



Design Patterns

Lecturer: Adel Vahdati



General Categories

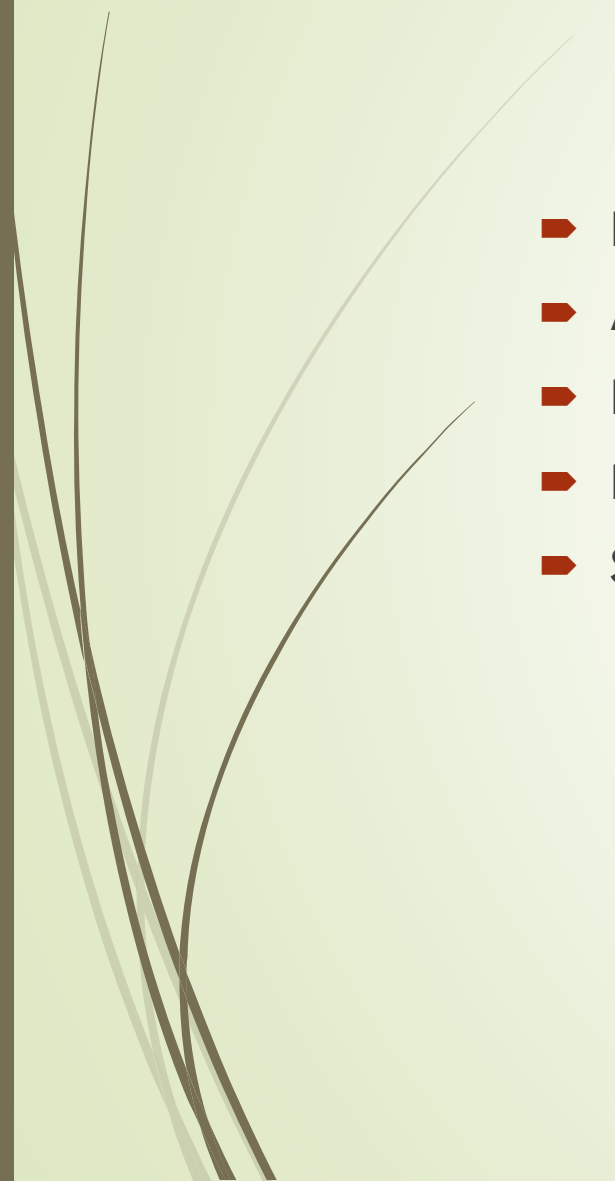
- **Creational** patterns
 - Deal with initializing and configuring classes and objects.
- **Structural** patterns
 - Deal with decoupling interface and implementation of classes and objects.
- **Behavioral** patterns
 - Deal with dynamic interactions among societies of classes and objects.

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

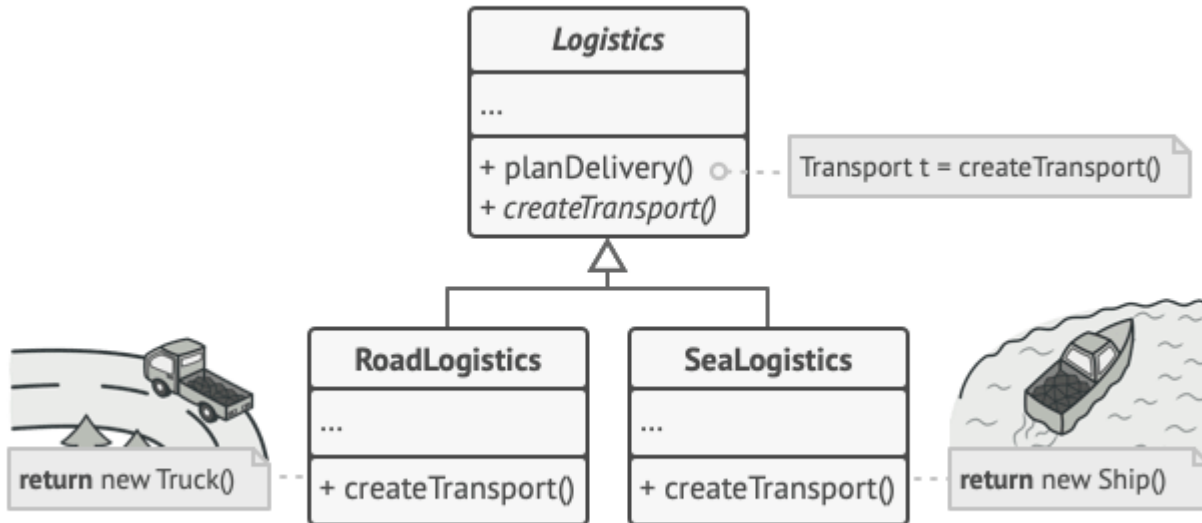
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making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



