Design Patterns

Lecturer: Adel Vahdati

General Categories

- **Creational** patterns
  - Deal with initializing and configuring classes and objects.
- **Structural** patterns
  - Deal with decoupling interface and implementation of classes and objects.
- **Behavioral** patterns
  - Deal with dynamic interactions among societies of classes and objects.

GoF Design Patterns: Purpose and Scope

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Creational Pattern

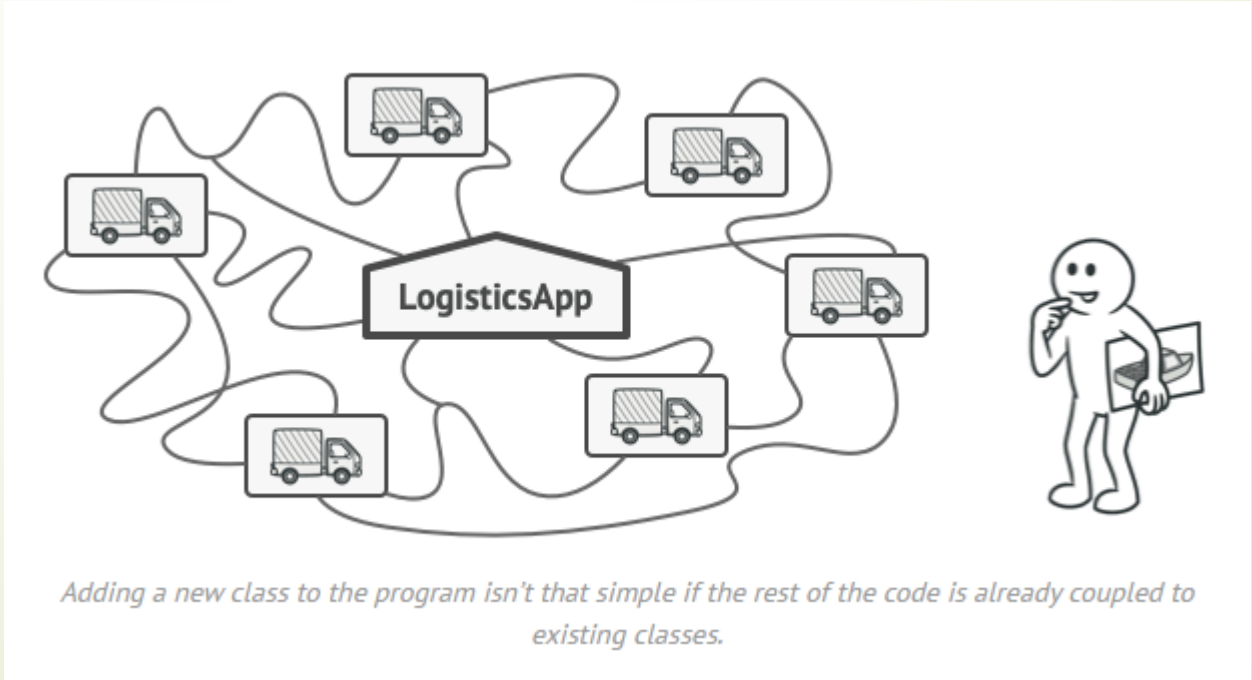
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Factory Method

