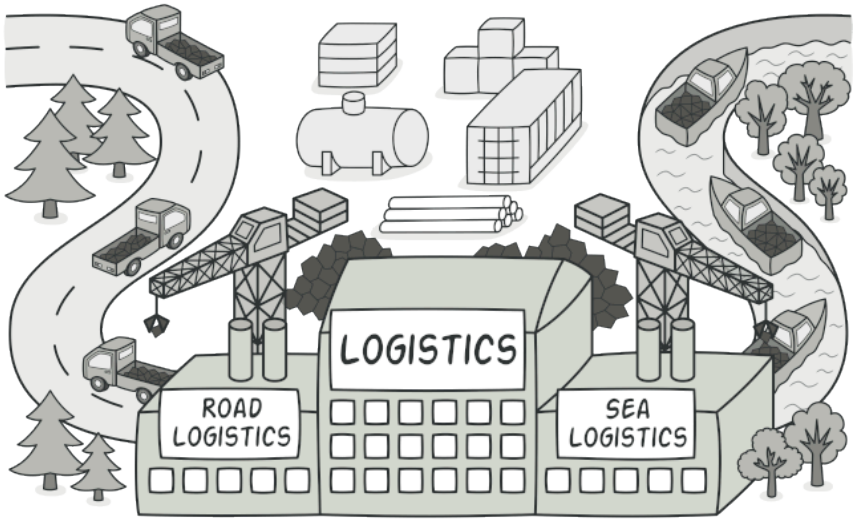
Prototype

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.

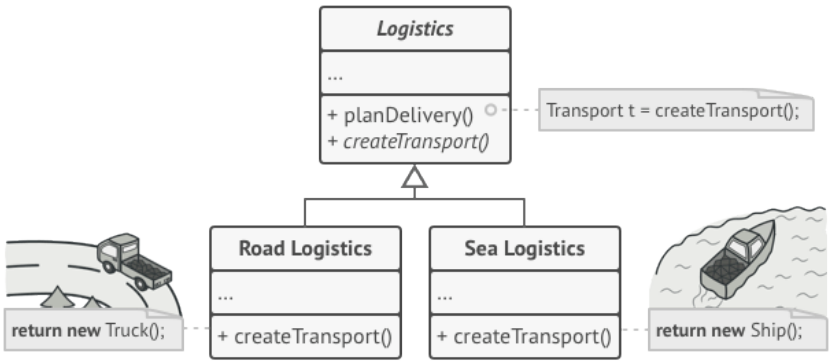
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

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

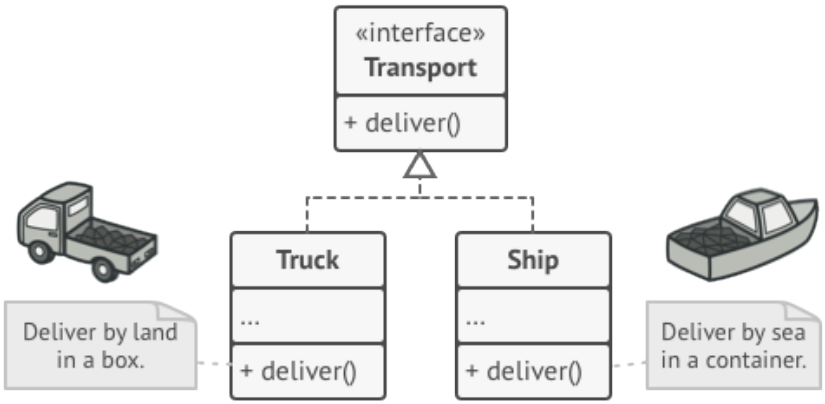
The Factory Method pattern suggests that you replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Don't worry: the objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory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

