DESIGN PATTERN: CREATIONAL PATTERNS

ITX 2001, CSX 3002, IT 2371

DESIGN PATTERNS

- They are typical solutions to commonly occurring problems in software design. [1]
- They represent the best practices used by experienced O-O software developers. [2]
- They are solutions commonly solved typical problems faced during developing program.

GANG OF FOUR (GOF)

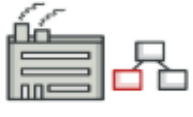
- Four authors – Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,
- published a book named “Design Patterns –Elements of Reusable Object-Oriented Software)
- initiated the concept of “Design Pattern in Software Development”

S.N.	Pattern & Description [2]
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

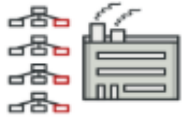
The Catalog of Design Patterns [1]

Creational patterns

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

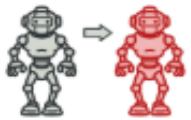


Factory Method



Abstract Factory

Builder

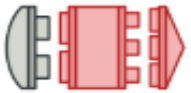


Prototype

Singleton

Structural patterns

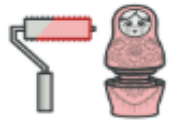
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter

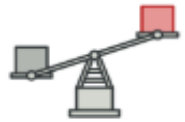
Bridge

Composite



Decorator

Facade



Flyweight

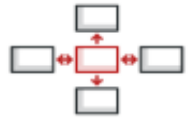
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

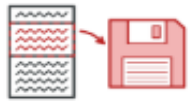
Chain of Responsibility

Command

Iterator

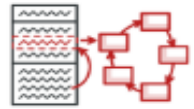


Mediator



Memento

Observer



State

Strategy

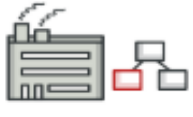
Template Method

Visitor

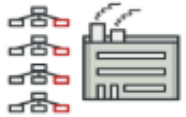
The Catalog of Design Patterns [1]

Creational patterns

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

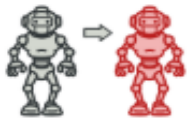


Factory Method



Abstract Factory

Builder

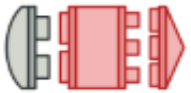


Prototype

Singleton

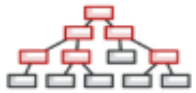
Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

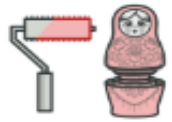


Adapter

Bridge

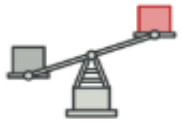


Composite



Decorator

Facade



Flyweight

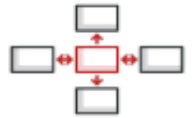
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Chain of Responsibility

Command

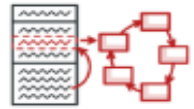
Iterator



Mediator

Memento

Observer



State

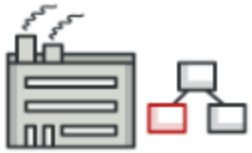
Strategy

Template Method

Visitor

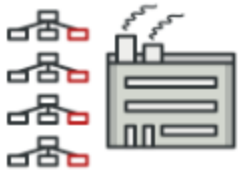
Creational Design Patterns [1]

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



Factory Method

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

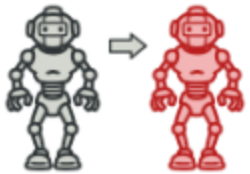


Abstract Factory

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

Builder

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



Prototype

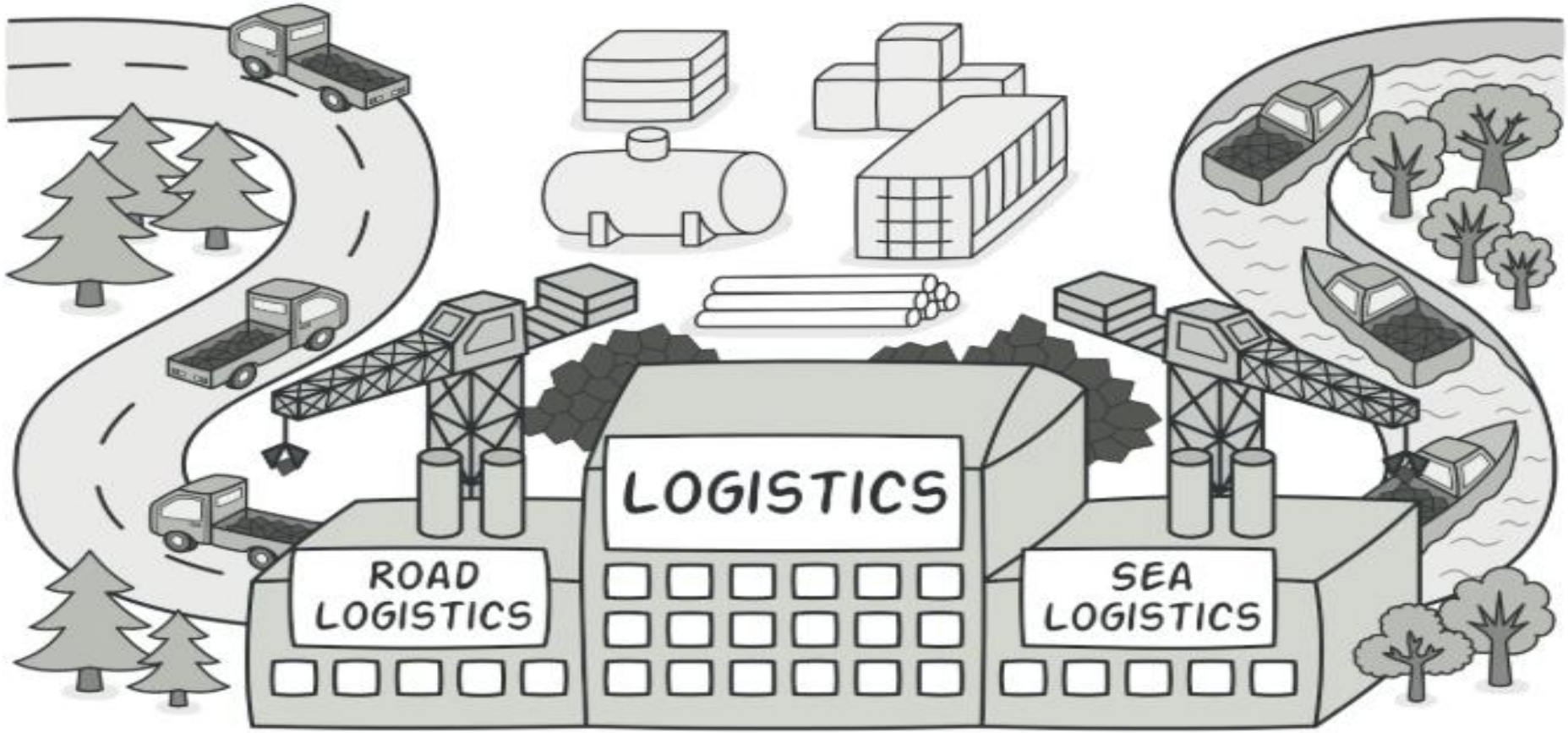
Lets you copy existing objects without making your code dependent on their classes.

Singleton

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

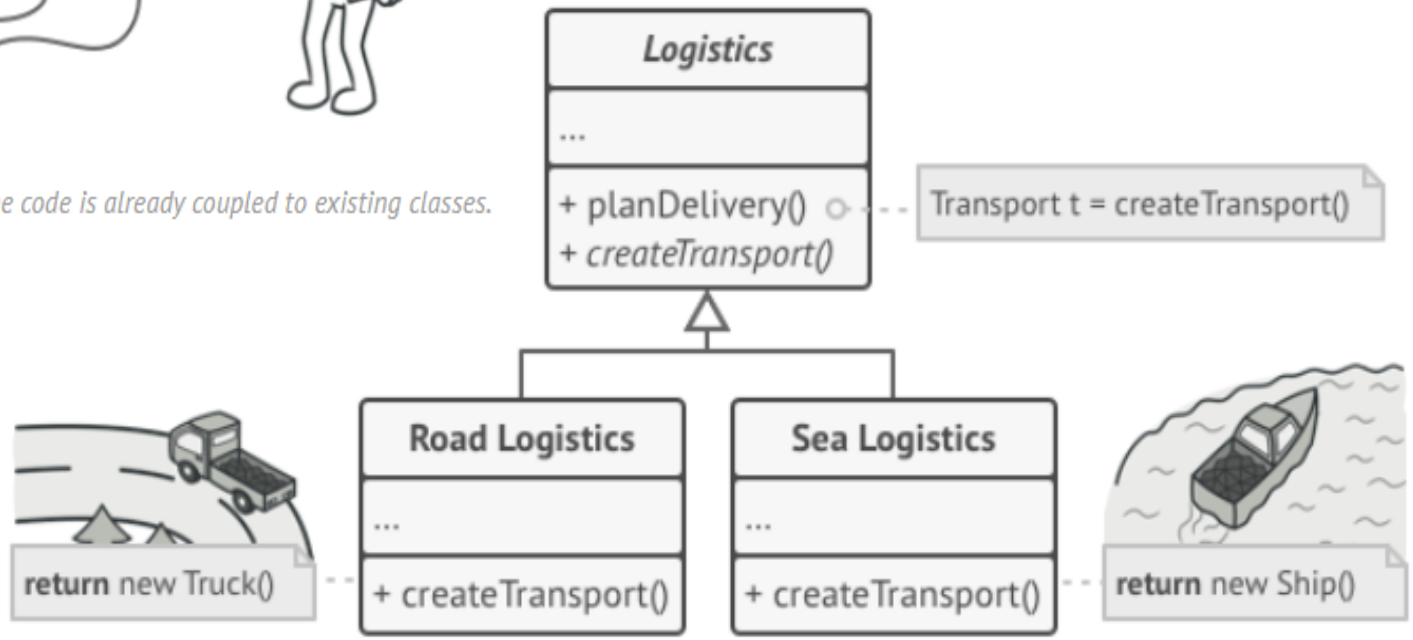
FACTORY METHOD [1]

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.



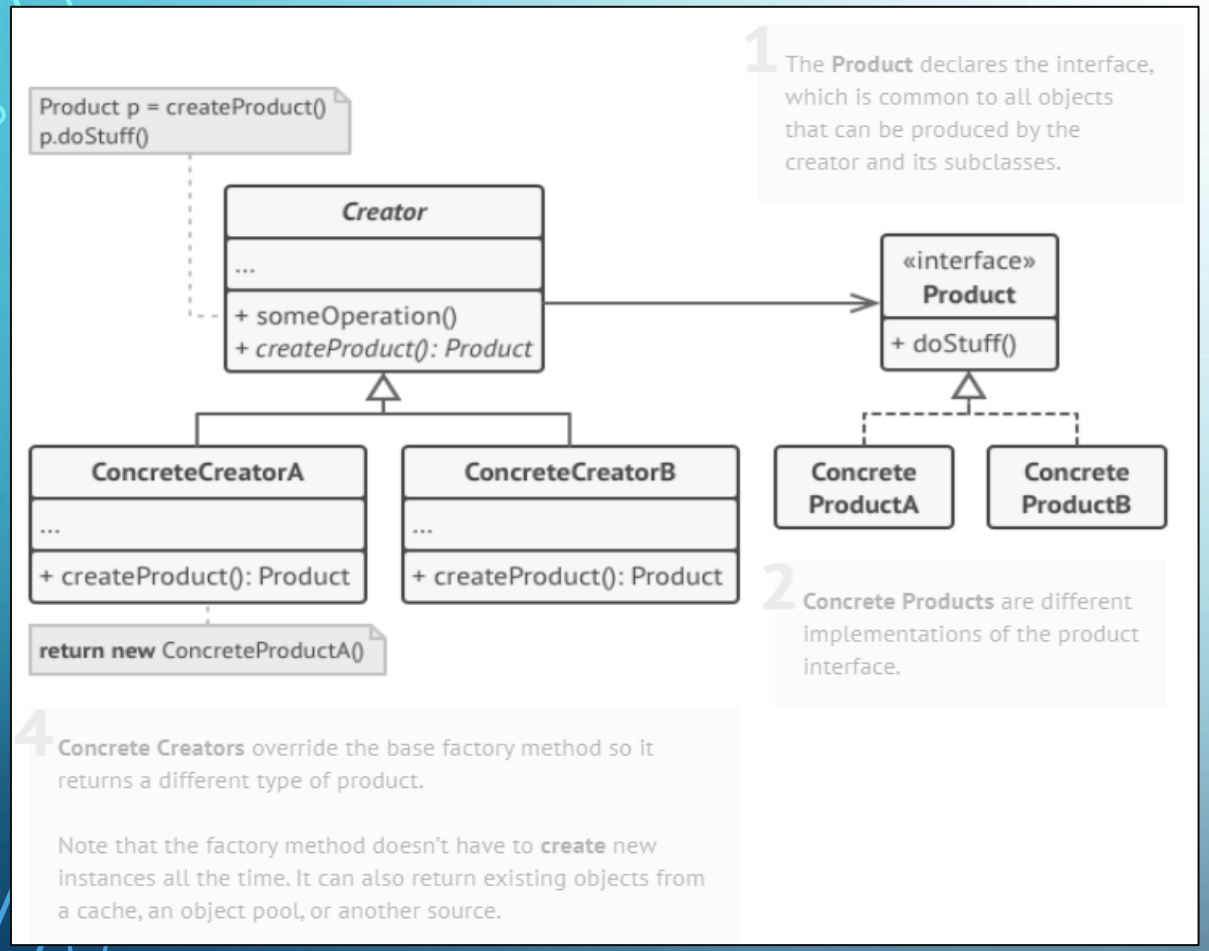
FACTORY METHOD [1]