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

Problem

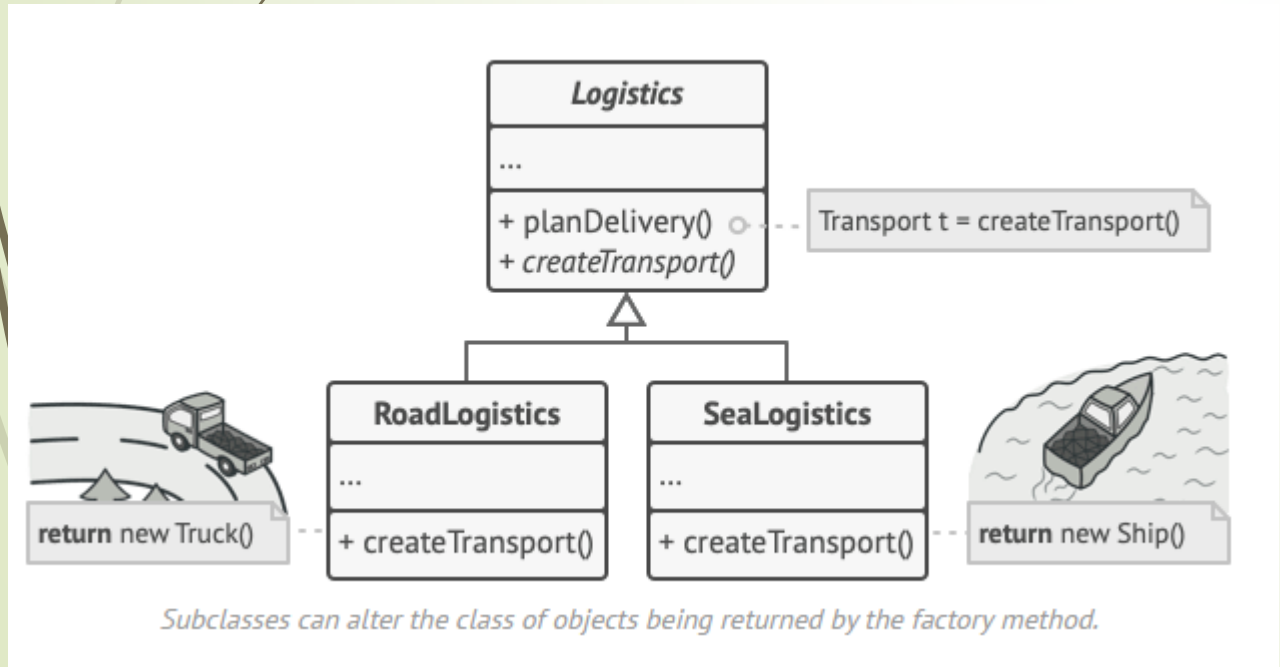
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase.



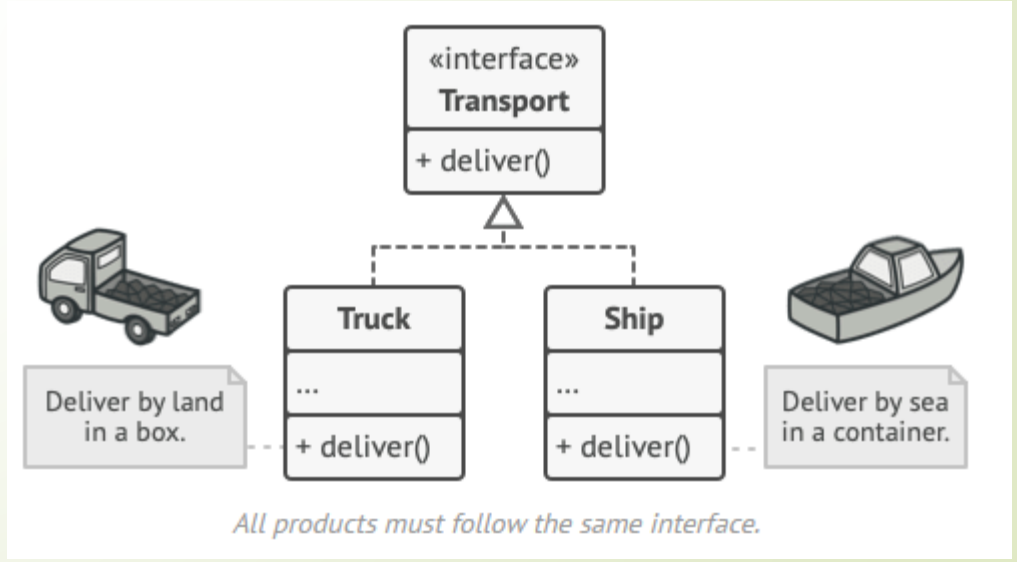
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.



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



All products must follow the same interface.

