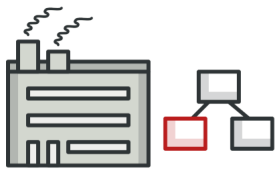
CATALOG OF DESIGN PATTERNS

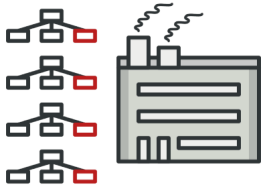
Creational Design Patterns

Creational design 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.

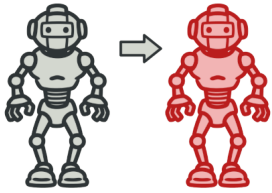


Abstract Factory

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

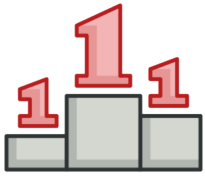
Builder

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.



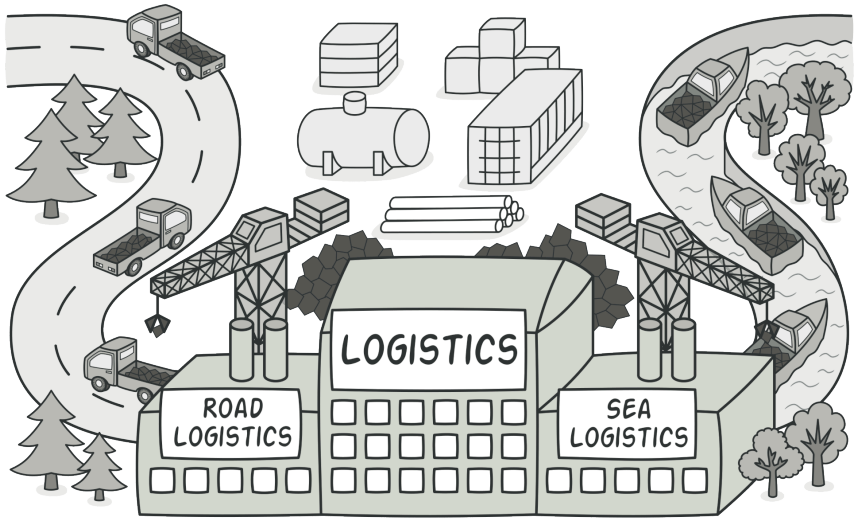
Prototype

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

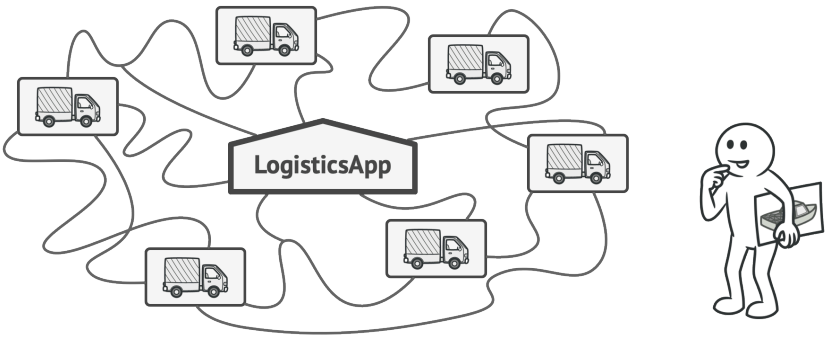
Also known as: Virtual Constructor

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.



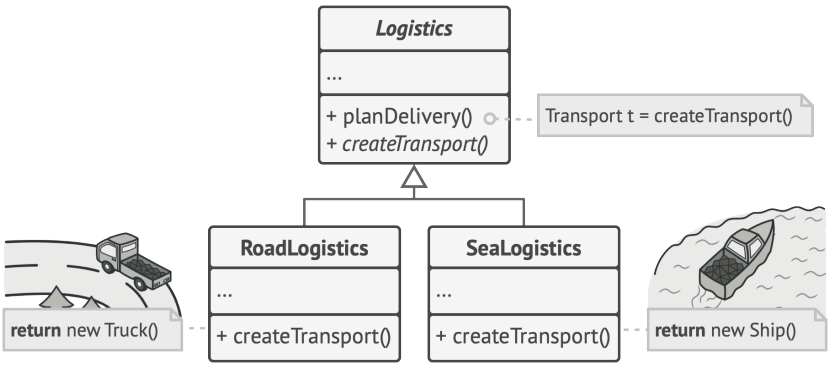
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

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

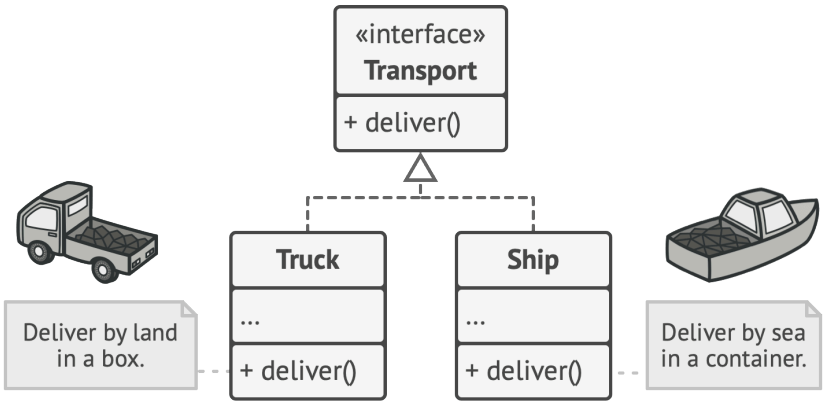
The Factory Method pattern suggests that you replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Don't worry: the objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory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

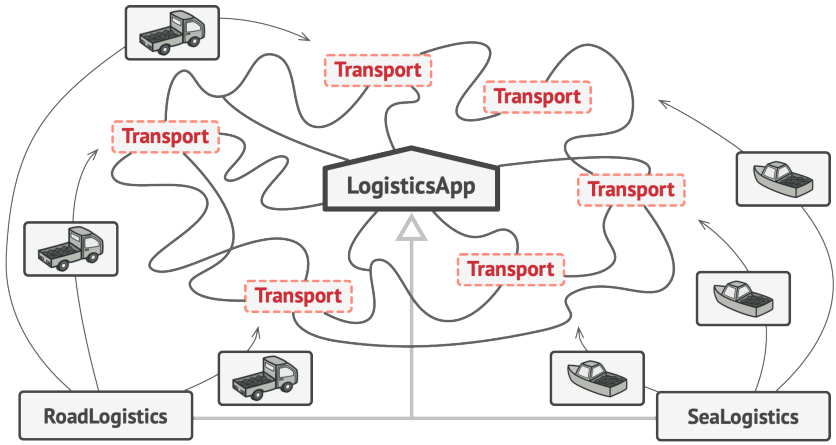
There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.