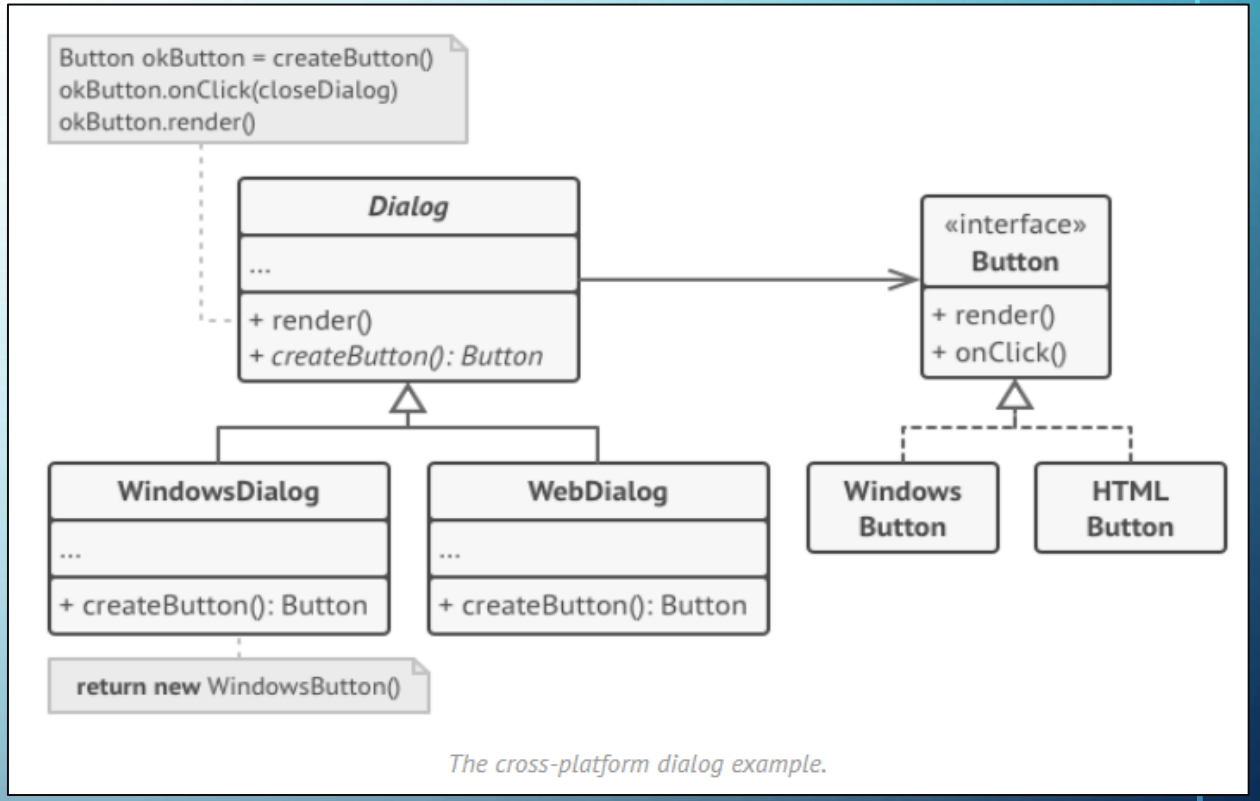
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.



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

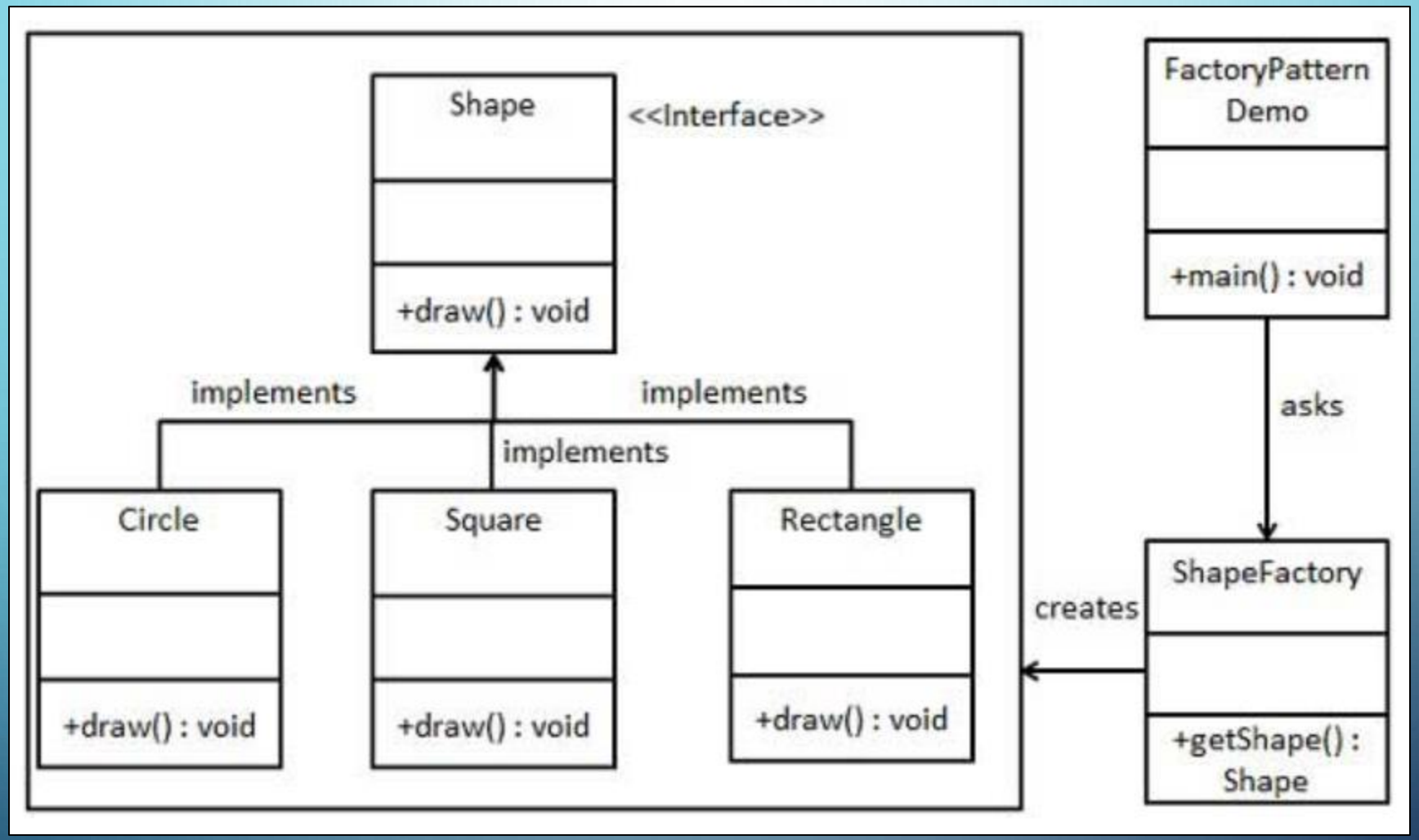
FACTORY METHOD [1]





The cross-platform dialog example.

FACTORY METHOD [2]

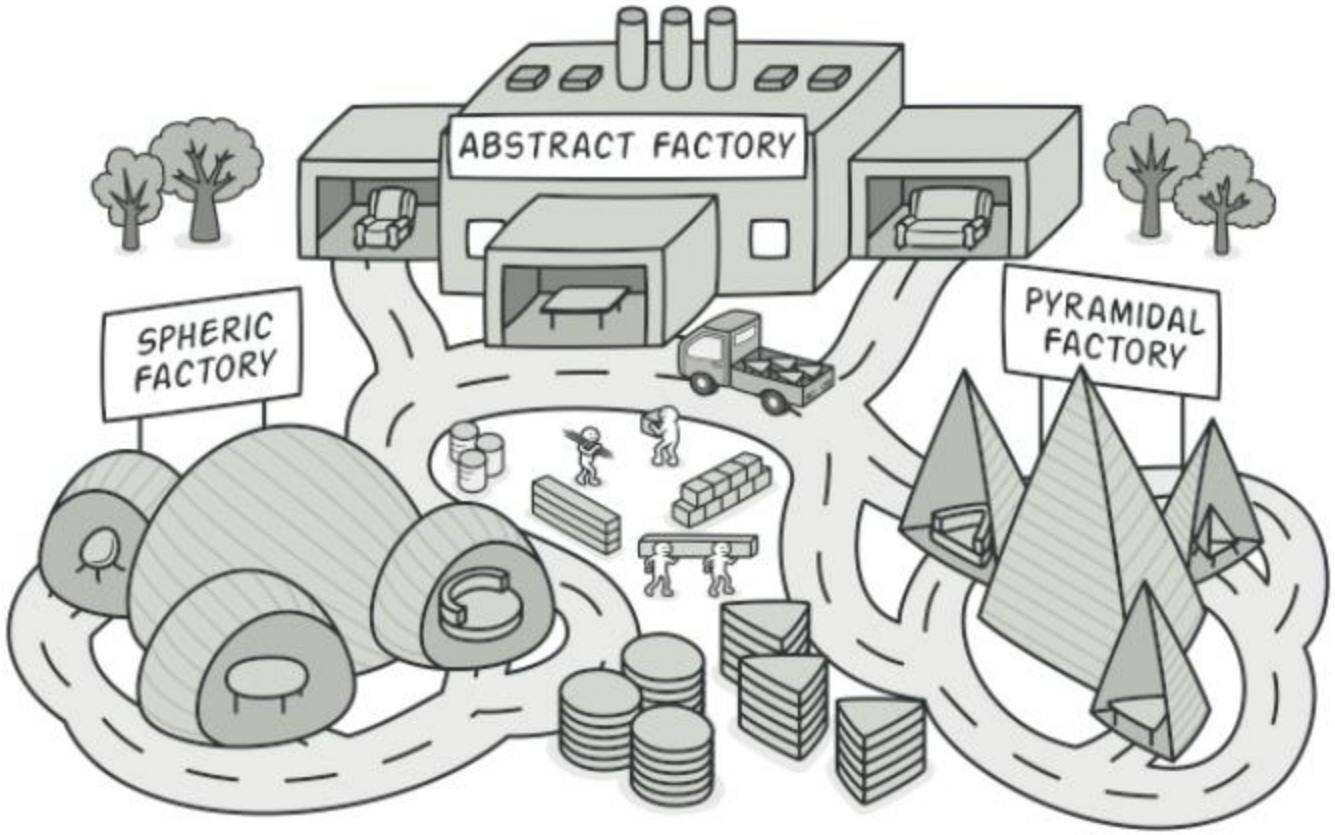


TUTORIAL: “SHAPE” FACTORY METHOD

```
1  public class CPFactoryDemo {
2      public static void main(String[] args) {
3          // Create a factory object
4          ShapeFactory objShapeFactory = new ShapeFactory();
5
6          Shape objShape1 = objShapeFactory.getShape("Rectangle");
7          objShape1.draw();
8
9          Shape objShape2 = objShapeFactory.getShape("Square");
10         objShape2.draw();
11
12         //Tutorial: Please draw a new shape - Rectangle, via Shape's draw method.
13     }
14 }
```

ABSTRACT FACTORY [1]