Abstract Factory

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

Problem

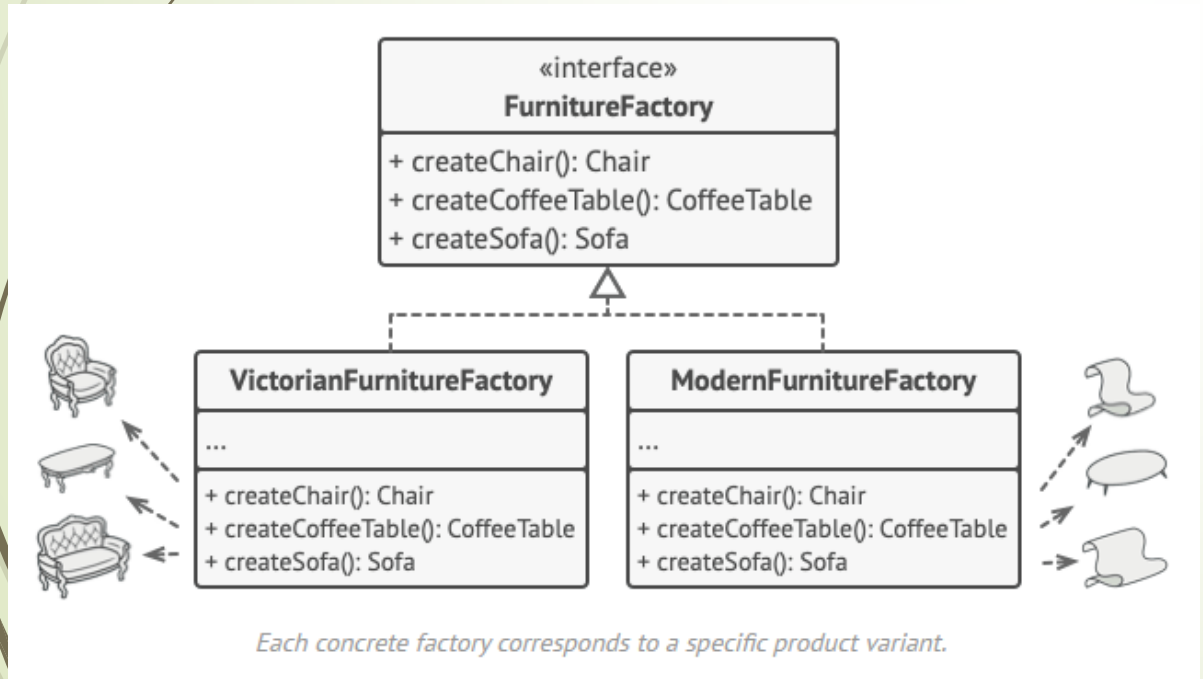
- Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
- A family of related products, say: Chair + Sofa + Coffee Table.
- Several variants of this family. For example, products Chair + Sofa + Coffee Table are available in these variants: Modern, Victorian, ArtDeco.
- You need a way to create individual furniture objects so that they match other objects of the same family.

Problem

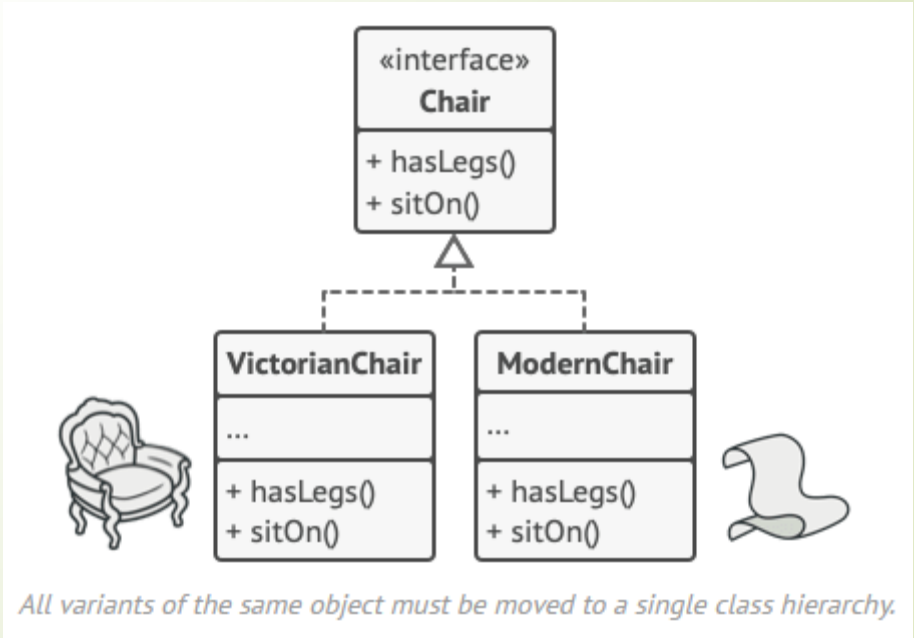
Product families and their variants.

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.



Each concrete factory corresponds to a specific product variant.