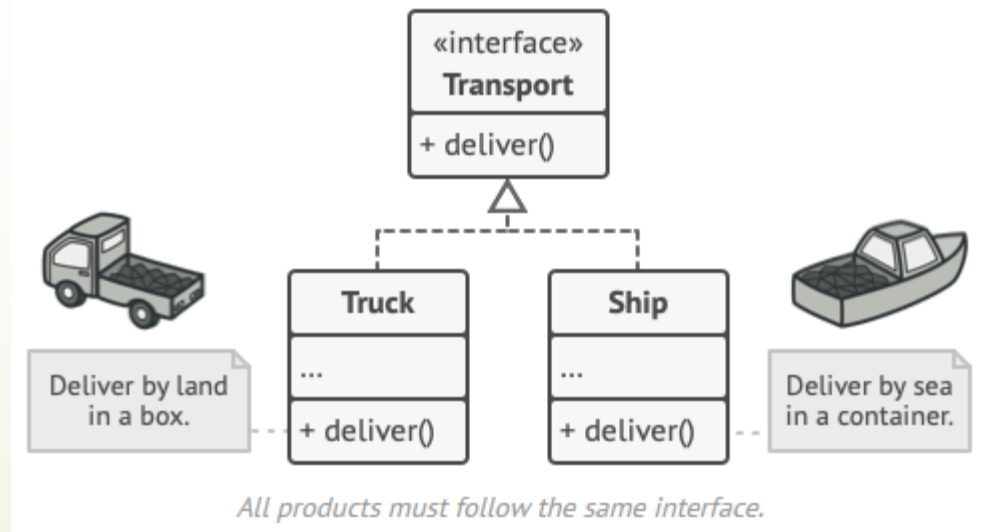
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

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.