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.

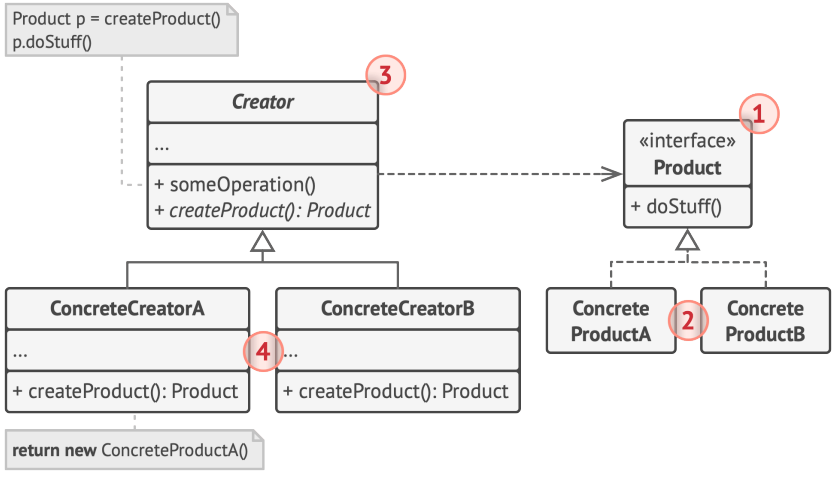
The code that uses the factory method (often called the *client* code) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as abstract `Transport`.



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.

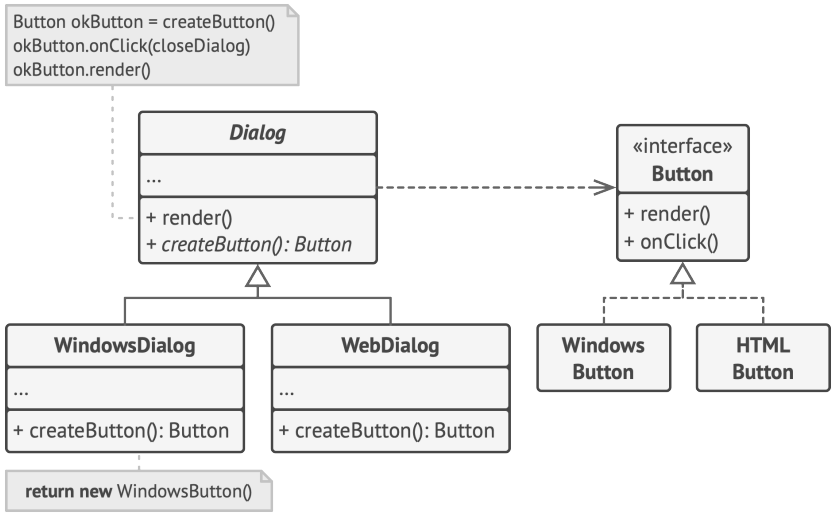
4. **Concrete Creators** override the base factory method so it returns a different type of product.

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` class for each operating system. If we declare a factory method that produces buttons inside the base `Dialog` class, we can later create a subclass that returns Windows-styled buttons from the factory method. The subclass then inherits most of the 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 code within `Dialog` 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():Button

  // 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():Button is
    return new WindowsButton()

class WebDialog extends Dialog is
  method createButton():Button 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` statement 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

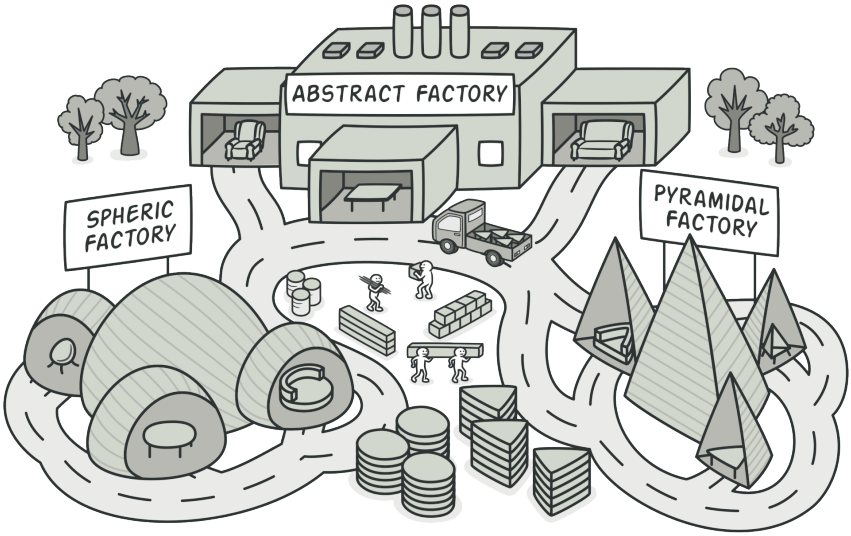
- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **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

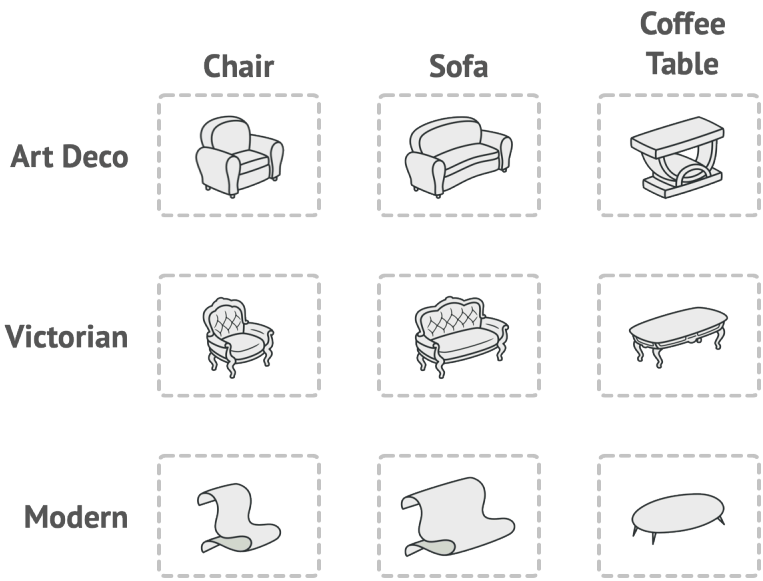
Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