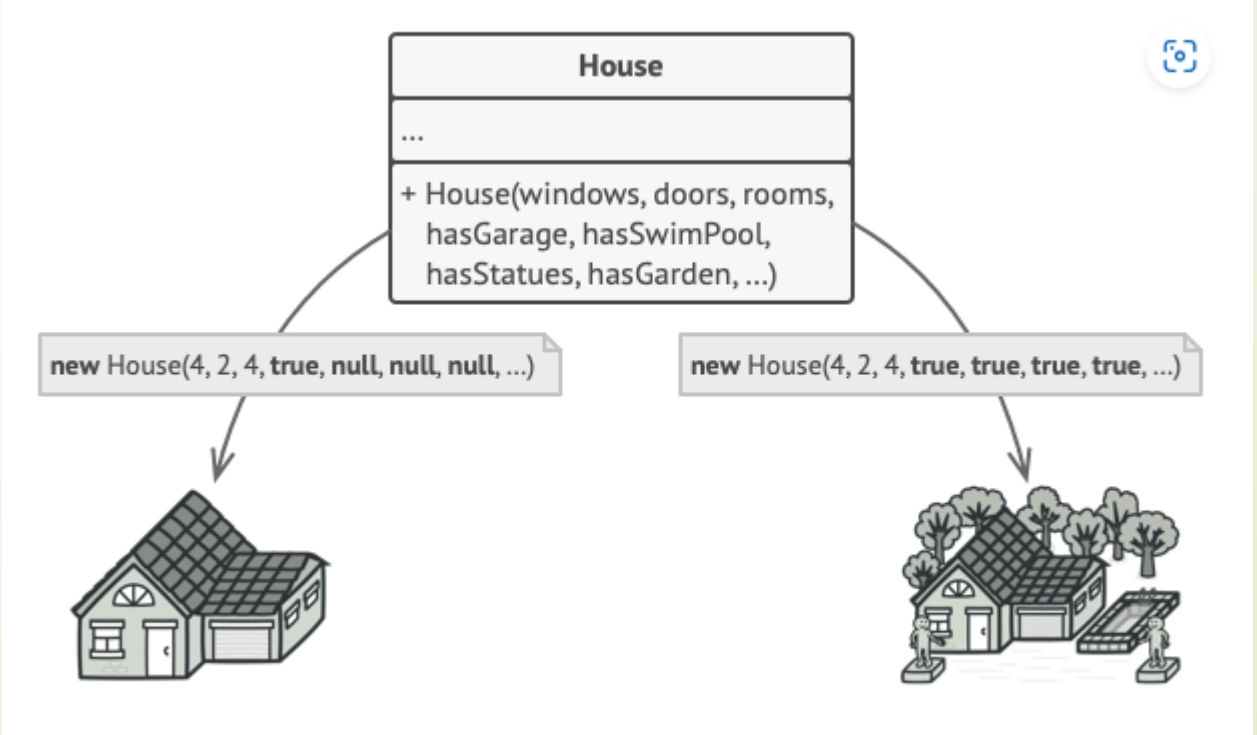
All variants of the same object must be moved to a single class hierarchy.

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

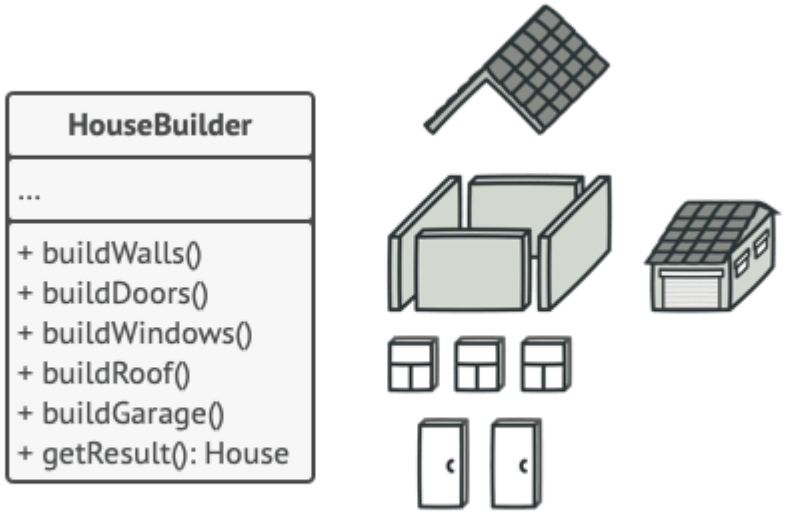
- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.



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

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.