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

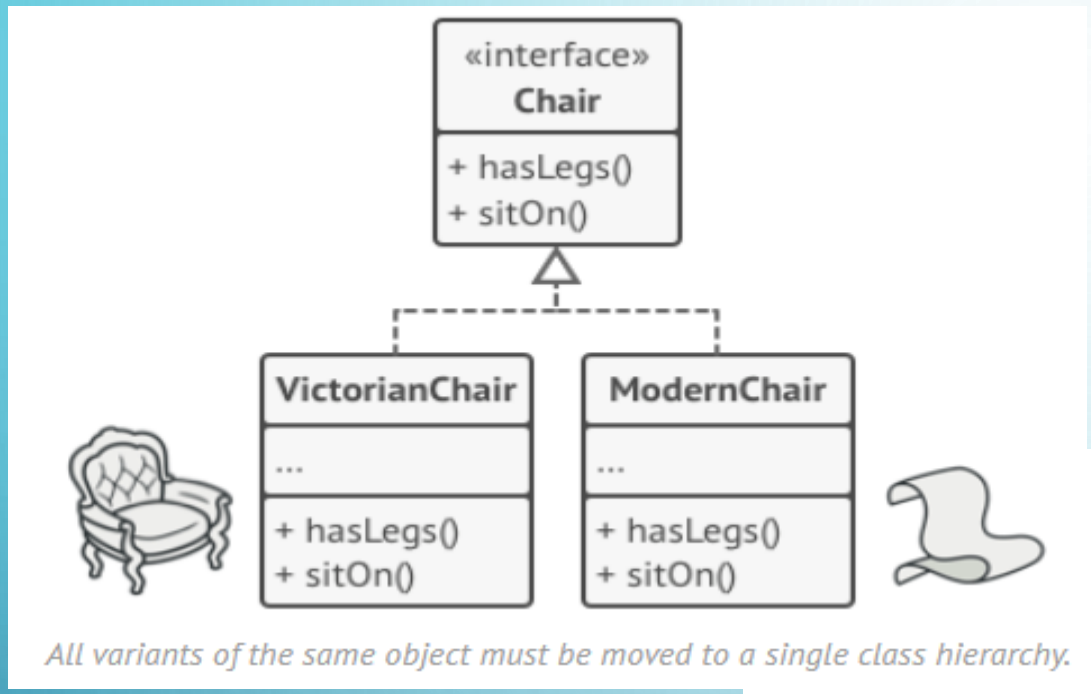


ABSTRACT FACTORY [1]

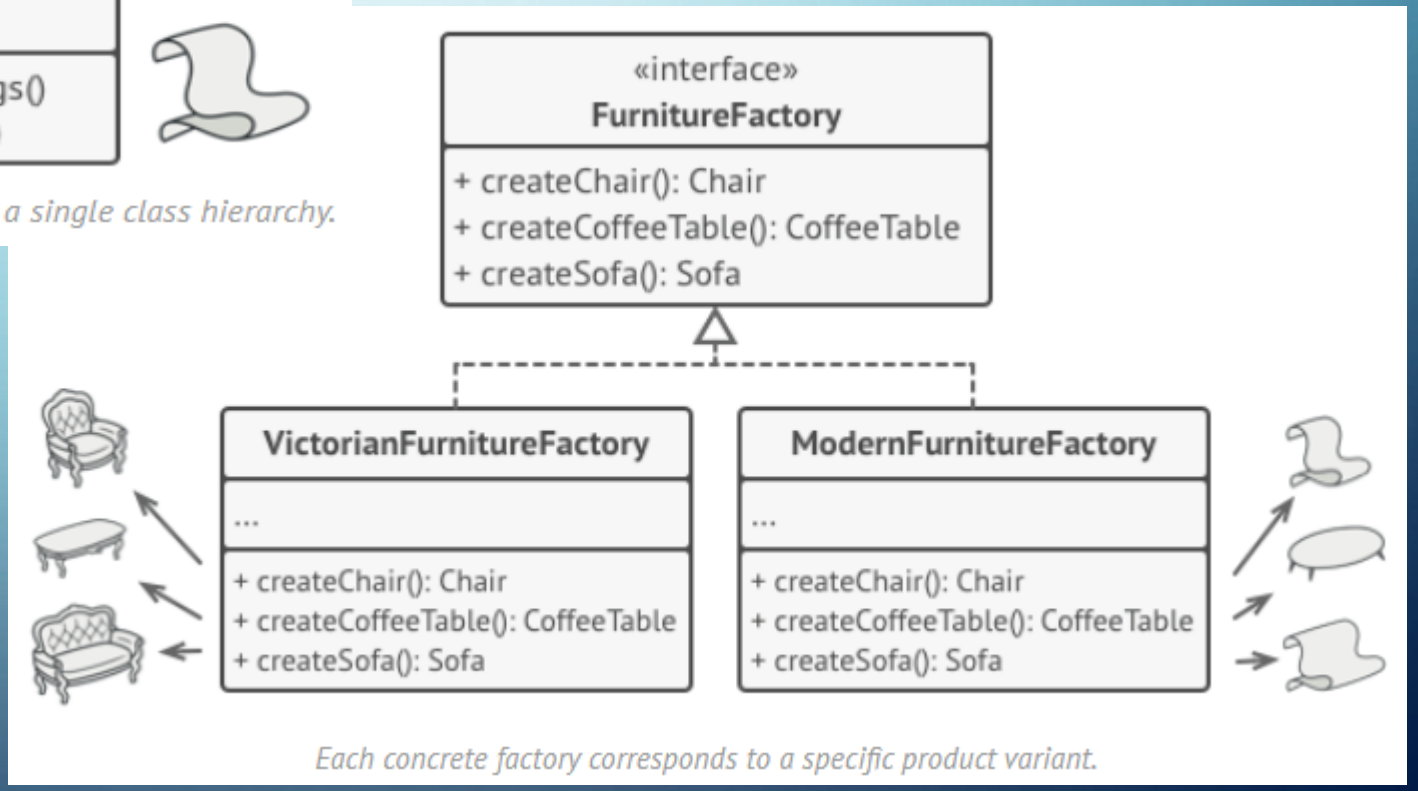
A Modern-style sofa doesn't match Victorian-style chairs.

Product families and their variants.

ABSTRACT FACTORY [1]

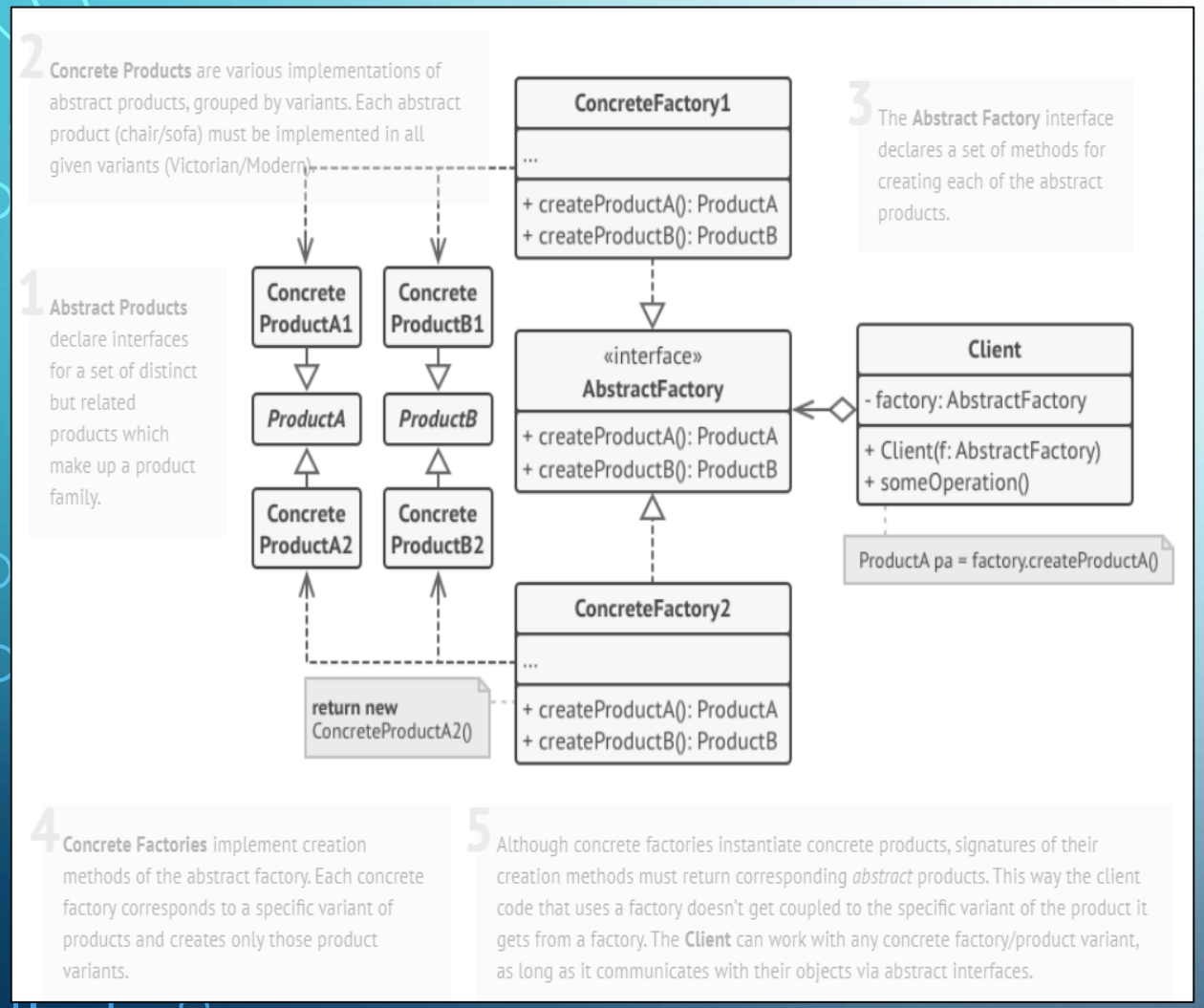


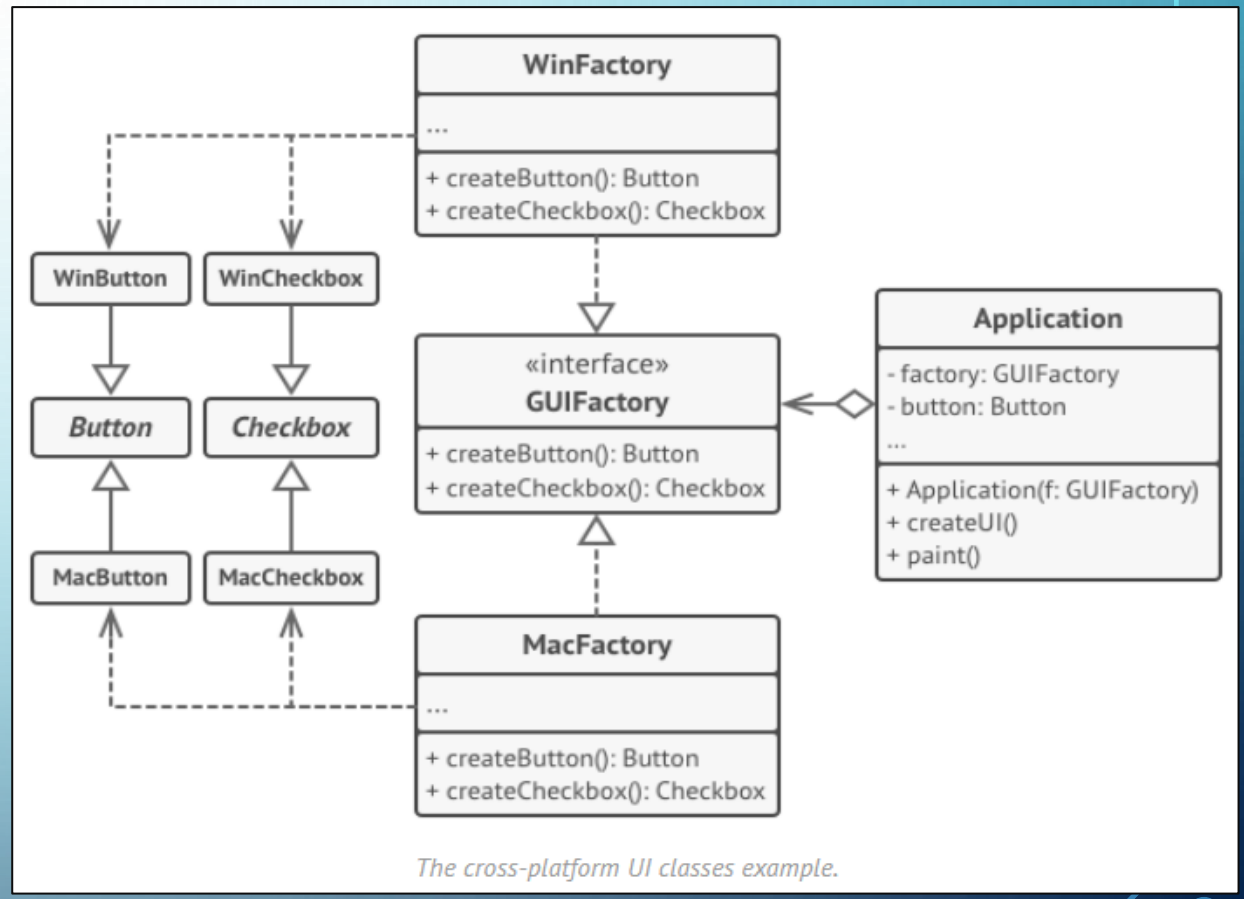
All variants of the same object must be moved to a single class hierarchy.