☹ Problem

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

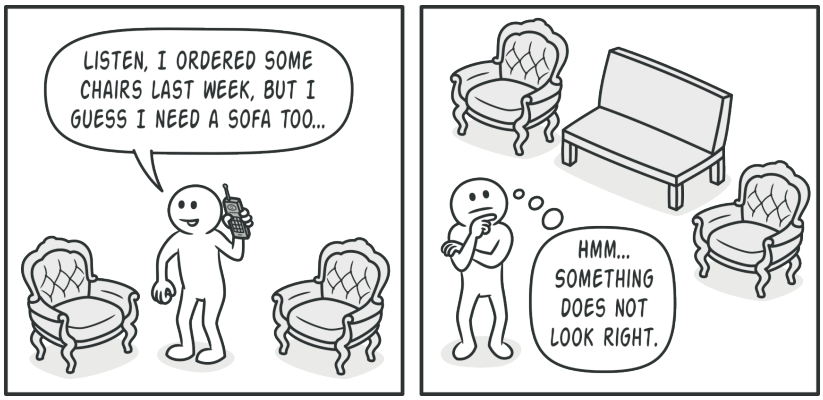
1. A family of related products, say: `Chair` + `Sofa` + `CoffeeTable`.

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.

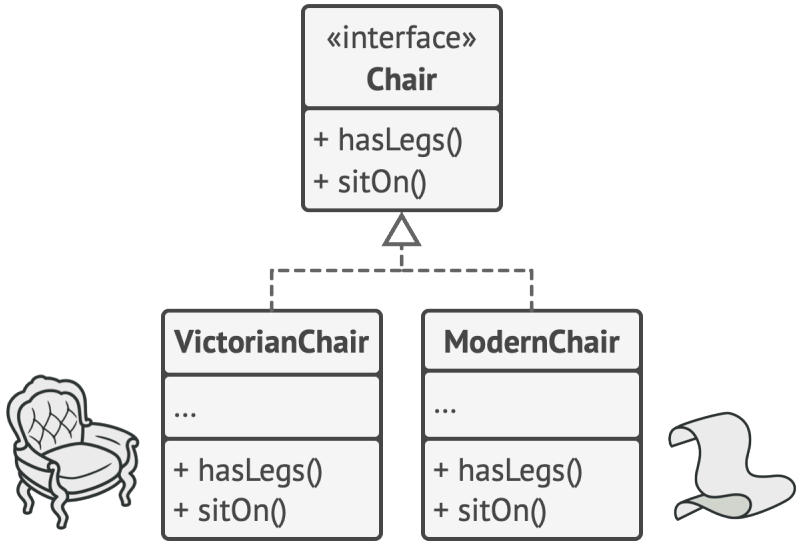


A Modern-style sofa doesn't match Victorian-style chairs.

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

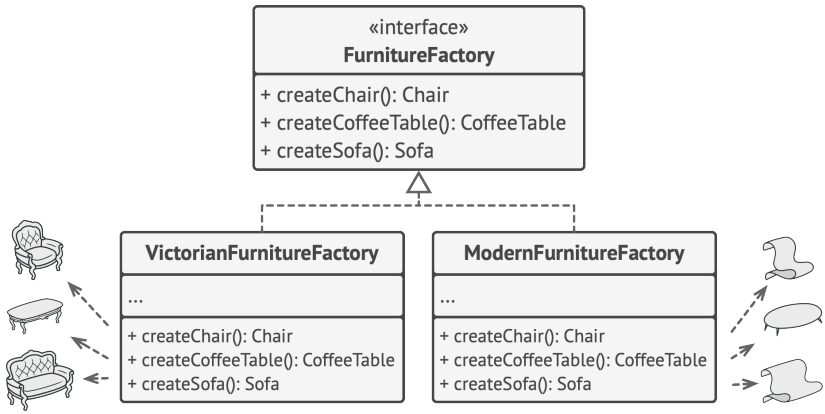
The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the `Chair`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

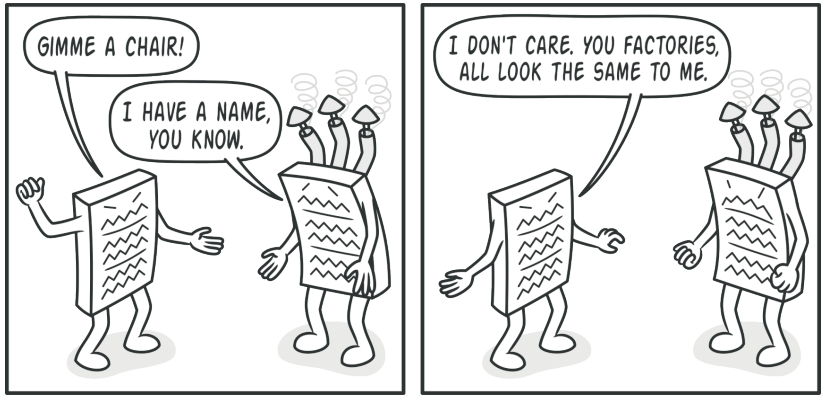
The next move is to declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (for example, `createChair`, `createSofa` and `createCoffeeTable`). These methods must return **abstract** product types represented by the interfaces we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