Prototype

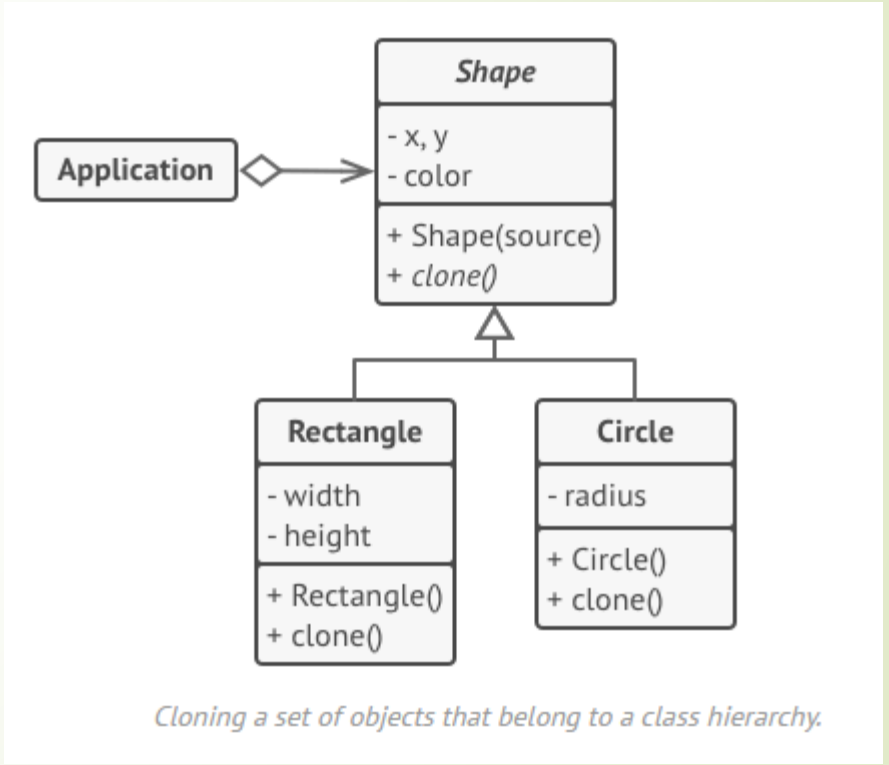
- **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
-

Problem

- You have an object, and you want to create an exact copy of it.
- 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.
- 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.
- Since you have to know the object's class to create a duplicate, your code becomes dependent on that class.
- Sometimes you only know the interface that the object follows, but not its concrete class

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.

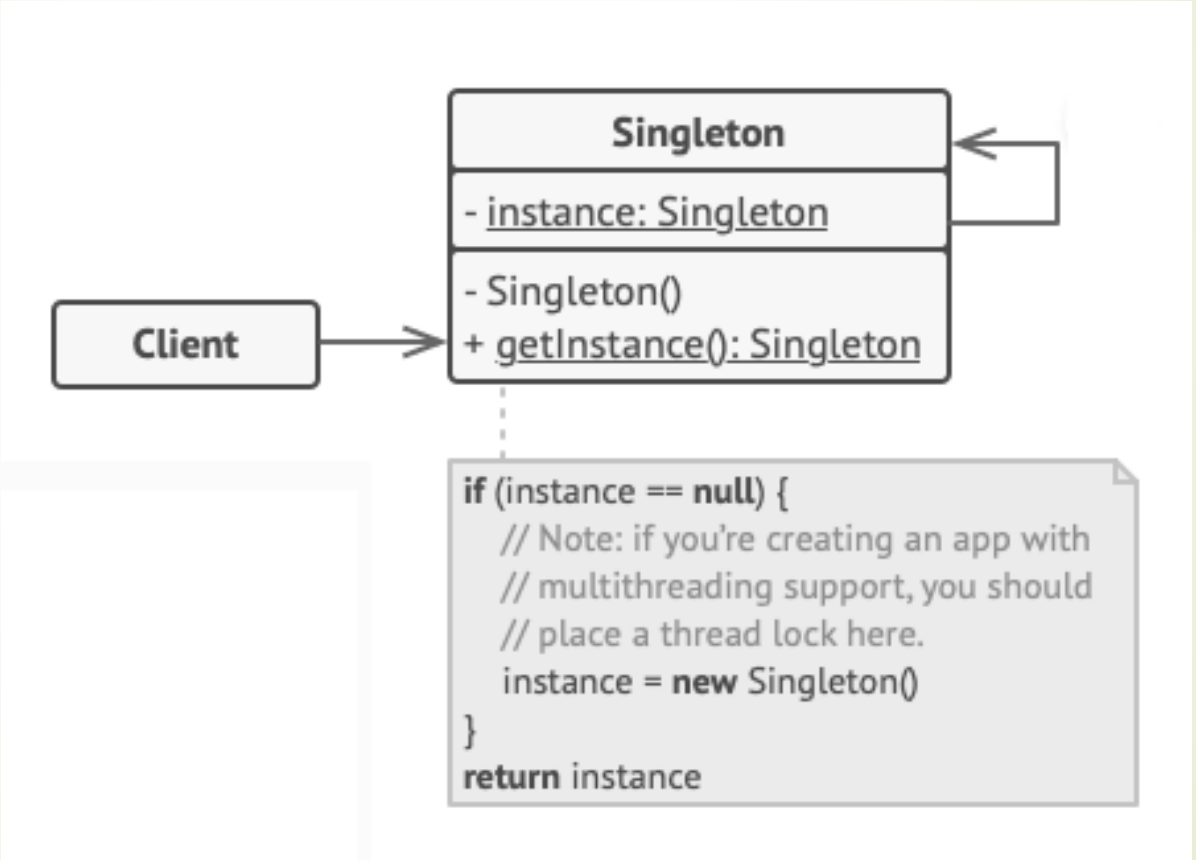


Cloning a set of objects that belong to a class hierarchy.