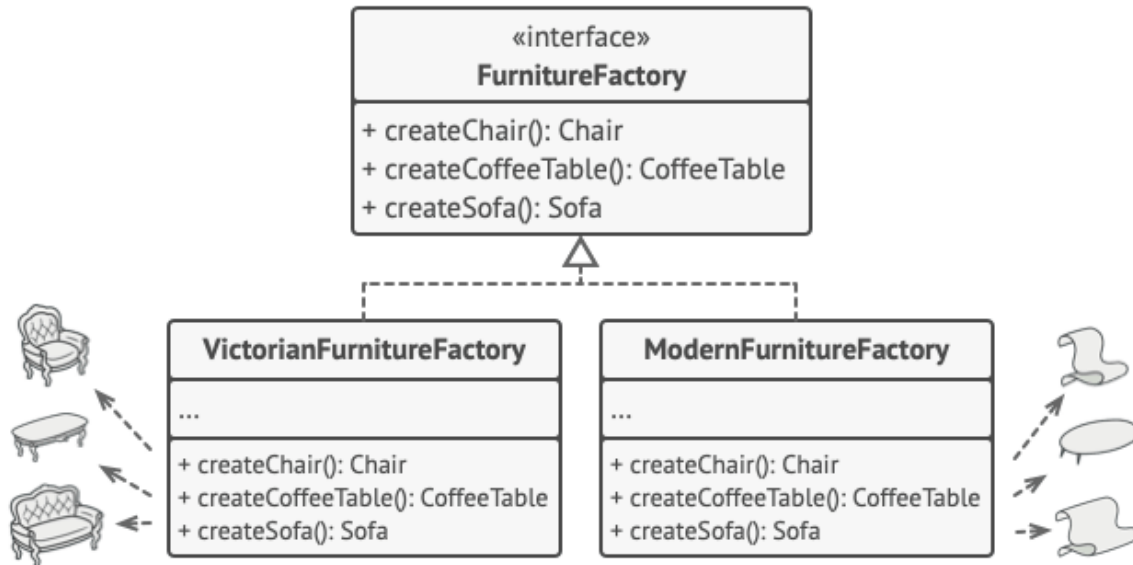
Problem

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

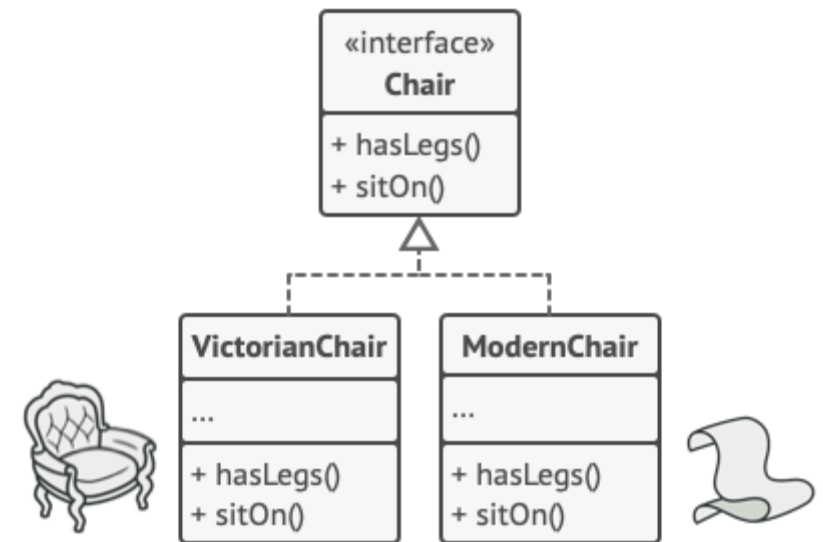
Product families and their variants.

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.