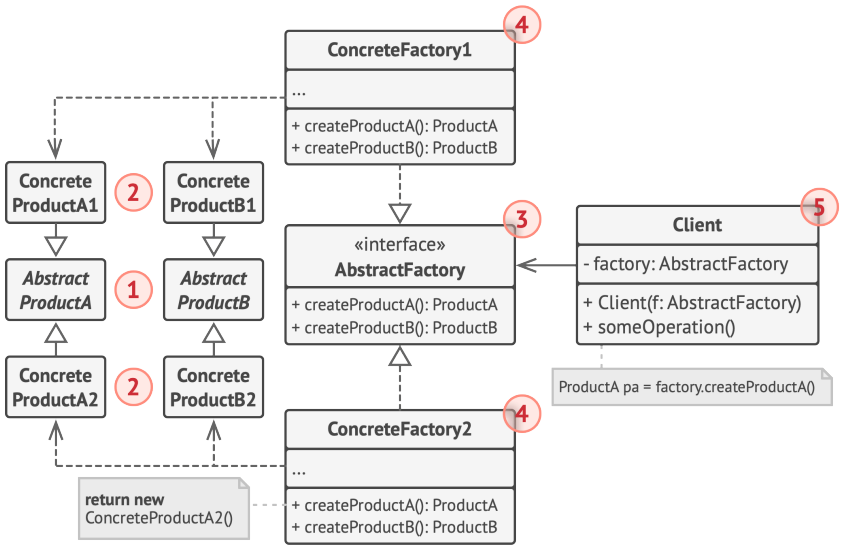


The client shouldn't care about the concrete class of the factory it works with.

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).

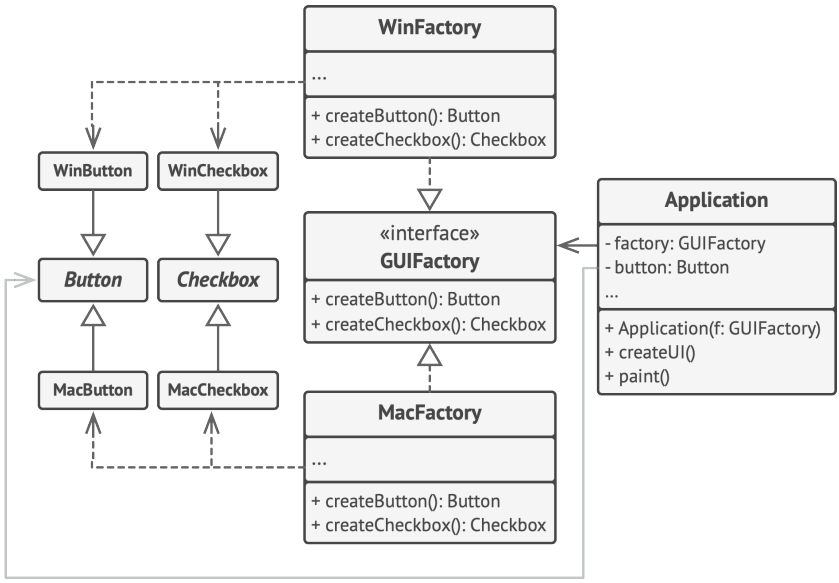
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.

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 factory: GUIFactory
  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 stand-alone 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

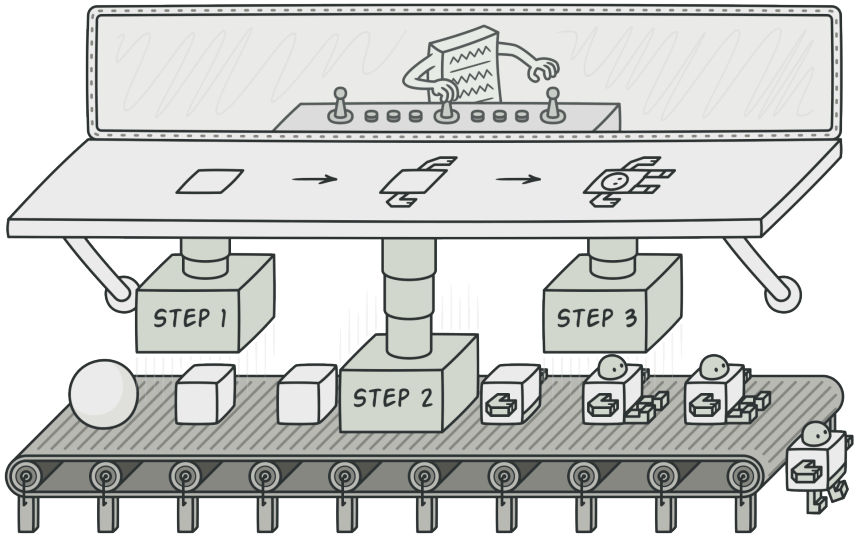
objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.

- **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**.

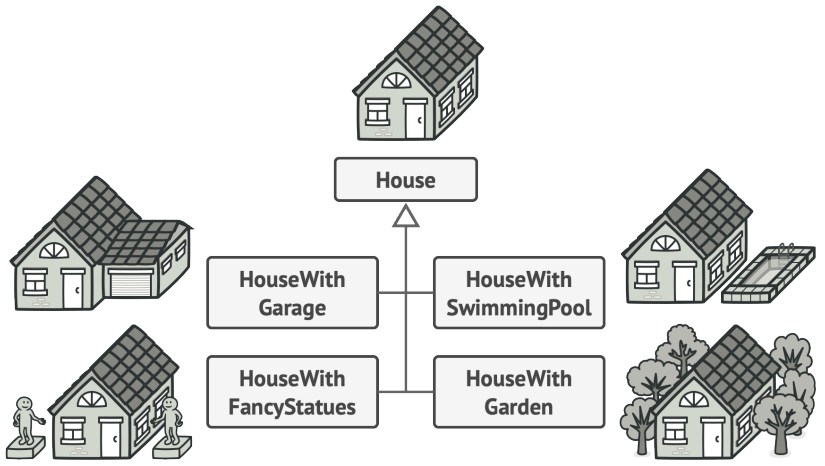


BUILDER

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.

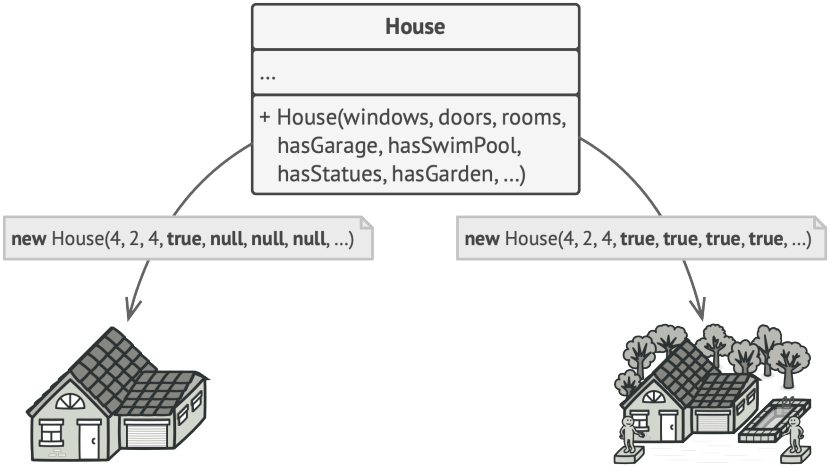


You might make the program too complex by creating a subclass for every possible configuration of an object.