Singleton

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

Singleton


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
if (instance == null) {
// Note: if you're creating an app with
// multithreading support, you should
// place a thread lock here.
instance = new Singleton()
}
return instance

Structural Pattern

- **Adapter:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge:** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite:** Compose objects into tree structures to represent whole part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator:** Attach additional responsibilities to an object dynamically.
- **Façade:** Provide a unified interface to a set of interfaces in a subsystem.
- **Flyweight:** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy:** Provide a surrogate or placeholder for another object control access to it.

Adapter

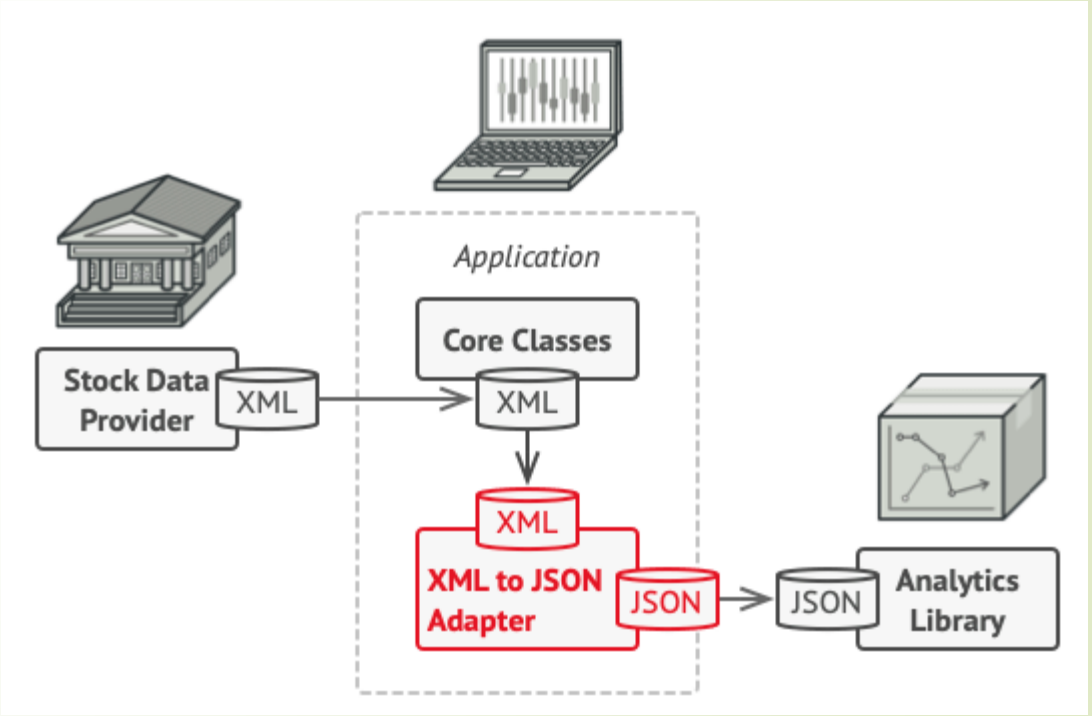
- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.

Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.