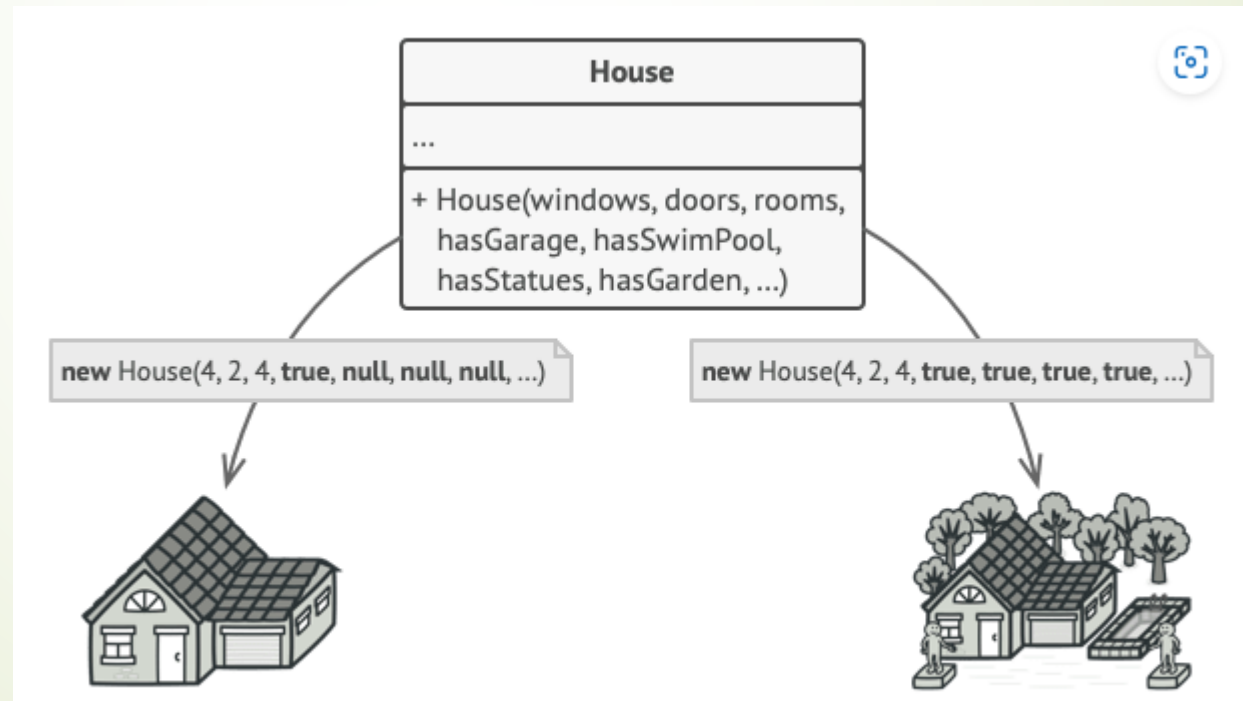


Builder

- 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

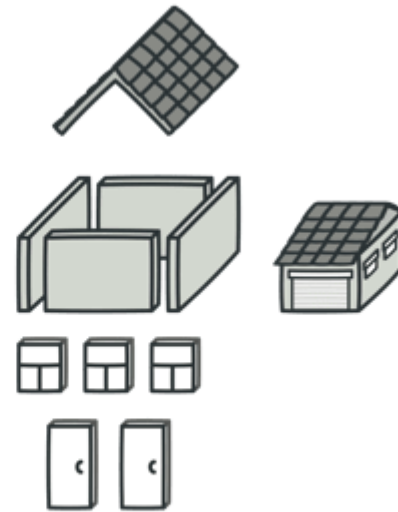
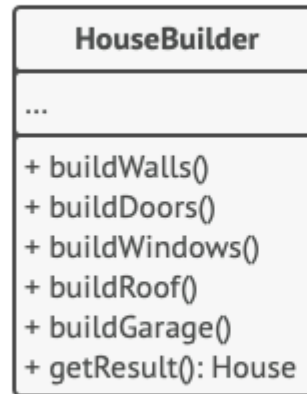
- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.



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.



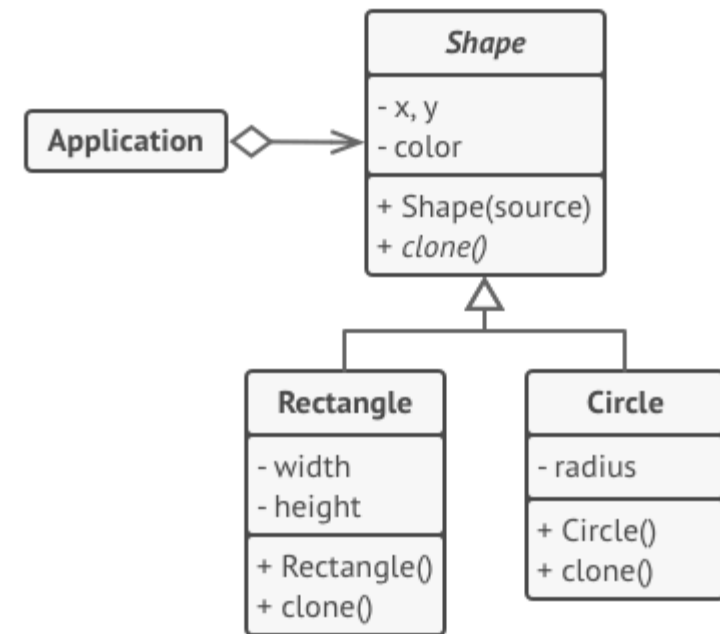


Problem

- 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.