Each concrete factory corresponds to a specific product variant.

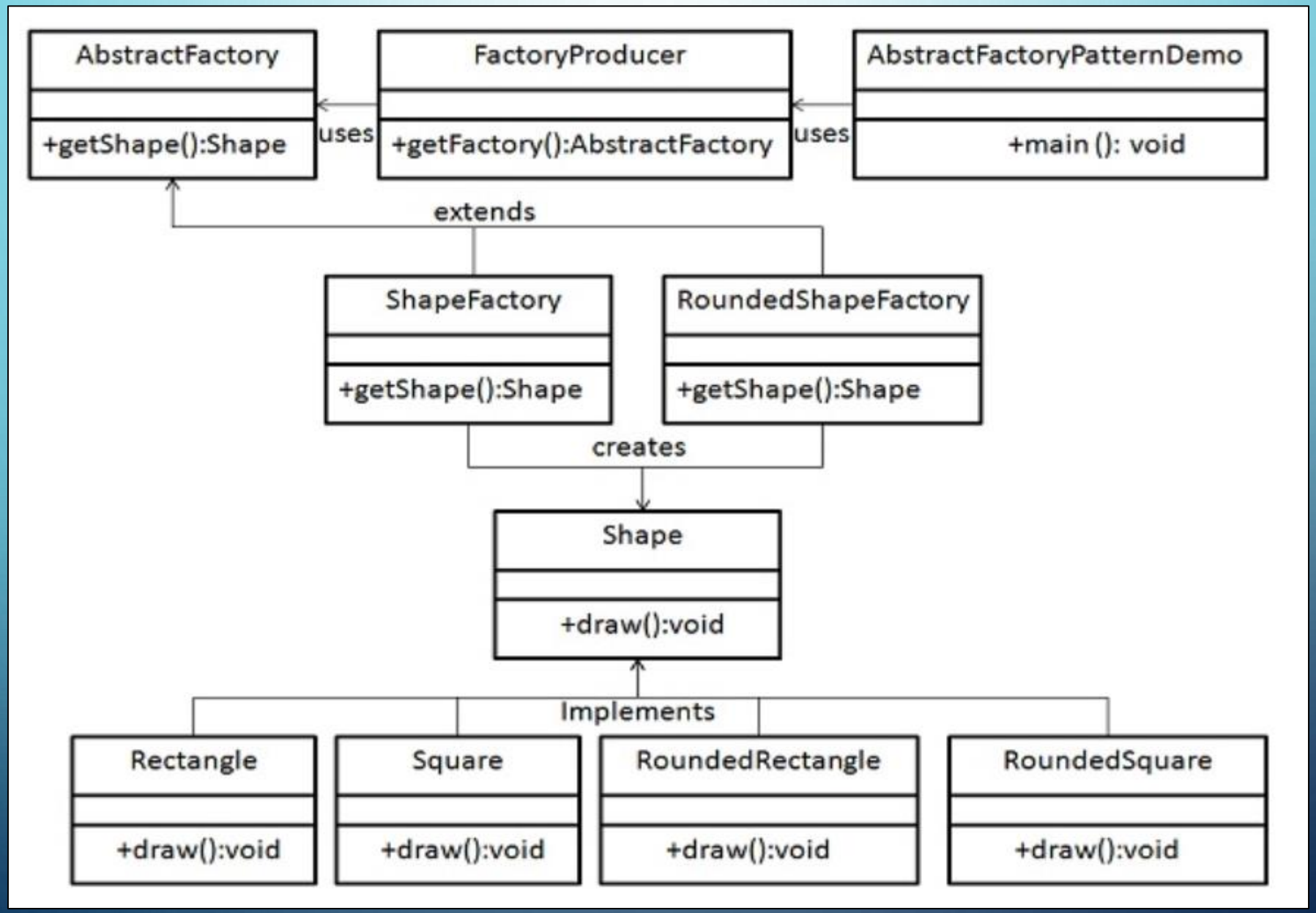
FACTORY METHOD [1]





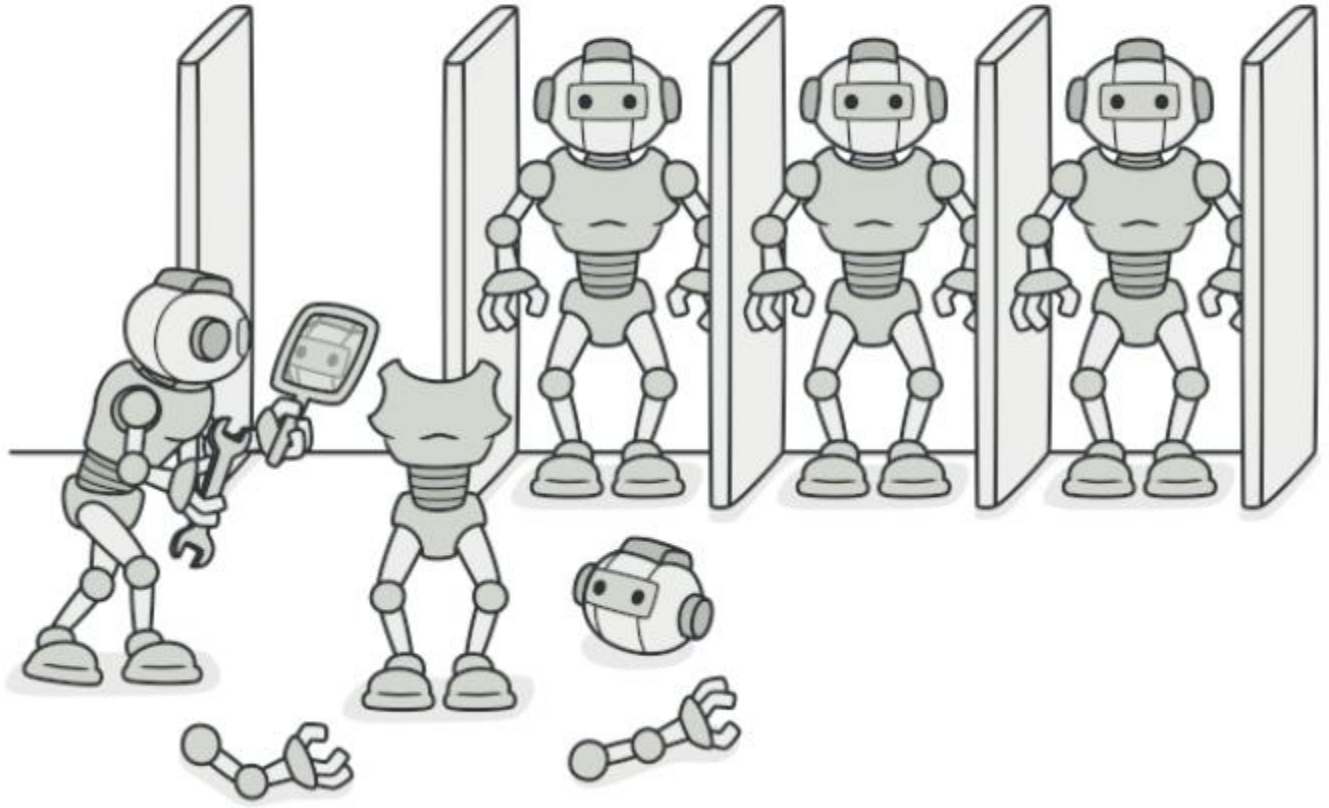
The cross-platform UI classes example.

ABSTRACT FACTORY [2]

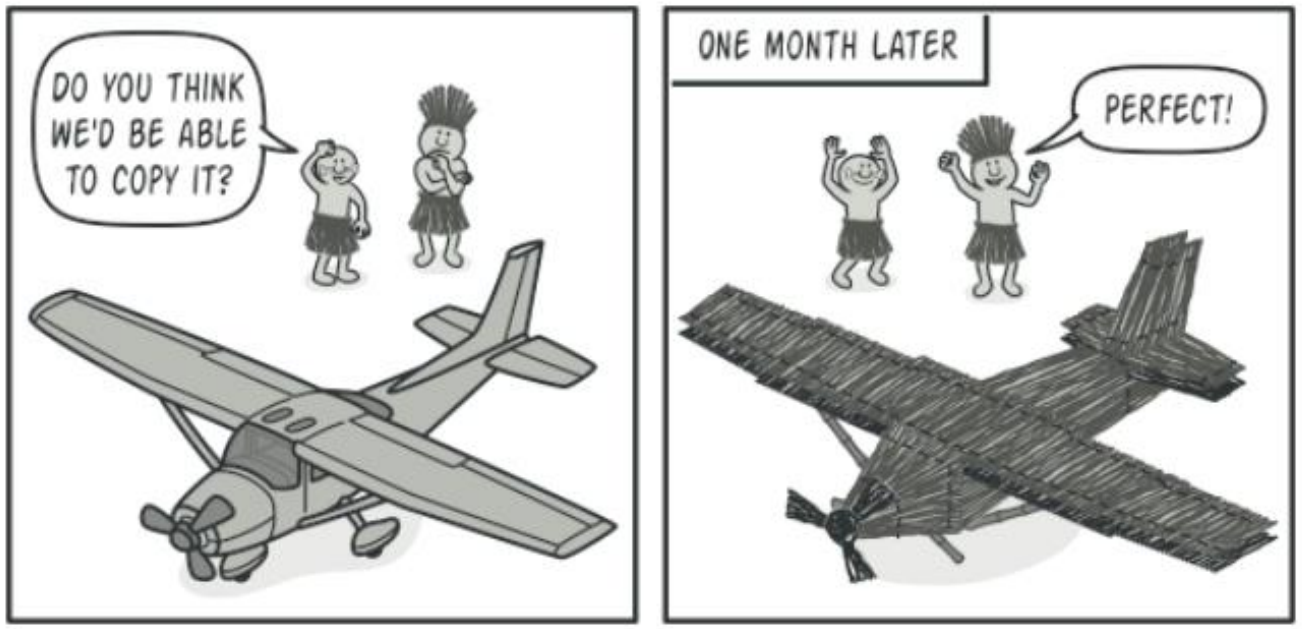


PROTOTYPE [1]

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



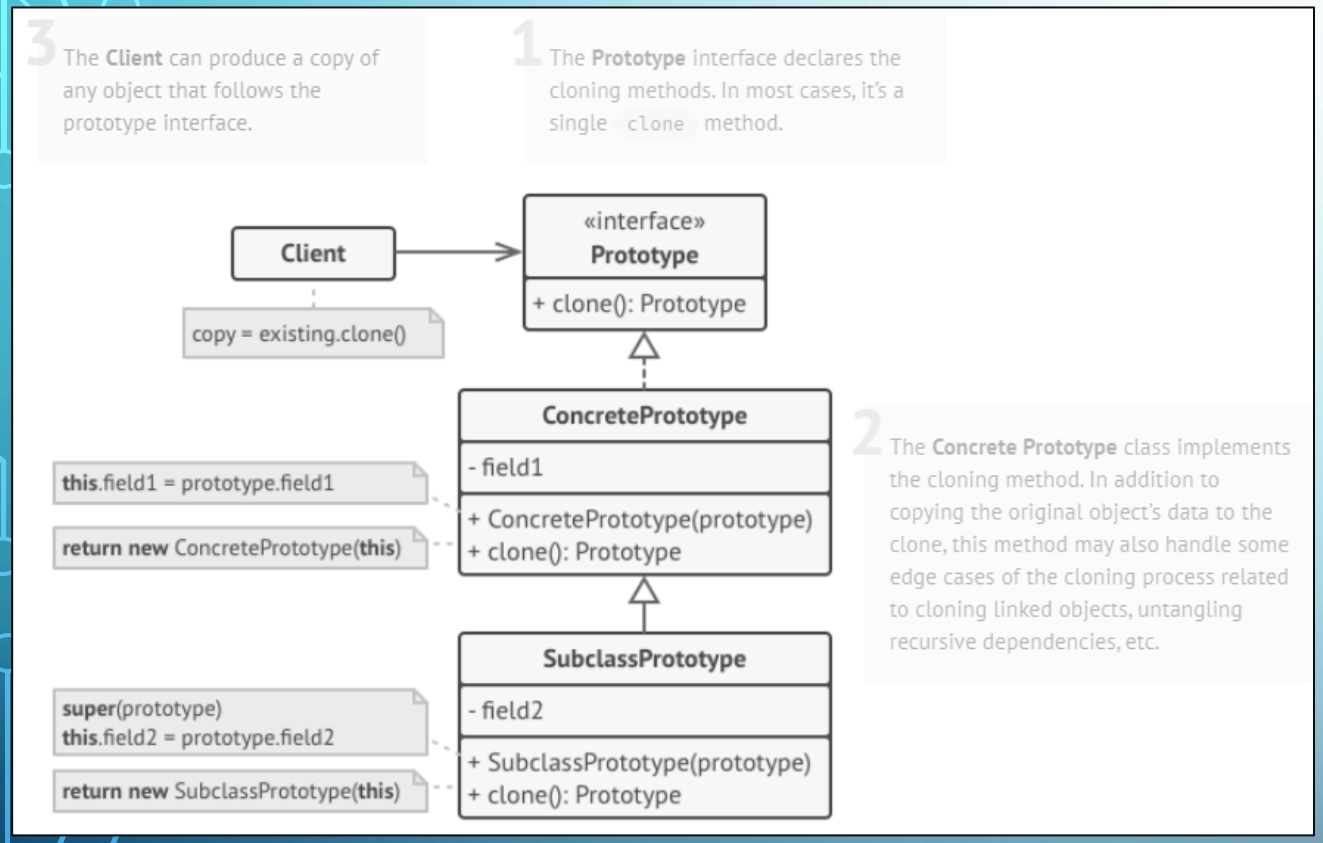
PROTOTYPE [1]