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.

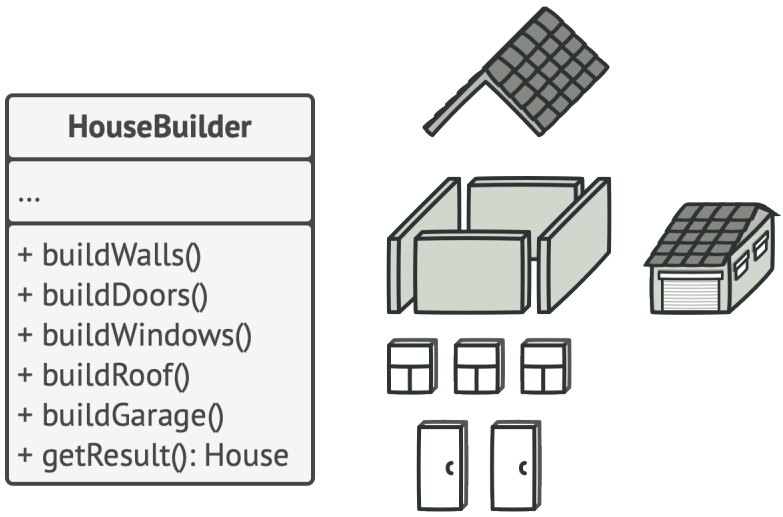


The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

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*.

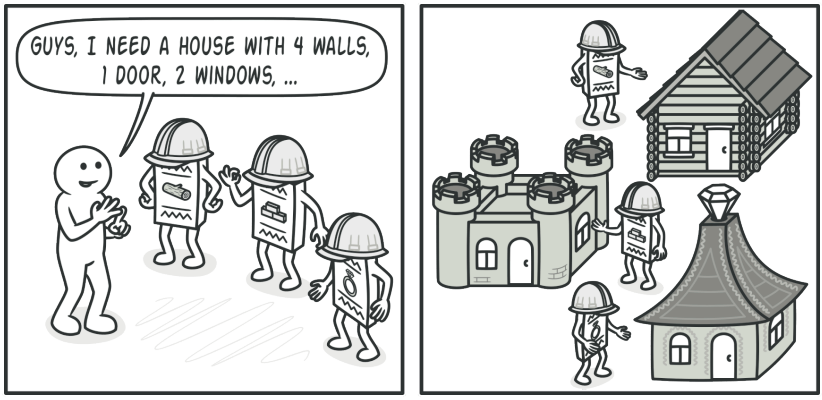


The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

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.



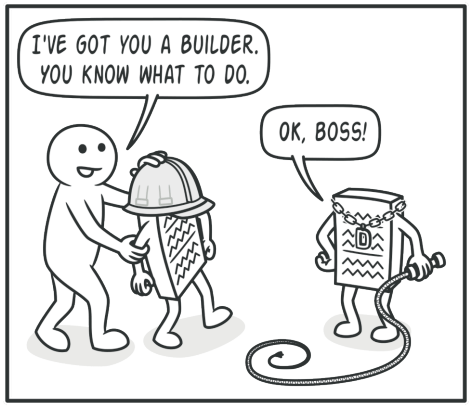
Different builders execute the same task in various ways.

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.

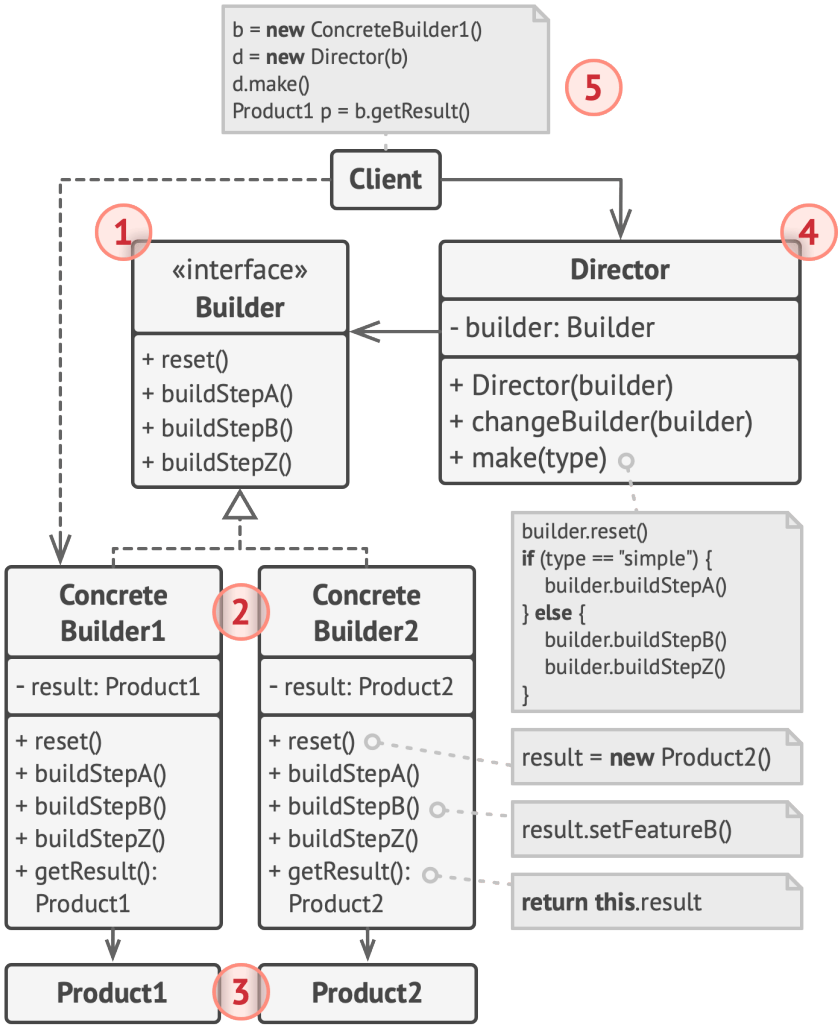


The director knows which building steps to execute to get a working product.