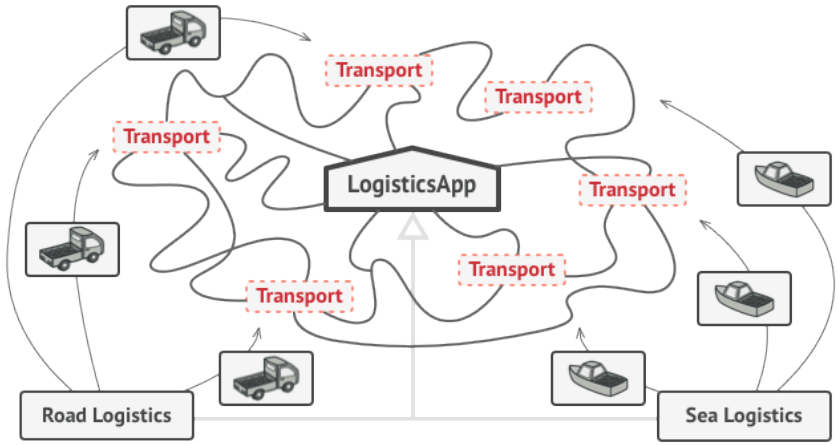
There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.



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.

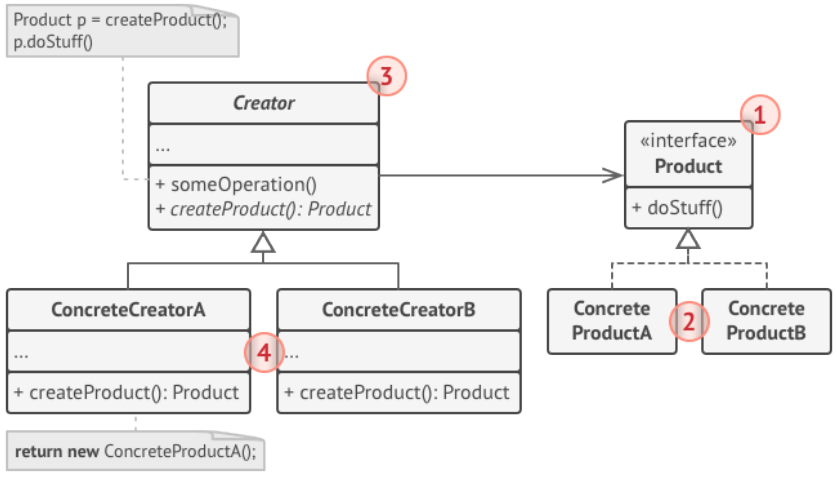
The code that uses the factory method (often called the *client* code) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as abstract `Transport`.



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.

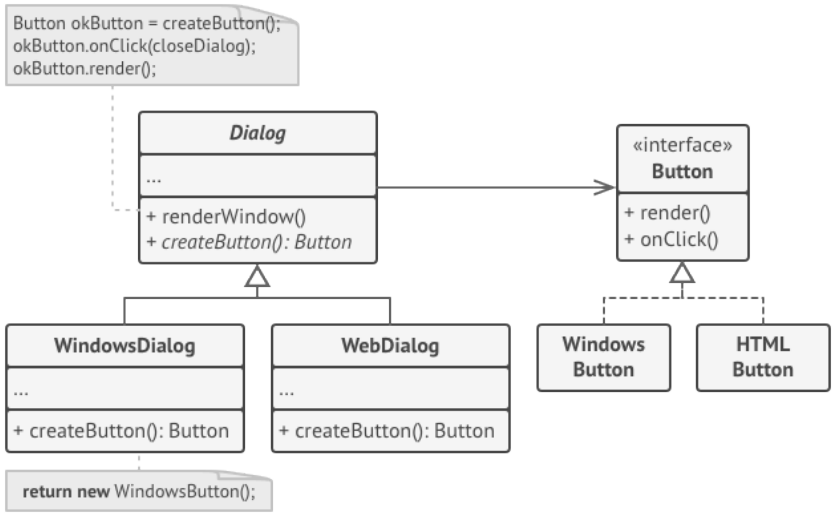
4. **Concrete Creators** override the base factory method so it returns a different type of product.

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 renderWindow() 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.
```

```
  method main() is
    dialog.initialize()
    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*.

320 pages

from the full book are omitted in the demo version