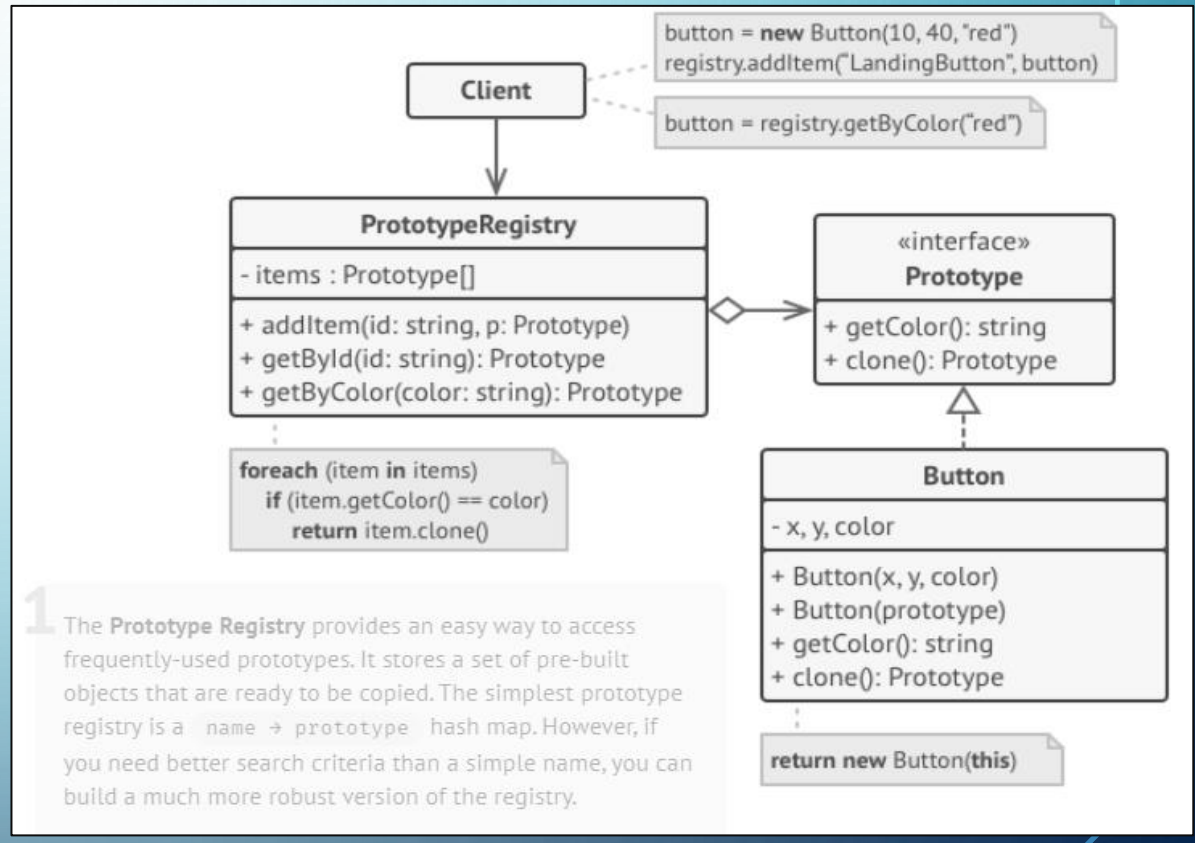


*Copying an object "from the outside" **isn't** always possible.*

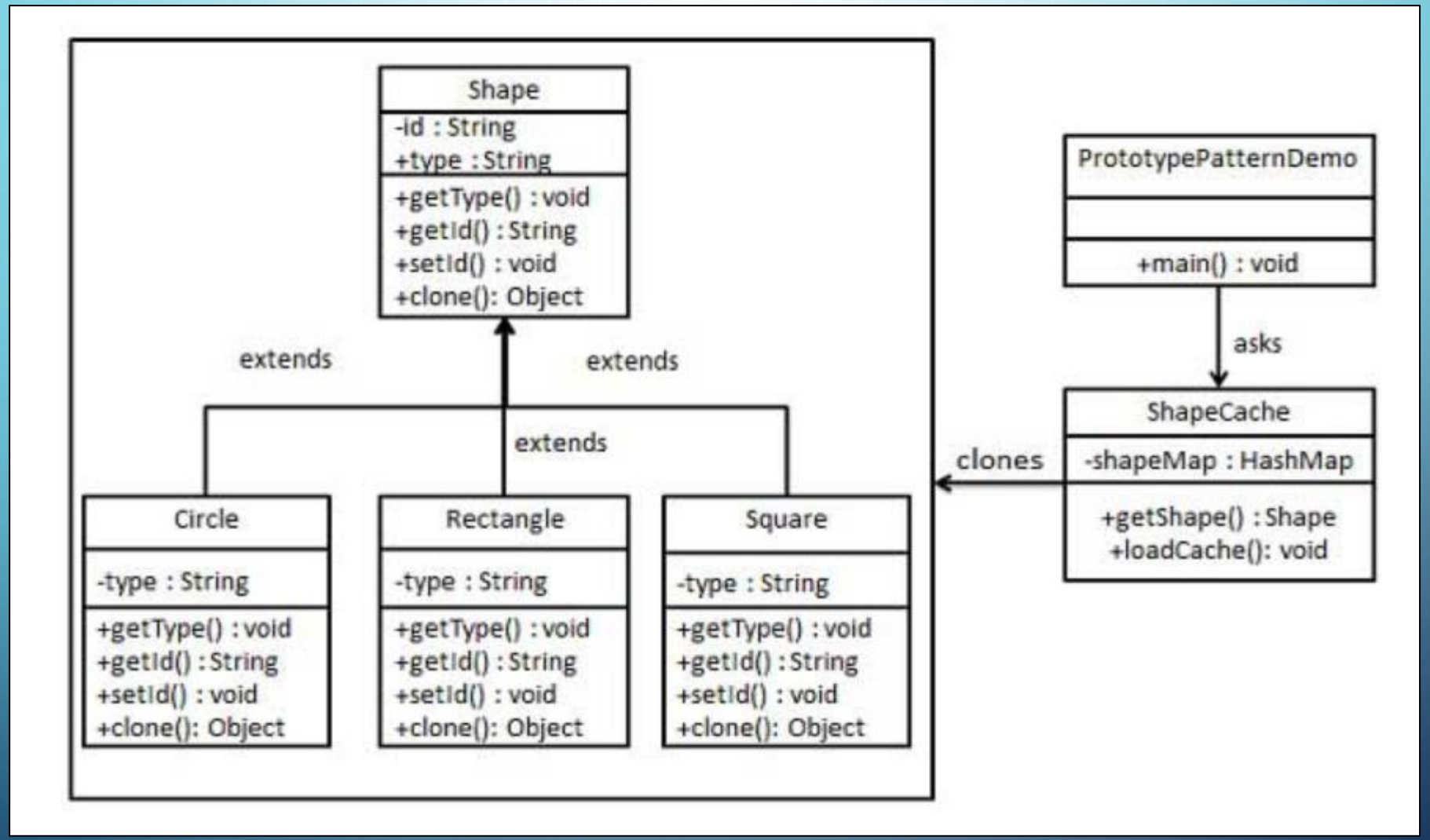
Pre-built prototypes can be an alternative to subclassing.

PROTOTYPE [1]


3 The Client can produce a copy of any object that follows the prototype interface.
1 The Prototype interface declares the cloning methods. In most cases, it's a single clone method.
Client
copy = existing.clone()
«interface»
Prototype
+ clone(): Prototype
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)
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.

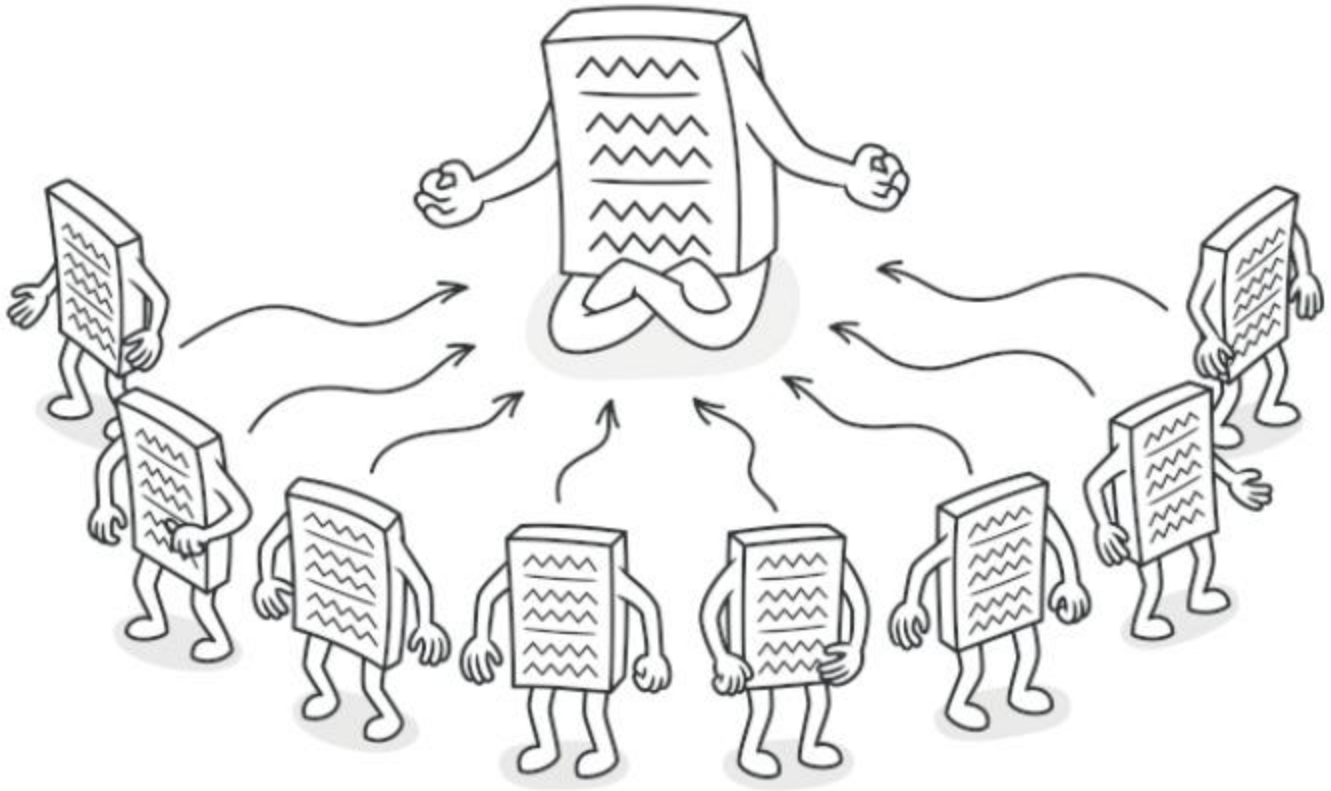

button = new Button(10, 40, "red")
registry.addItem("LandingButton", button)
Client
button = registry.getByColor("red")
PrototypeRegistry
- items : Prototype[]
+ addItem(id: string, p: Prototype)
+ getById(id: string): Prototype
+ getByColor(color: string): Prototype
foreach (item in items)
if (item.getColor() == color)
return item.clone()
«interface»
Prototype
+ getColor(): string
+ clone(): Prototype
Button
- x, y, color
+ Button(x, y, color)
+ Button(prototype)
+ getColor(): string
+ clone(): Prototype
return new Button(this)
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.