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.

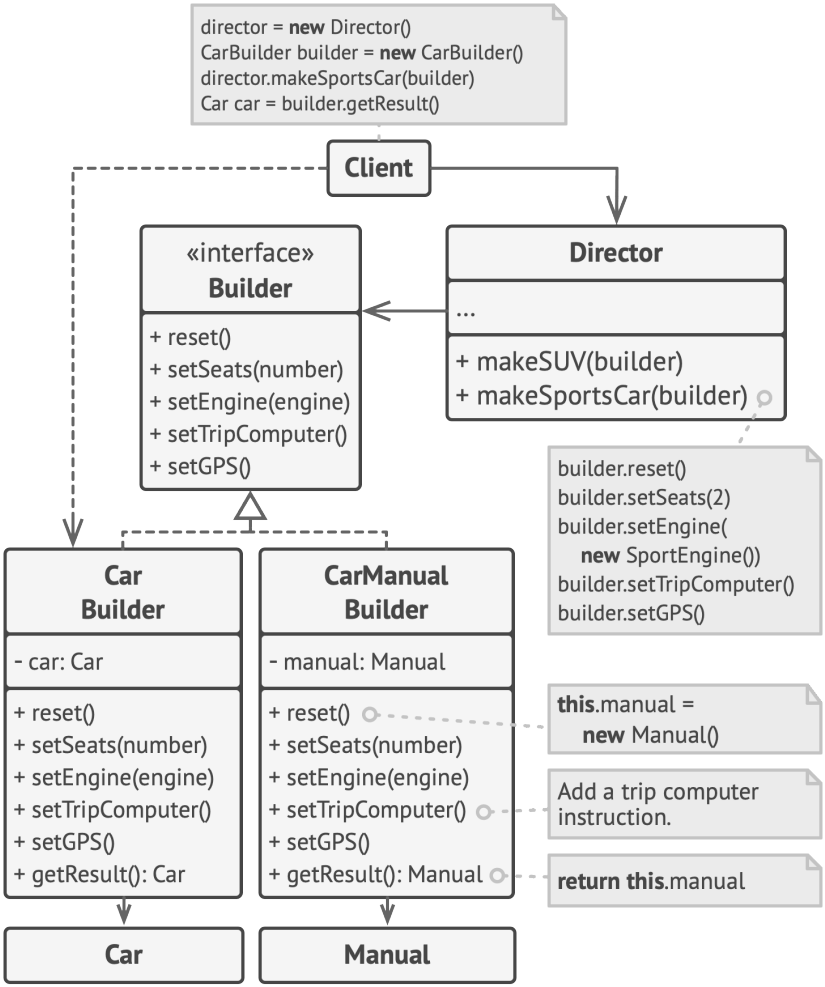
4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

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


```

```
    // A fresh builder instance should contain a blank product
    // object which it uses in further assembly.
    constructor CarBuilder() is
      this.reset()

    // The reset method clears the object being built.
    method reset() is
      this.car = new Car()

    // All production steps work with the same product instance.
    method setSeats(...) is
      // Set the number of seats in the car.

    method setEngine(...) is
      // Install a given engine.

    method setTripComputer(...) is
      // Install a trip computer.

    method setGPS(...) is
      // Install a global positioning system.

    // Concrete builders are supposed to provide their own
    // methods for retrieving results. That's because various
    // types of builders may create entirely different products
    // that don't all follow the same interface. Therefore such
    // methods can't be declared in the builder interface (at
    // least not in a statically-typed programming language).
    //
    // Usually, after returning the end result to the client, a
    // builder instance is expected to be ready to start
    // 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
// products that don't follow the 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
  // 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.
  // 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 “telescoping 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 class constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed to a specific product 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

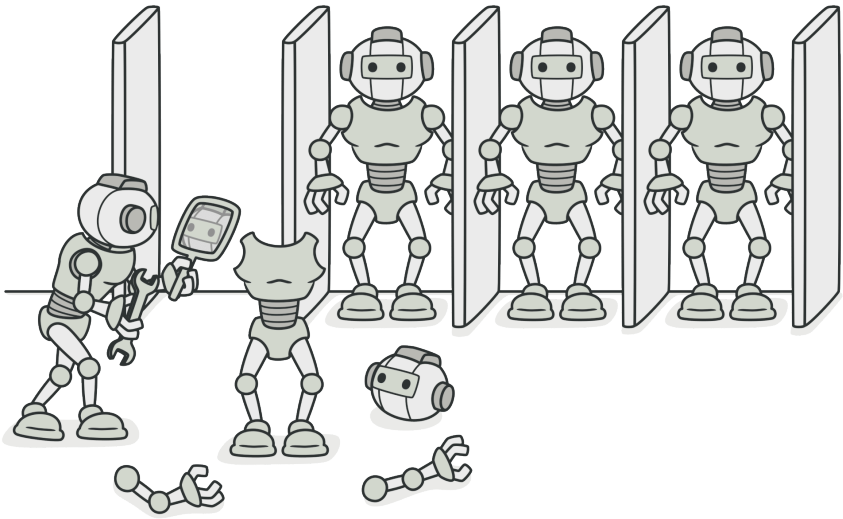
- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **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