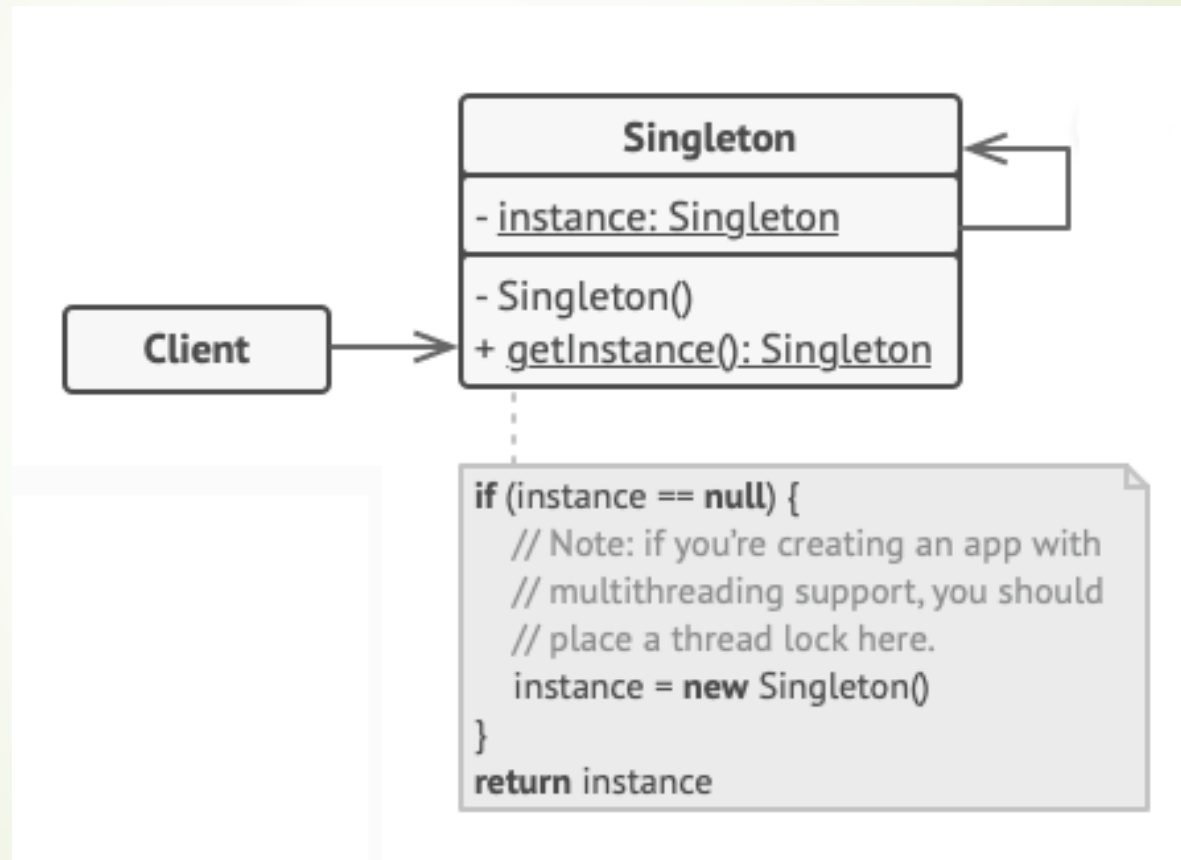


Singleton



- 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





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.



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.