PROTOTYPE [2]

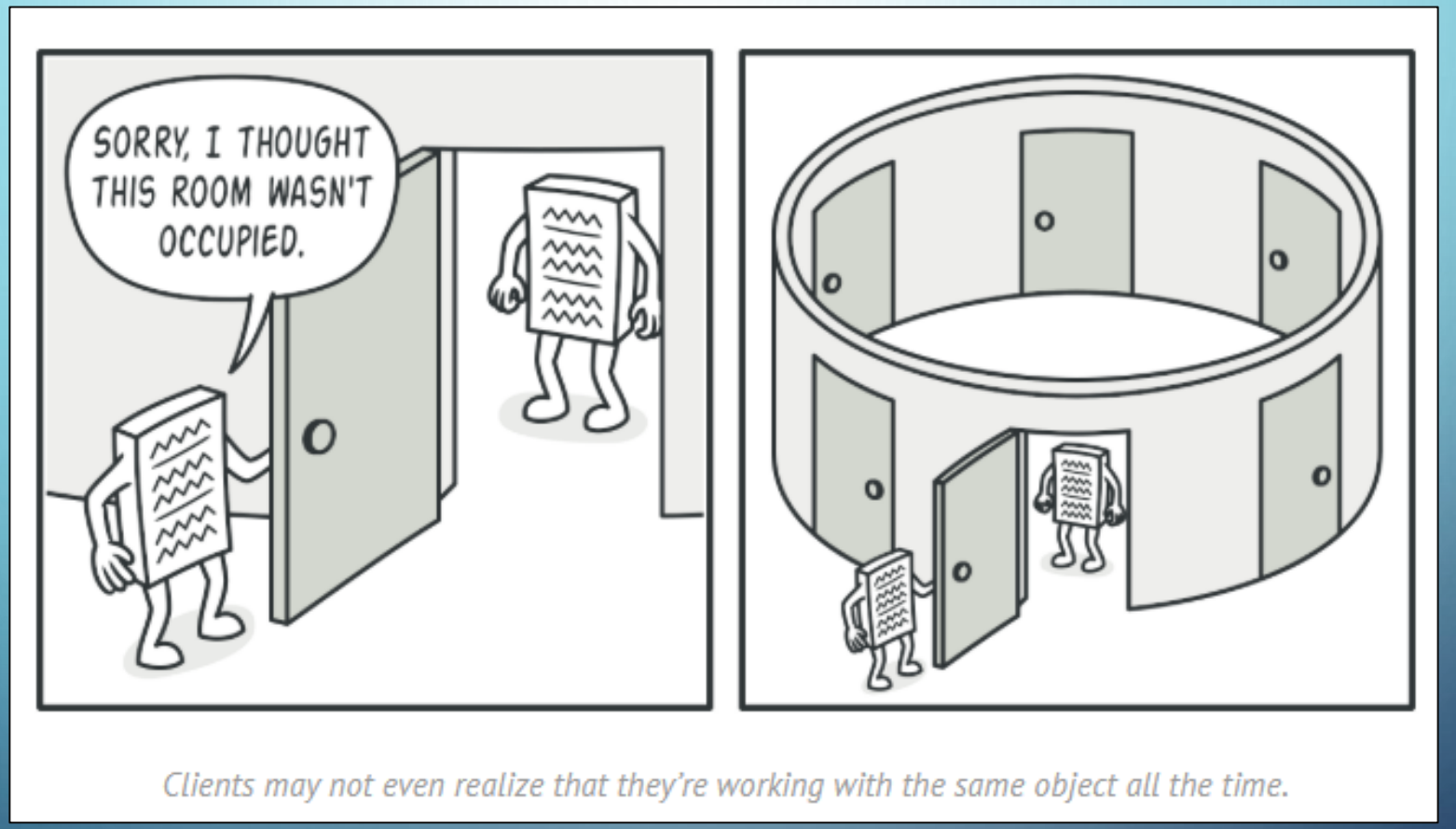

Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
clones
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void

SINGLETON [1]

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.

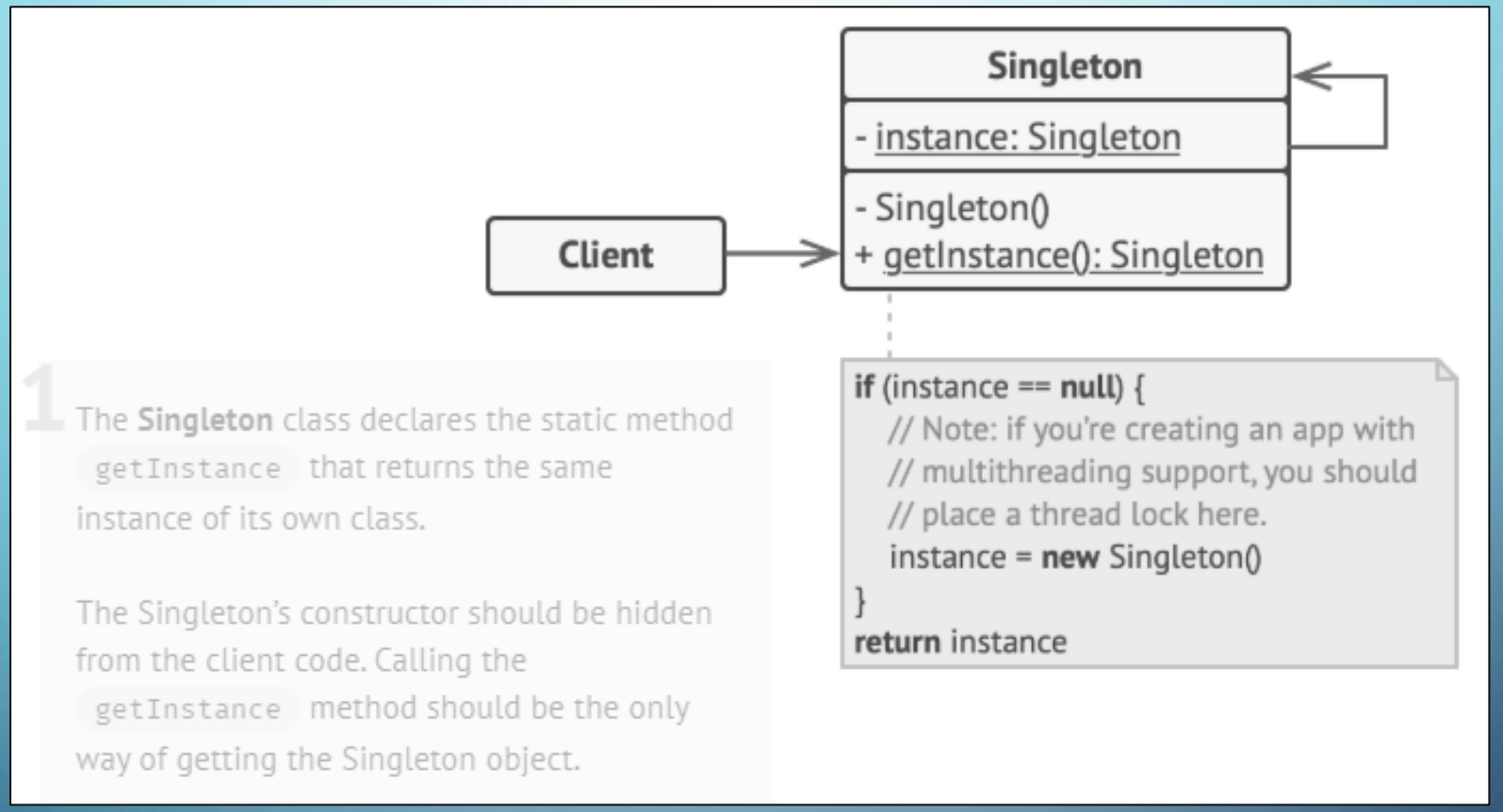


SINGLETON [1]

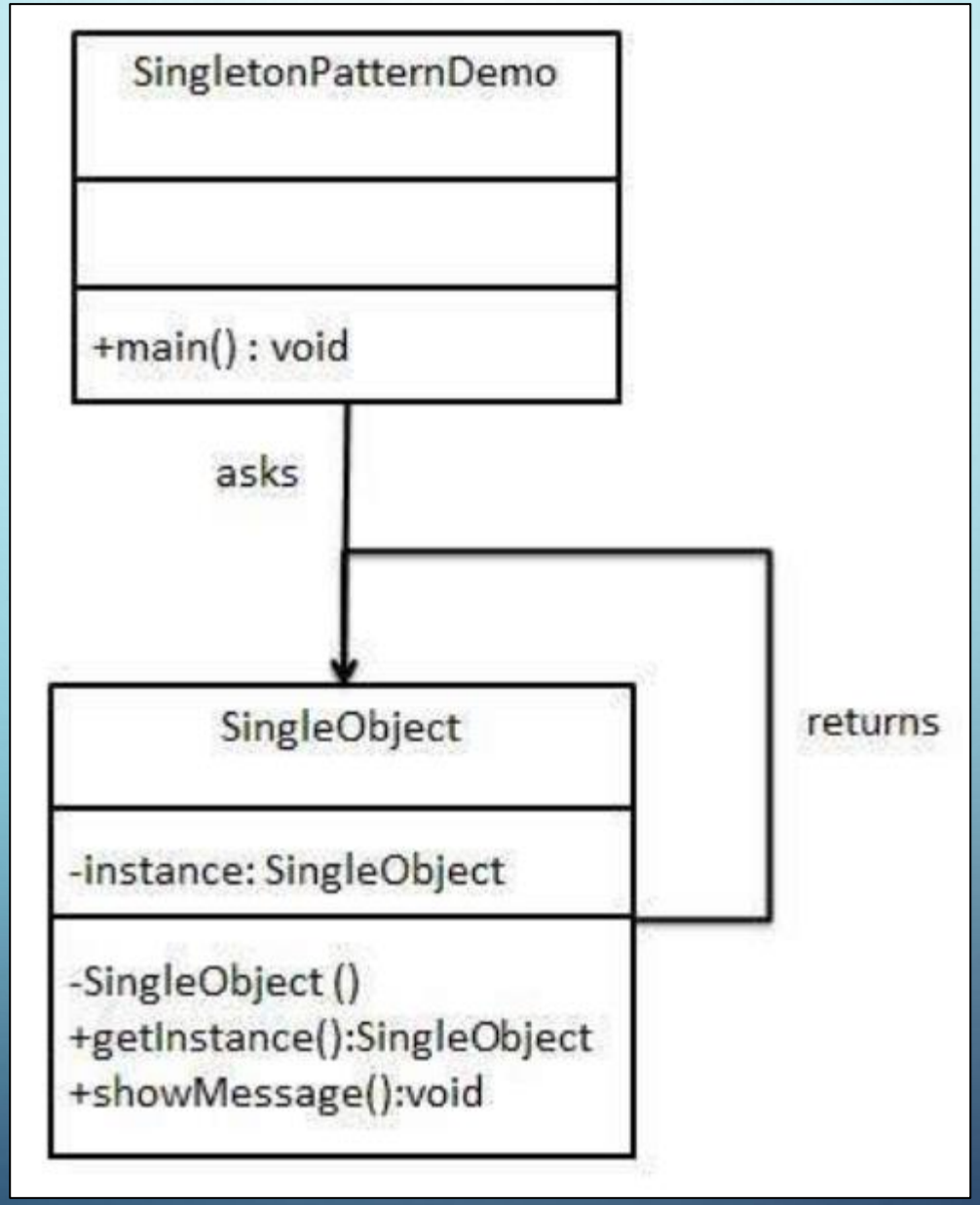


Clients may not even realize that they're working with the same object all the time.