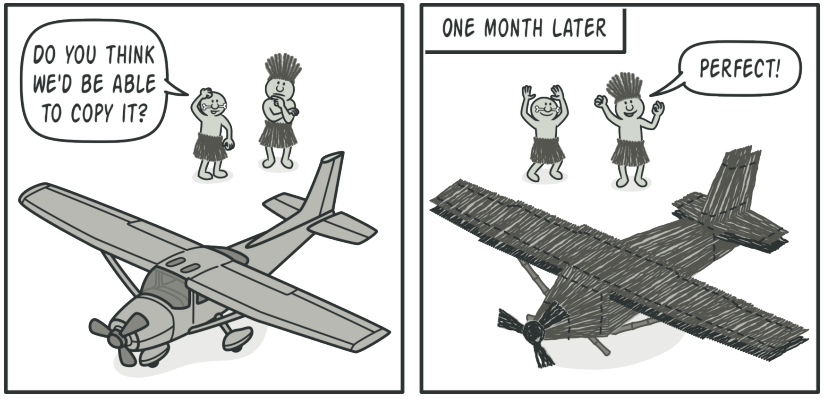
Also known as: Clone

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.*

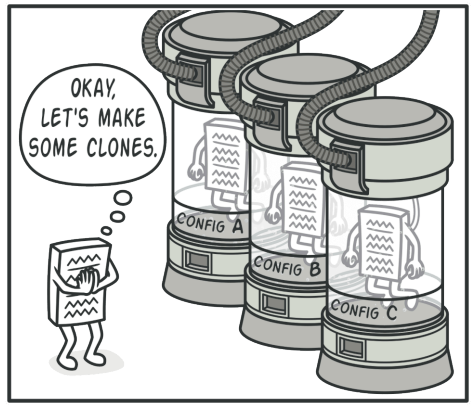
There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete

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.



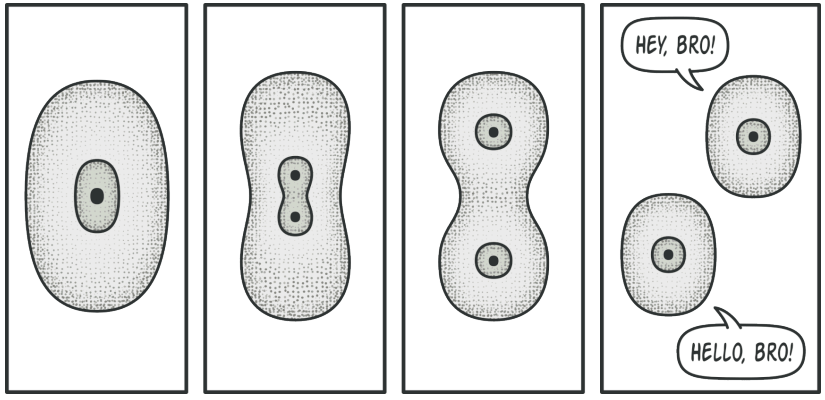
Pre-built prototypes can be an alternative to subclassing.

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.

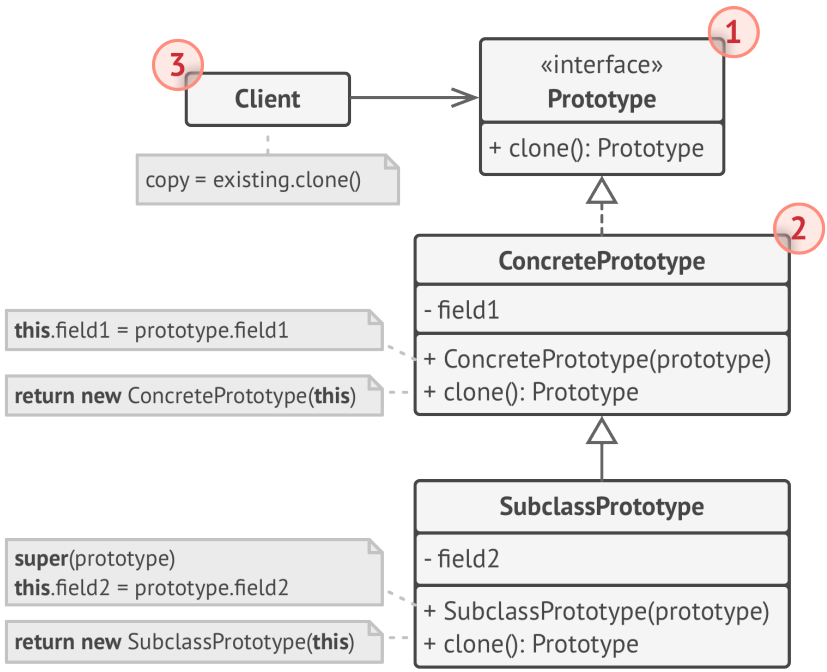


The division of a cell.

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

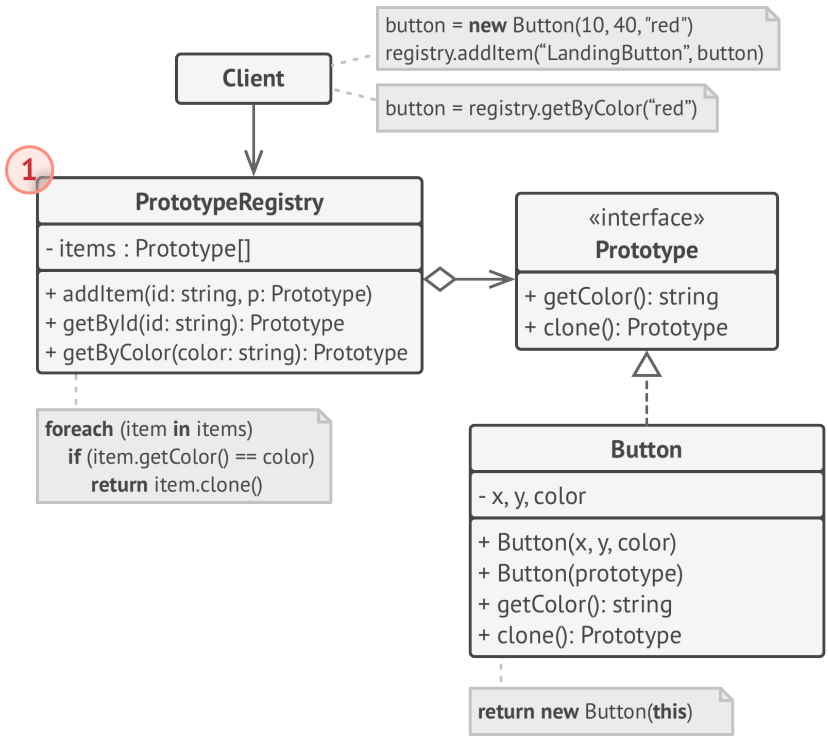


1. The **Prototype** interface declares the cloning methods. In most cases, it's a single `clone` method.

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.