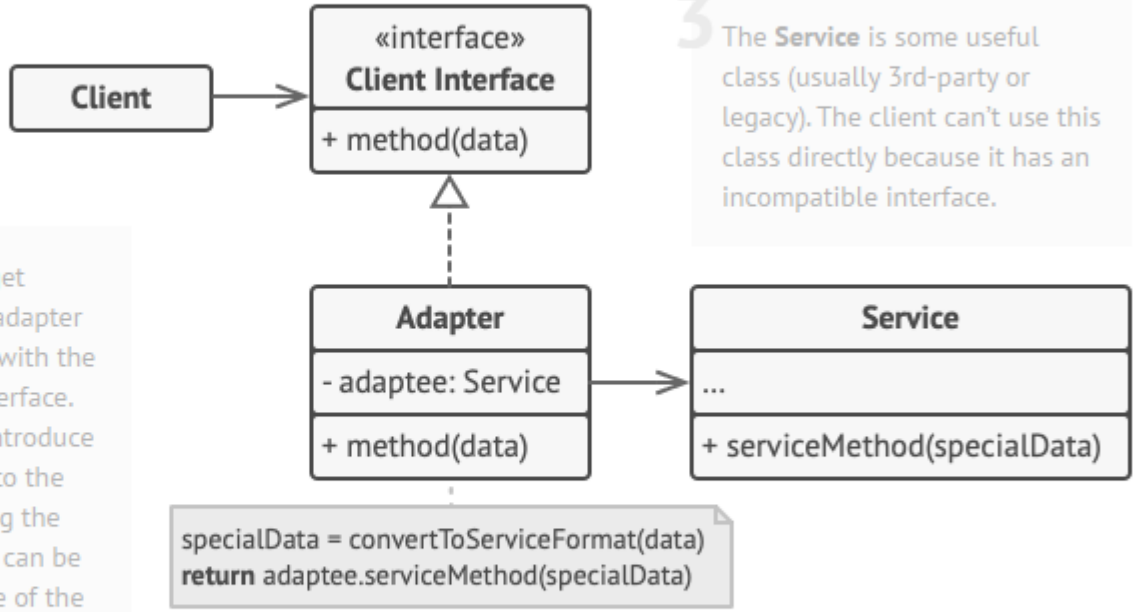
Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
  - The adapter gets an interface, compatible with one of the existing objects.
  - Using this interface, the existing object can safely call the adapter's methods.
  - Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Solution

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

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



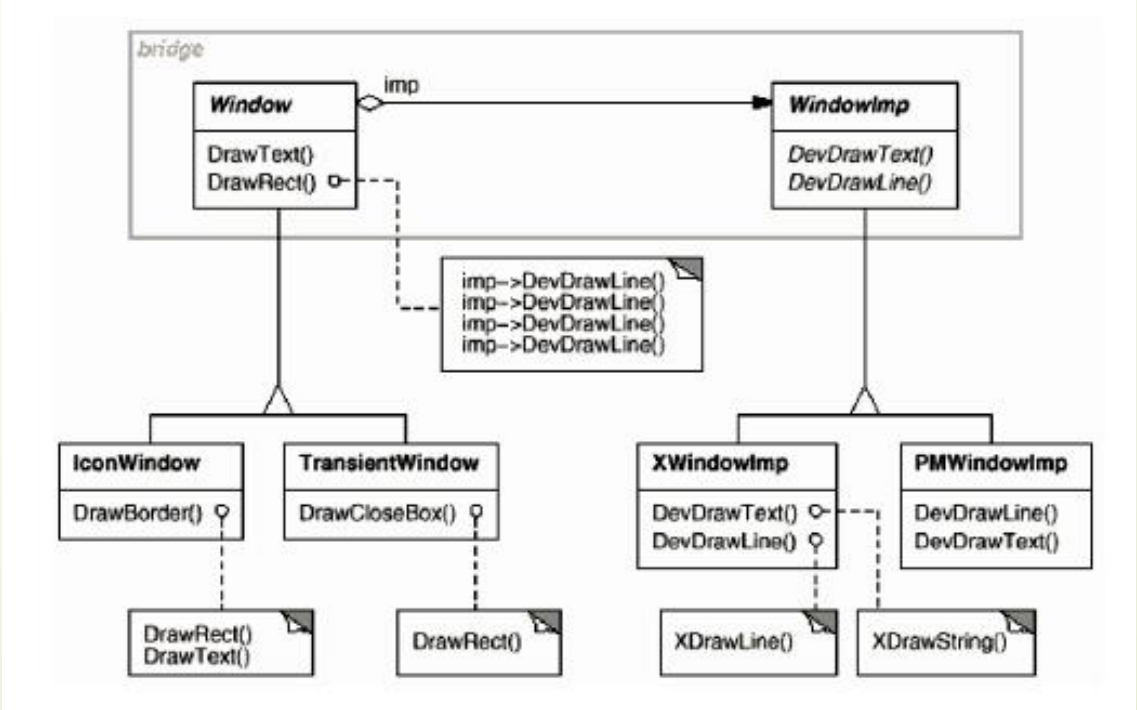
3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

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

Bridge

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



Bridge: Applicability

- you want to avoid a permanent binding between an abstraction and its implementation; for example, when the implementation must be selected or switched at run-time.
- both the abstractions and their implementations should be extensible by sub-classing; combine different abstractions and implementations and extend them independently.
- you want to share an implementation among multiple objects and this fact should be hidden from the client.

Composite

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

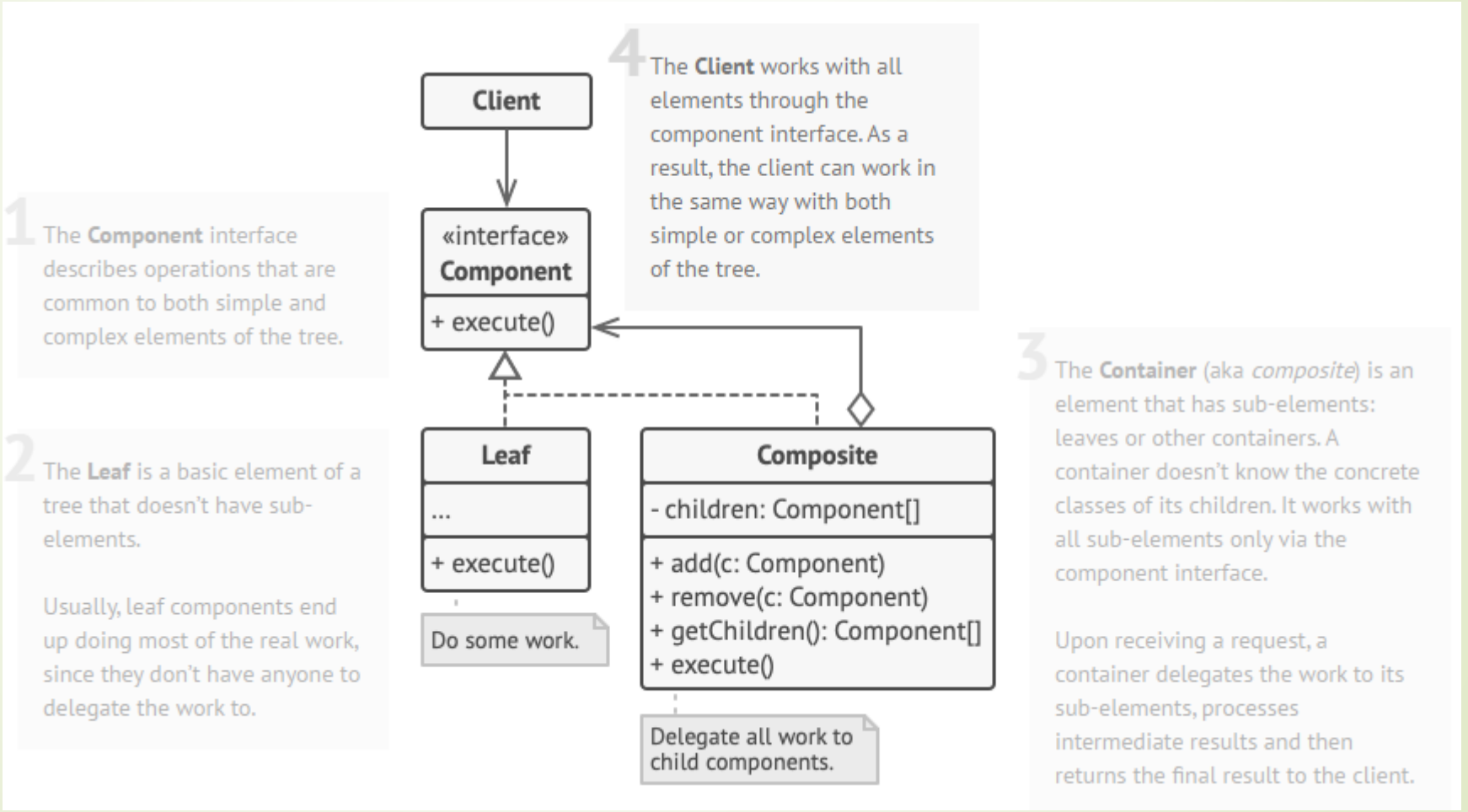
Problem

- imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order

Solution

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

Solution


1 The Component interface describes operations that are common to both simple and complex elements of the tree.
2 The Leaf is a basic element of a tree that doesn't have sub-elements.
Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.
Client
«interface»
Component
+ execute()
Leaf
...
+ execute()
Do some work.
Composite
- children: Component[]
+ add(c: Component)
+ remove(c: Component)
+ getChildren(): Component[]
+ execute()
Delegate all work to child components.
4 The Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.
3 The Container (aka composite) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.
Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

Behavioral Patterns

- **Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator:** Define an object that encapsulates how a set of objects interact; promotes loose coupling by keeping objects from referring to each other explicitly.

Behavioral Patterns (2)

- **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable; lets the algorithm vary independently from clients that use it
- Visitor: Represent an operation to be performed on the elements of an object structure; lets you define a new operation without changing the classes of the elements.