SINGLETON [1]

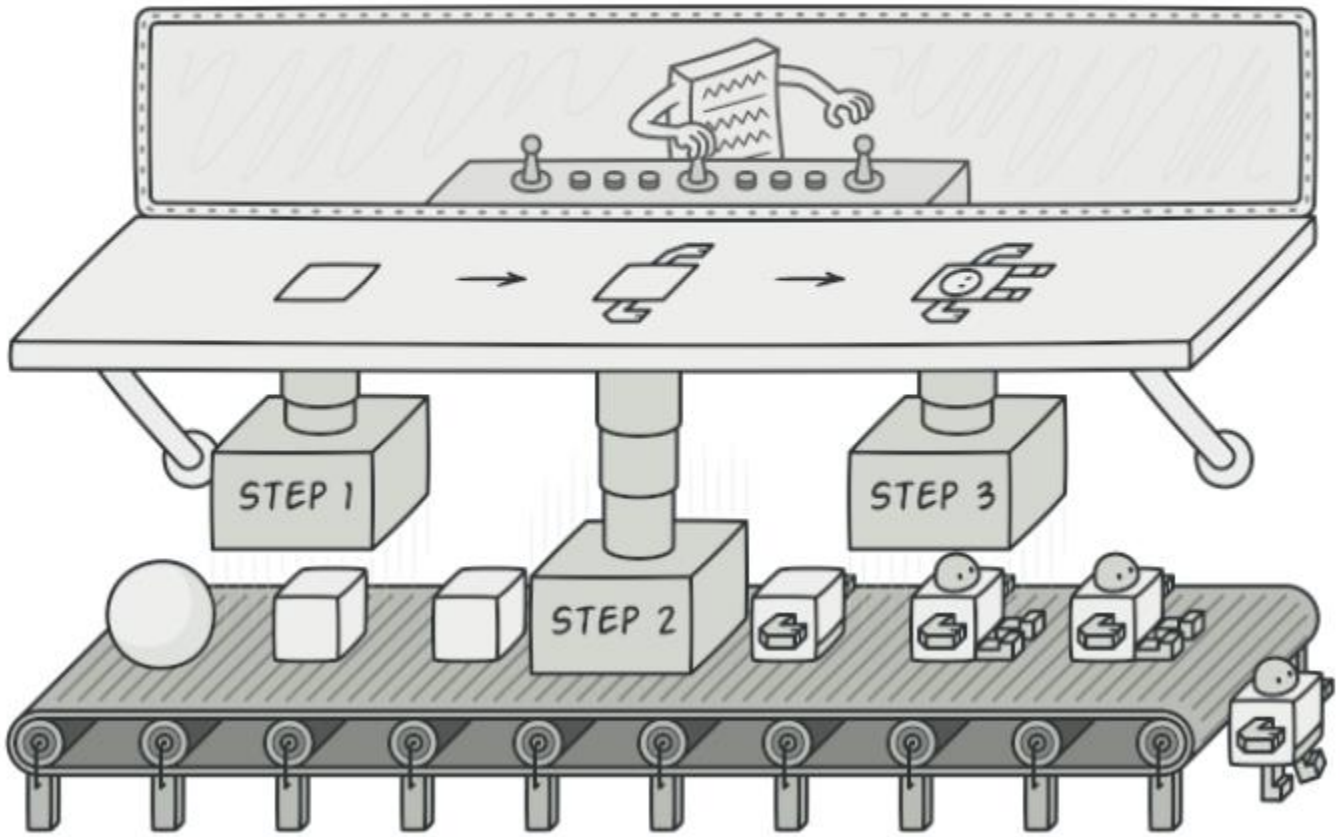

Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
1
The Singleton class declares the static method getInstance that returns the same instance of its own class.
The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.
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

SINGLETON [2]

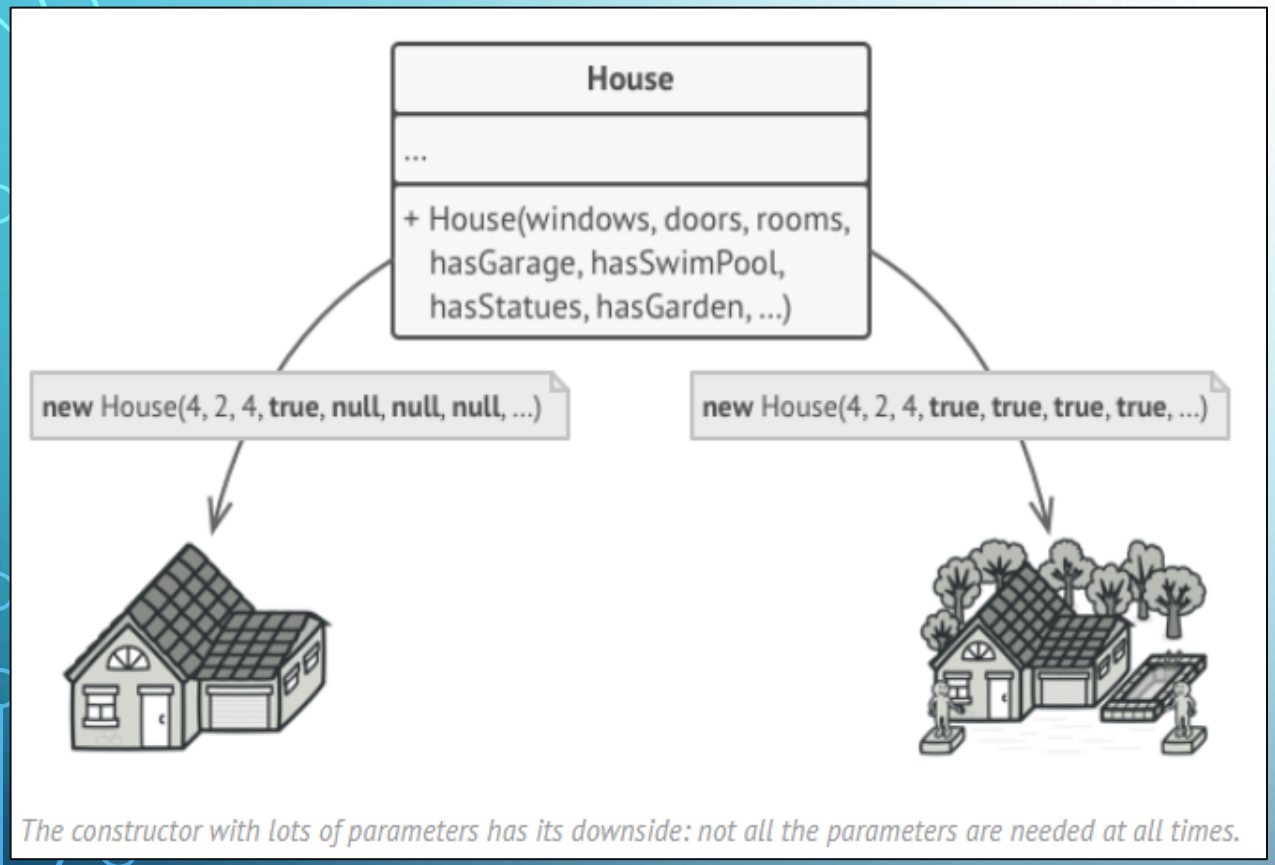

SingletonPatternDemo
+main() : void
asks
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void
returns

BUILDER [1]

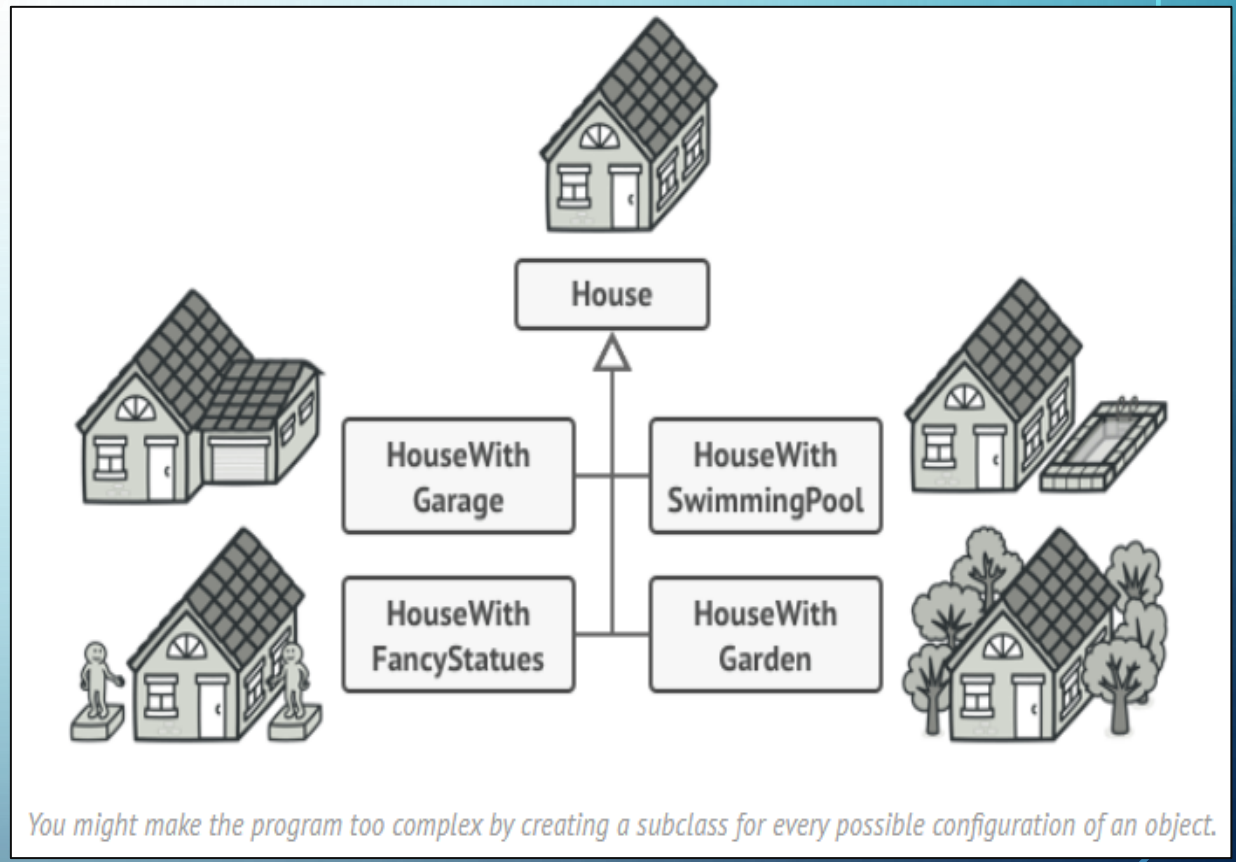
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



BUILDER [1]

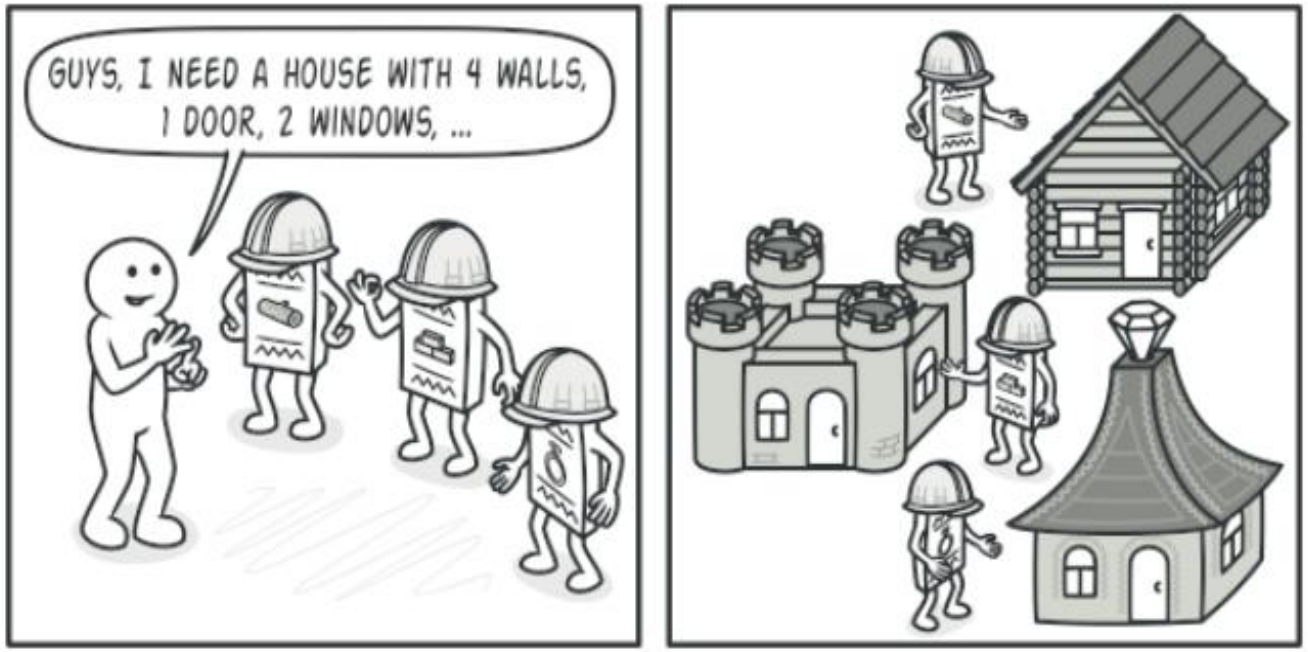


The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

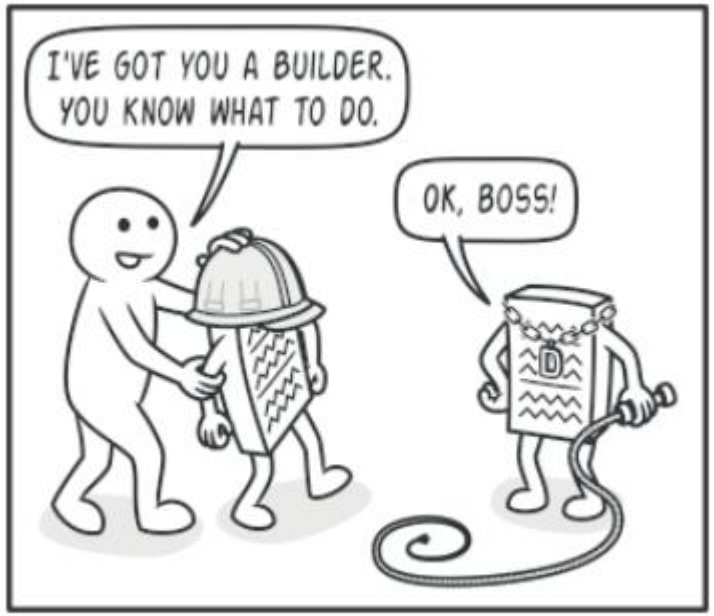


You might make the program too complex by creating a subclass for every possible configuration of an object.