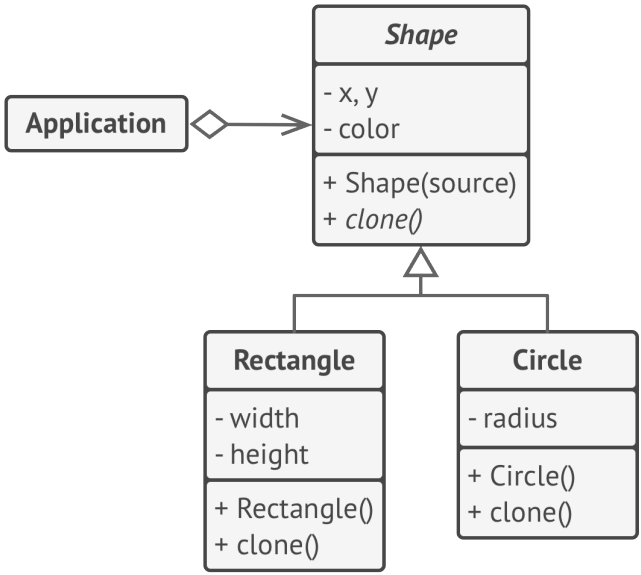
Prototype registry implementation



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
// in one go by calling the constructor of the current class and
// passing the current object as the constructor's argument.
// Performing all the actual copying in the constructor helps to
// keep the result consistent: the constructor will not return a
// result until the new object is fully built; thus, no object
// can have a reference to a partially-built clone.
```

```
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.

Suppose you have a complex class that requires a laborious configuration before it can be used. There are several common ways to configure this class, and this code is scattered through your app. To reduce the duplication, you create several subclasses and put every common configuration code into their constructors. You solved the duplication problem, but now you have lots of dummy subclasses.

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

ing 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 won't be able to create a separate "prototype" constructor. Thus, copying the object's data into the newly created clone will have to be performed within the `clone` method. Still, having this code in a regular constructor is safer because the resulting object is returned fully configured 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` oper-

ator. 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.

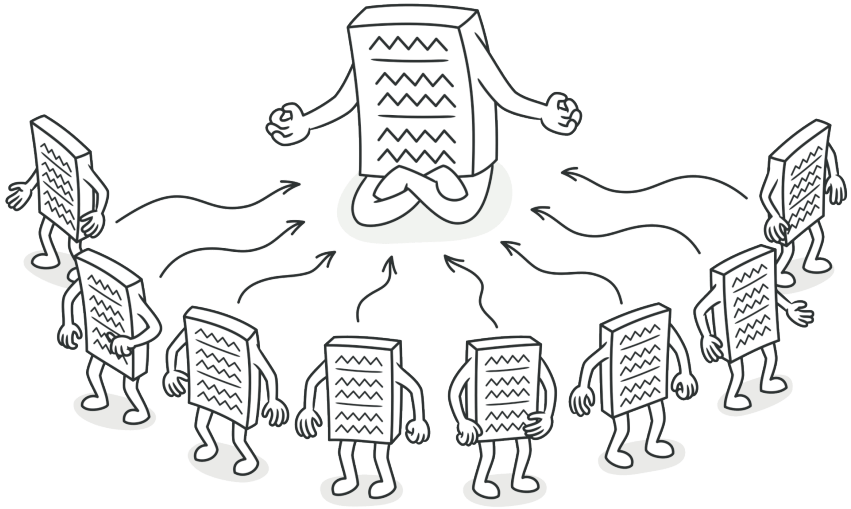
✓ You can produce complex objects more conveniently.

✓ 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.

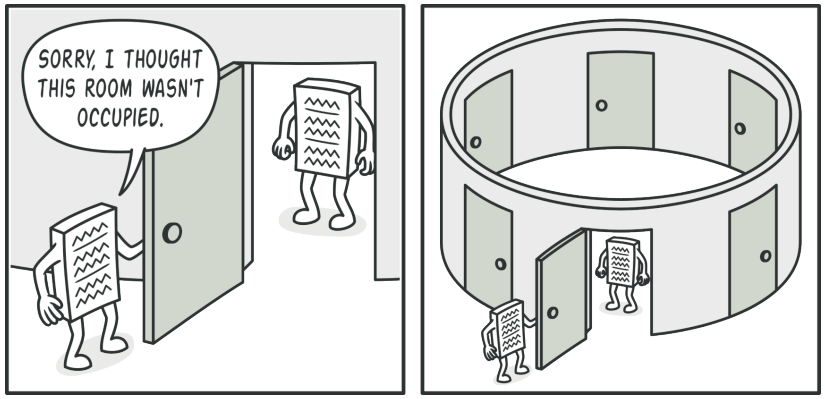
☹ Problem

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

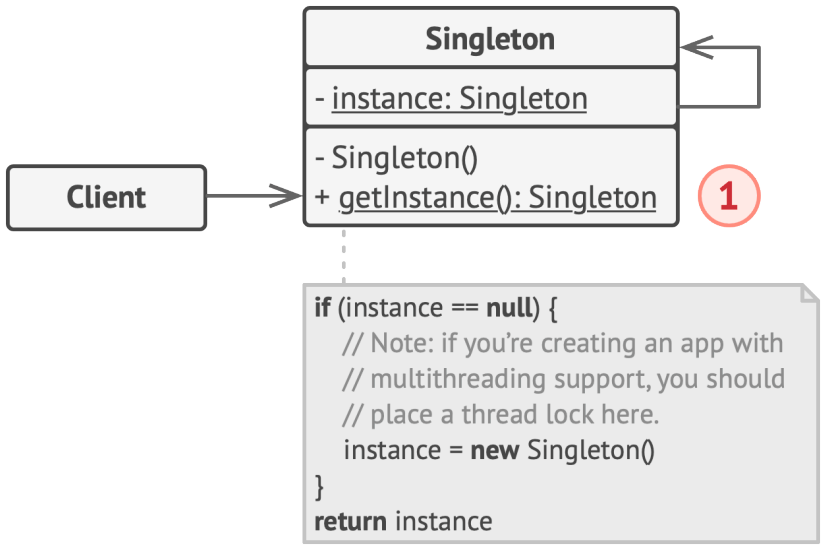
create an object and saves it in a static field. All following calls to this method return the cached object.

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 (Database.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 (Database.instance == null) then
            Database.instance = new Database()
      return Database.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
    // 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**.