BUILDER [1]

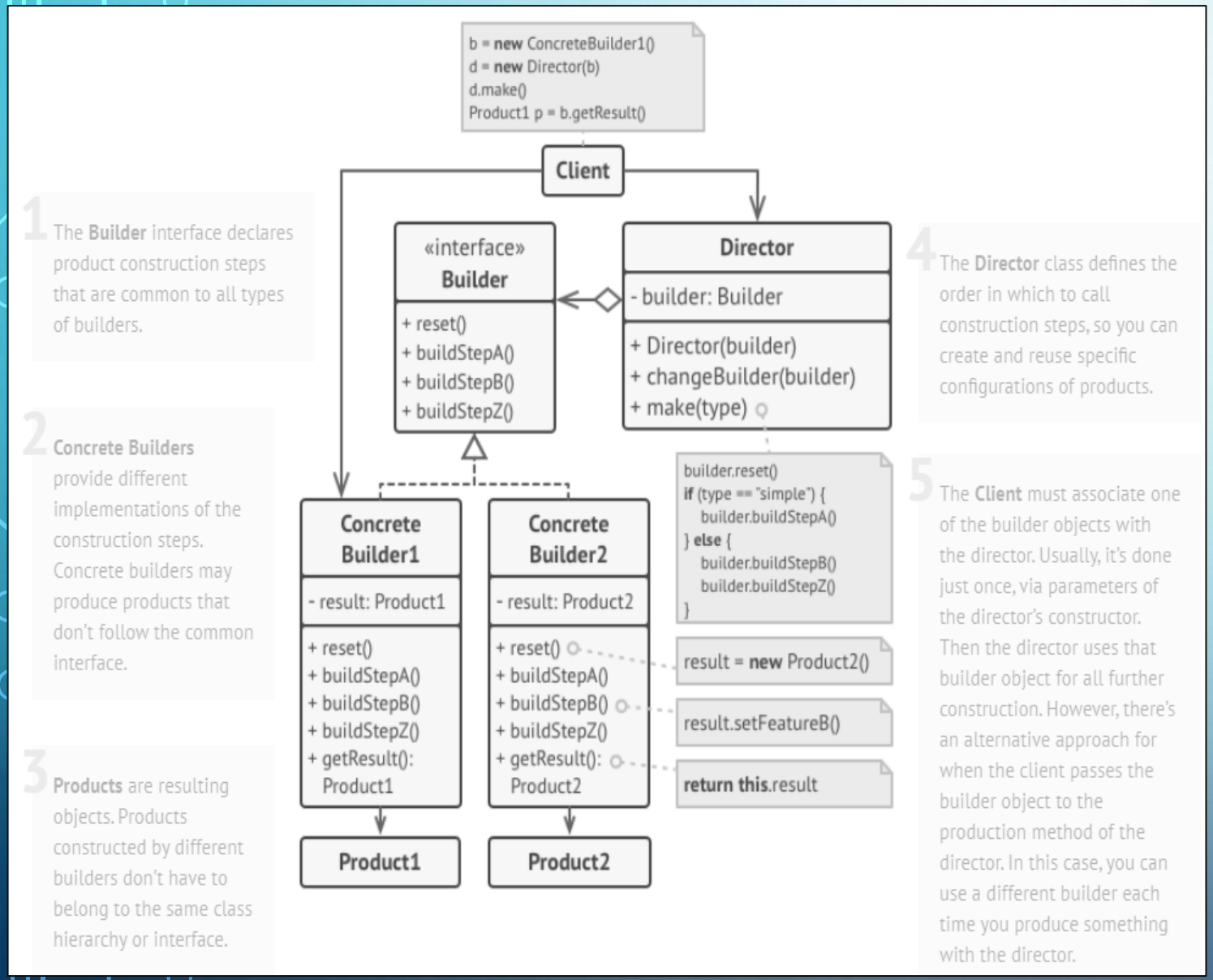


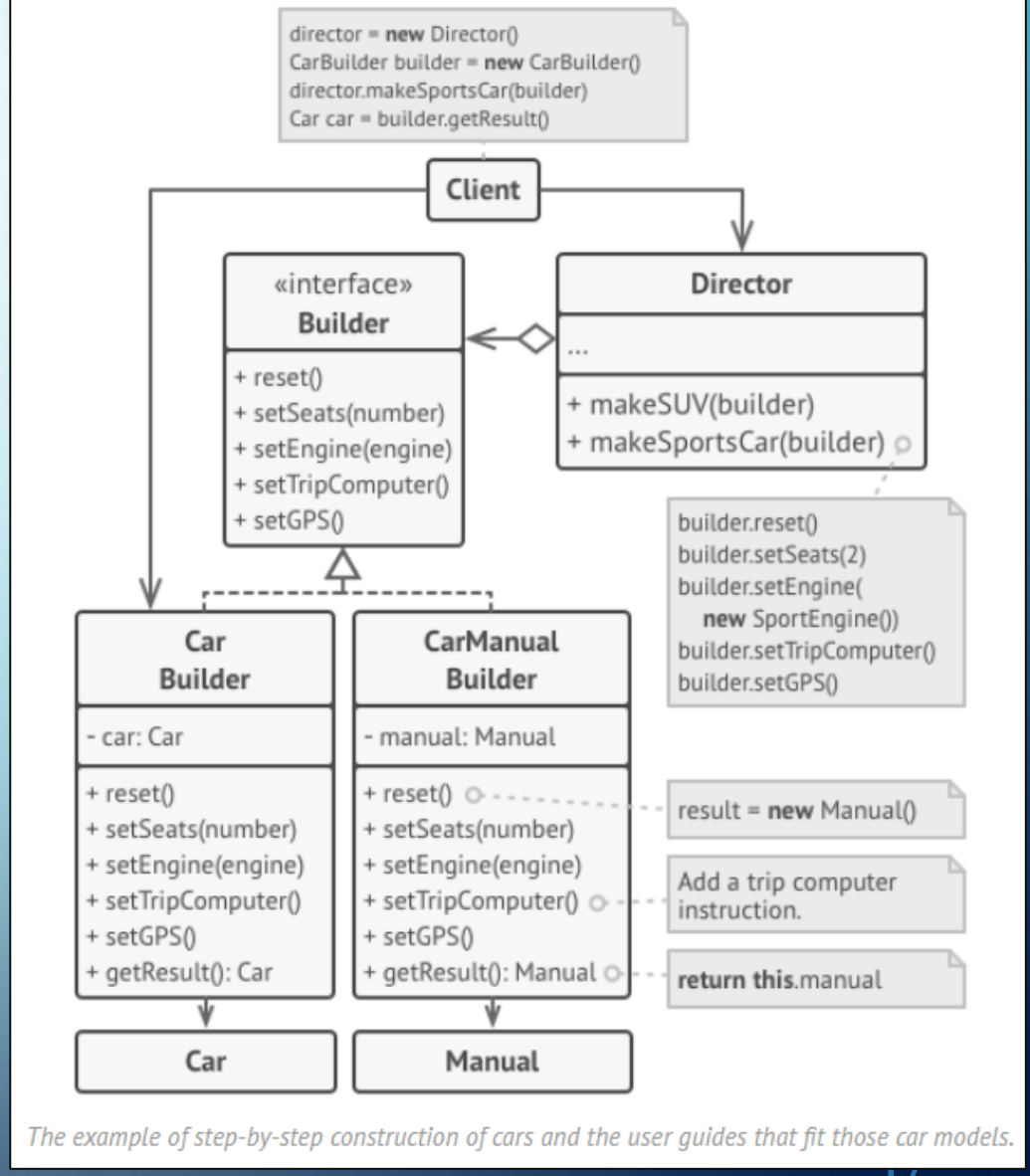
Different builders execute the same task in various ways.



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

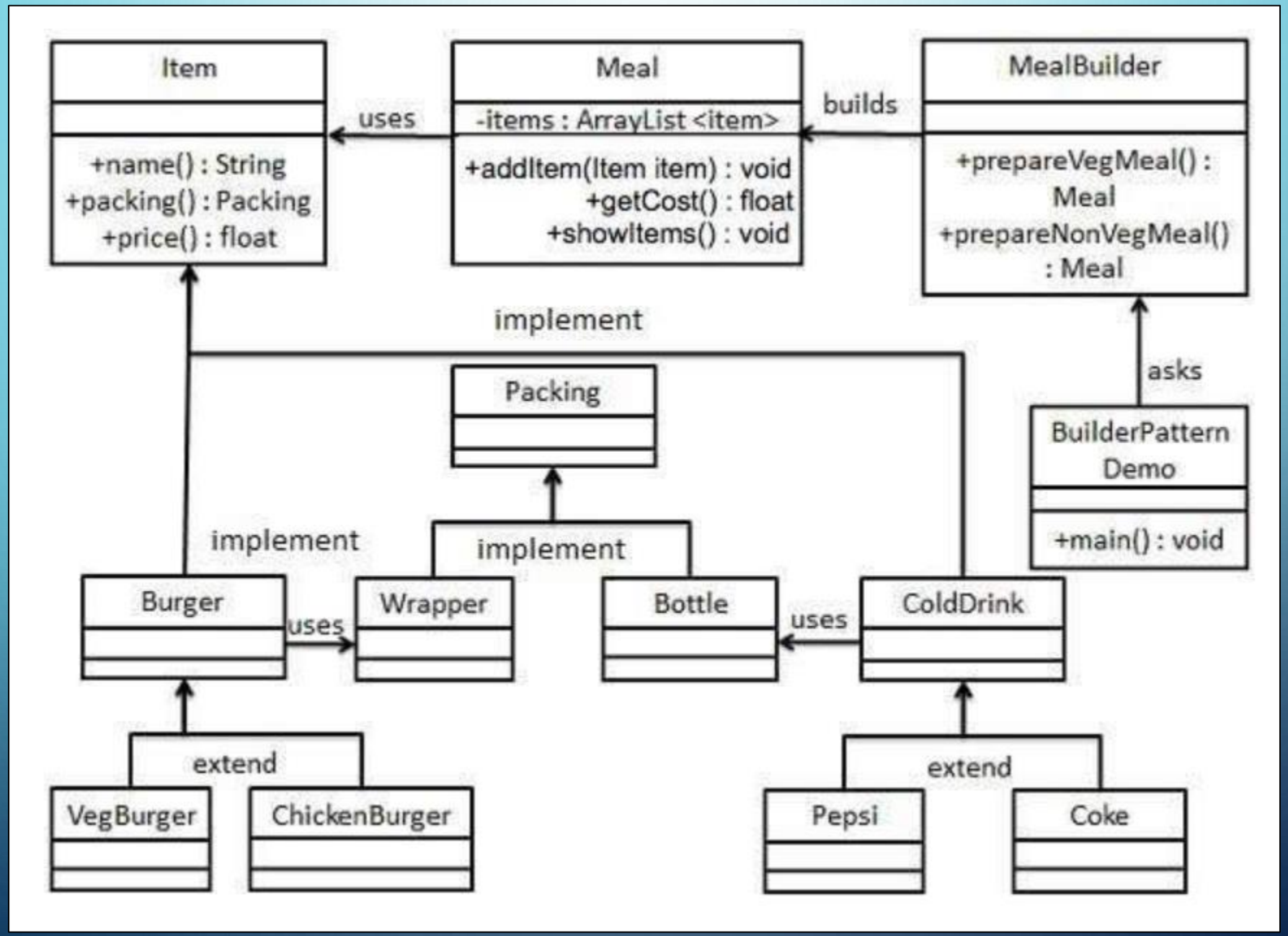
BUILDER [1]





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

BUILDER [2]


Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Packing
BuilderPattern Demo
+main() : void
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
extend
VegBurger
ChickenBurger
Pepsi
Coke

REFERENCES

[1] https://refactoring.guru/design-patterns/what-is-pattern

[2] https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm