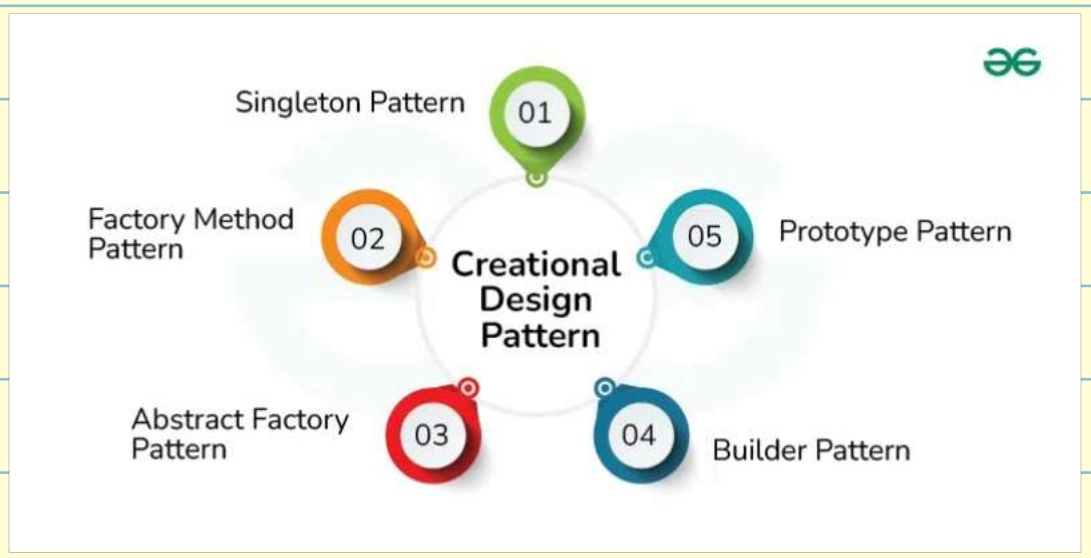
Design Patterns

Design patterns → Creational
→ Structural
→ Behavioral

Creational patterns : Provide more flexibility in how the objects are actually created.

Structural patterns : deal with how inheritance and composition can be used to provide extra functionality.

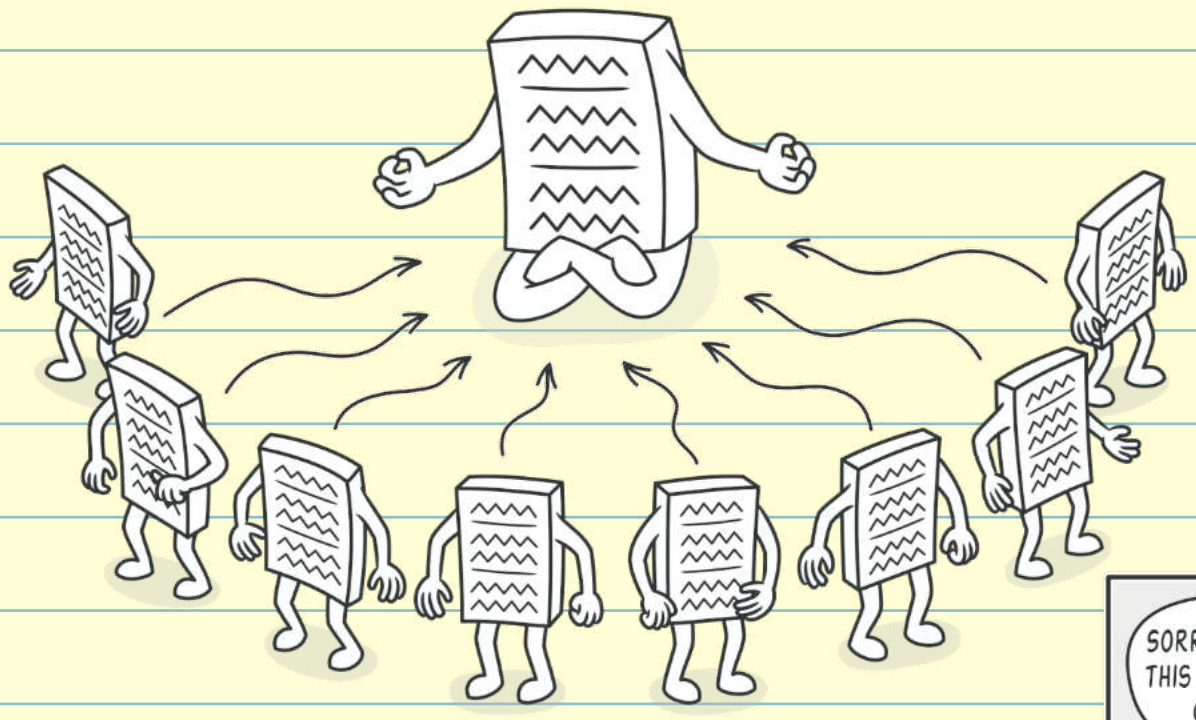
Behavioral patterns : are about communication and assignment of responsibilities between our objects.



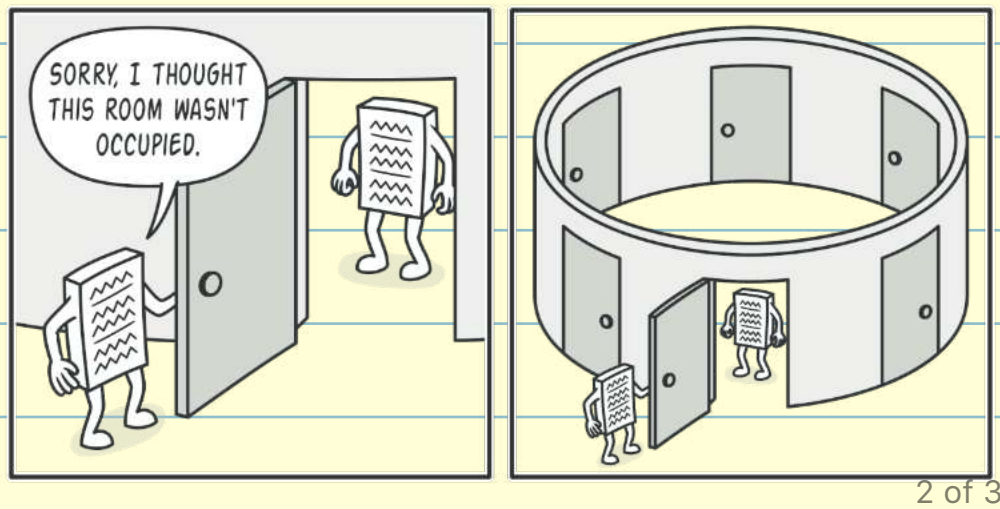
Singleton

- creational
- assure only one instance of its kind exists.
- provide single point of access to it.

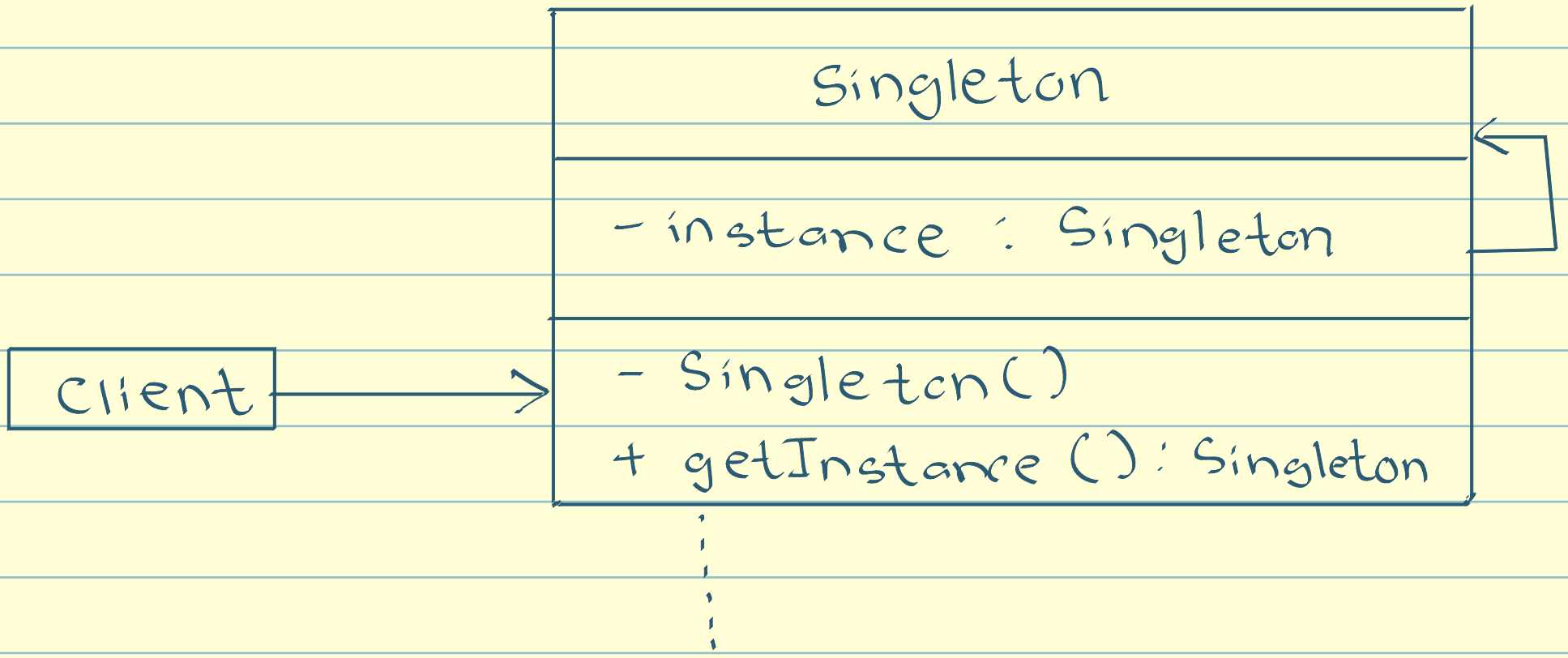
- should be used when a class must have a single instance available.
- disable all means of creating objects of a class except for the special static creation method
- returns the existing instance if it has already been created.
- its code needs to be adapted to handle multiple threads.



(violate single Responsibility Principle)



Structure



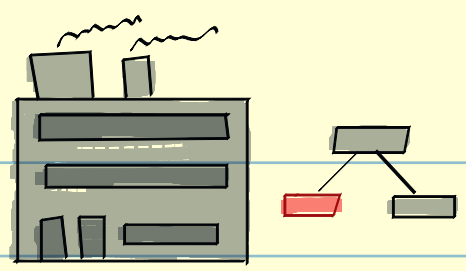

```
if (instance == null) {
    /* Note: if you're creating
    an app with multithreading
    support, you should place a
    thread lock here */
    instance = new Singleton()
}
return instance
```

intent:

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.

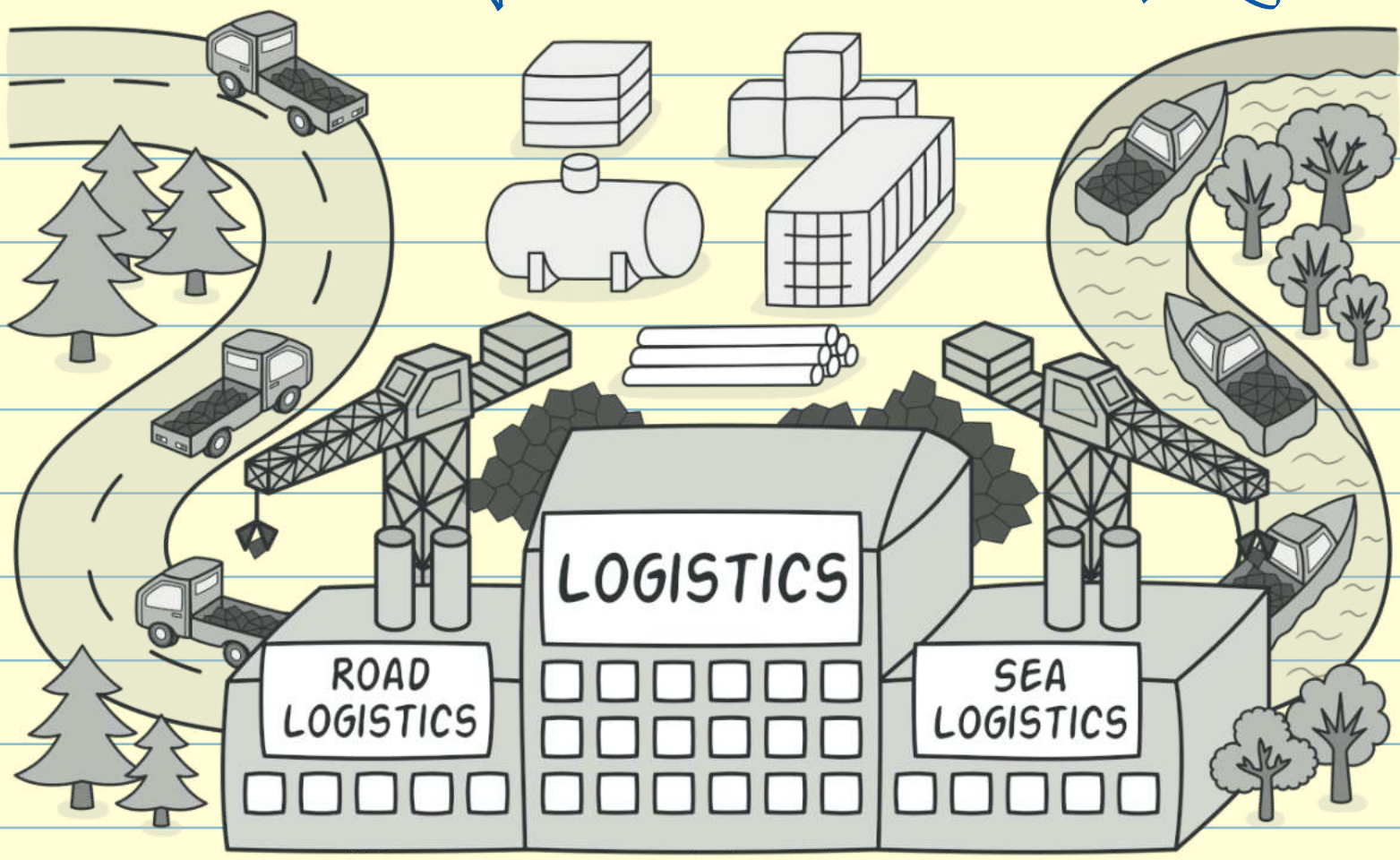
* ensure that a class has just a single instance
* provide global access point to that instance.

Factory method

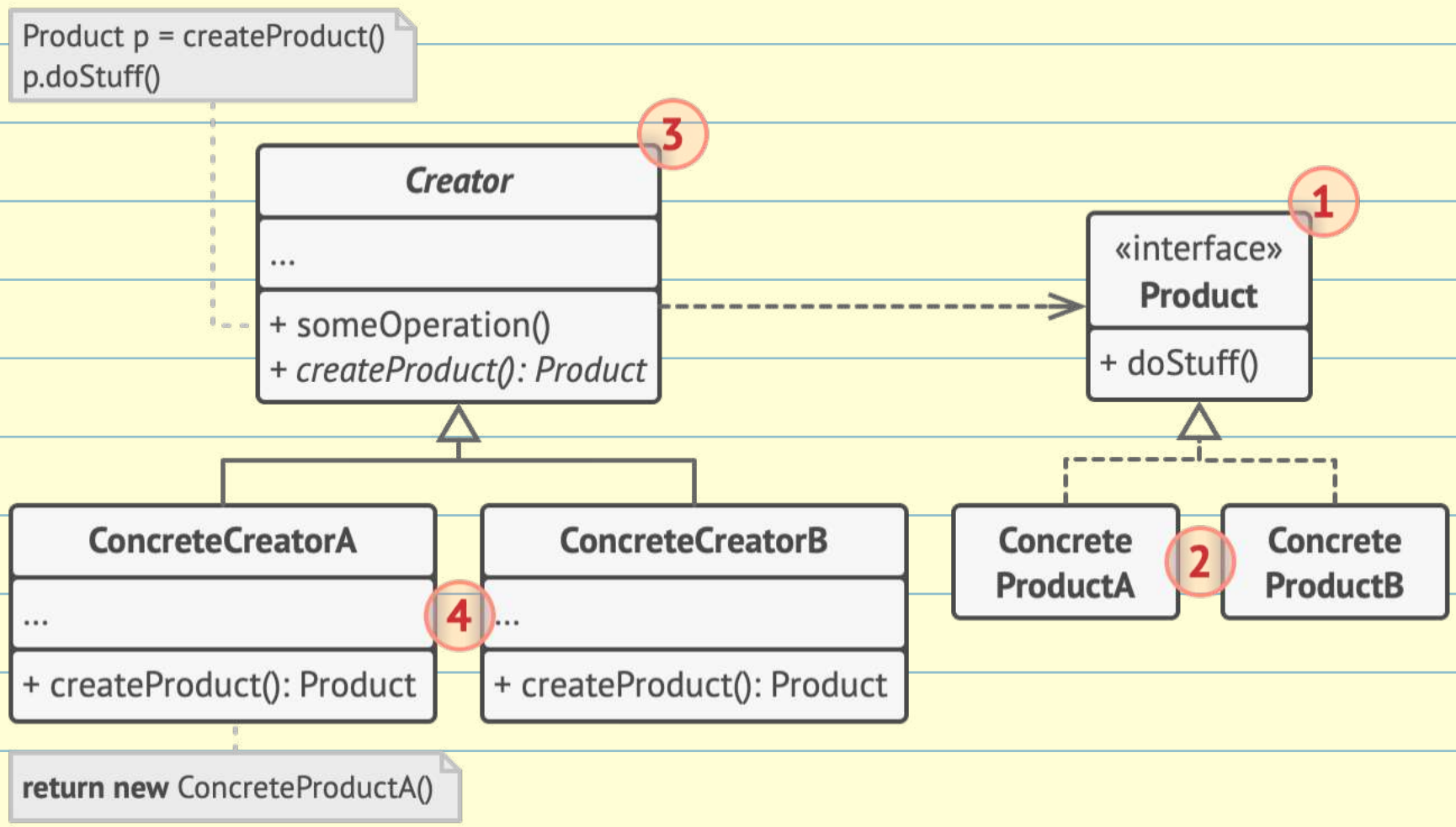


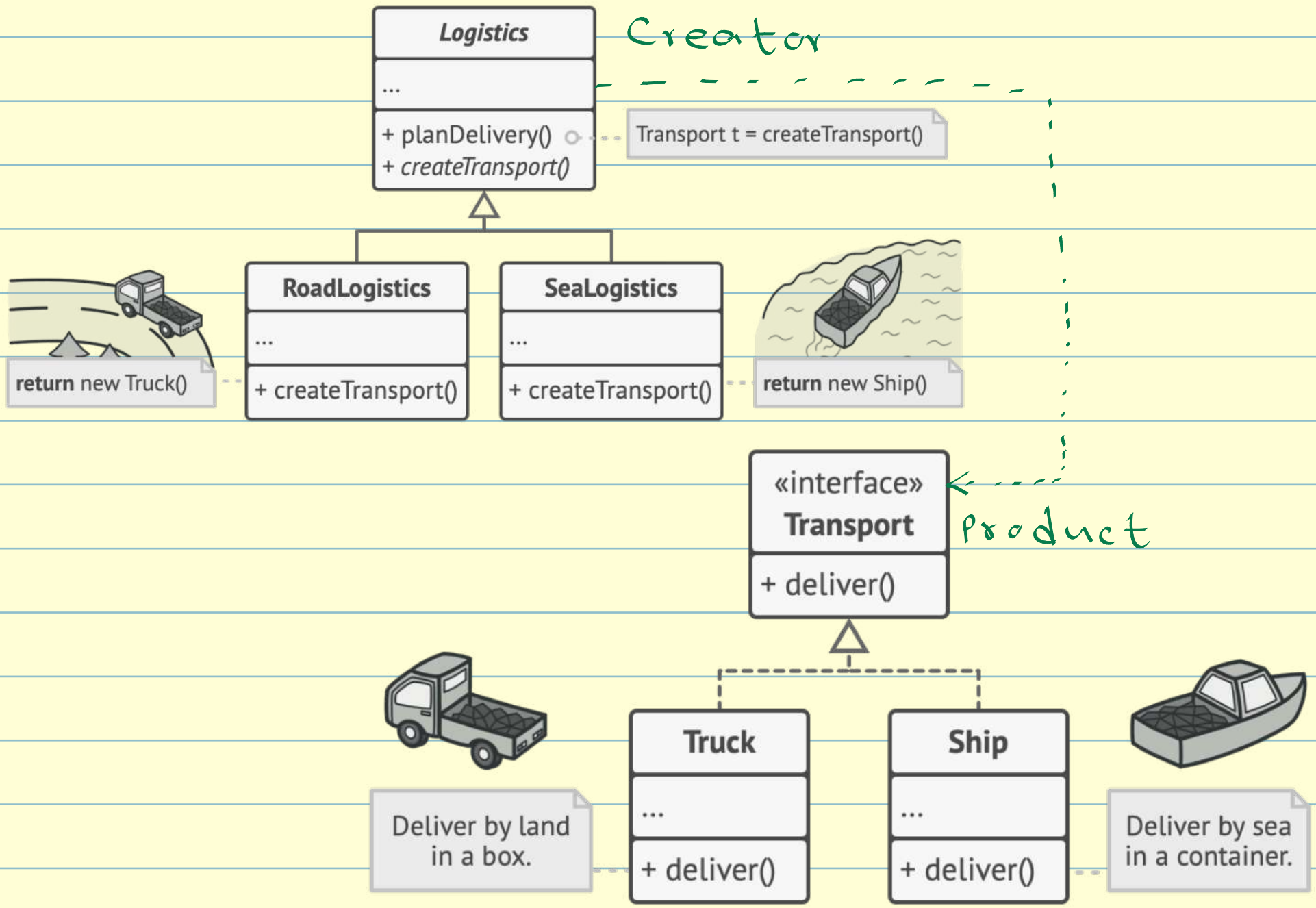
- creational
- loosens the coupling of a given code by seperating the product's construction code from the code that uses this product.

- use if you have no idea of the exact types of the objects your code will work with.
- makes it easy to extend the product construction code indipendently from the rest of the application.
- allows introducing new products without breaking existing code.
- centralizes the product creation code in one place in the program.

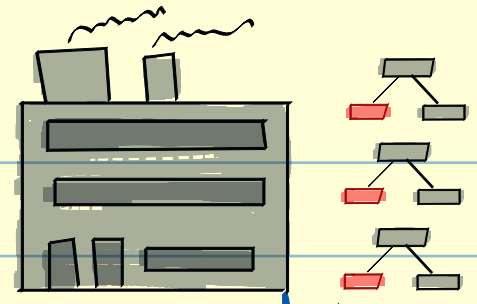


Structure


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
3
«interface»
Product
+ doStuff()
1
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
4
Concrete ProductA
Concrete ProductB
2
return new ConcreteProductA()


Logistics
...
+ planDelivery()
+ createTransport()
Creator
Transport t = createTransport()
RoadLogistics
...
+ createTransport()
SeaLogistics
...
+ createTransport()
return new Truck()
return new Ship()
«interface»
Transport
+ deliver()
Product
Truck
...
+ deliver()
Ship
...
+ deliver()
Deliver by land in a box.
Deliver by sea in a container.

Abstract factory

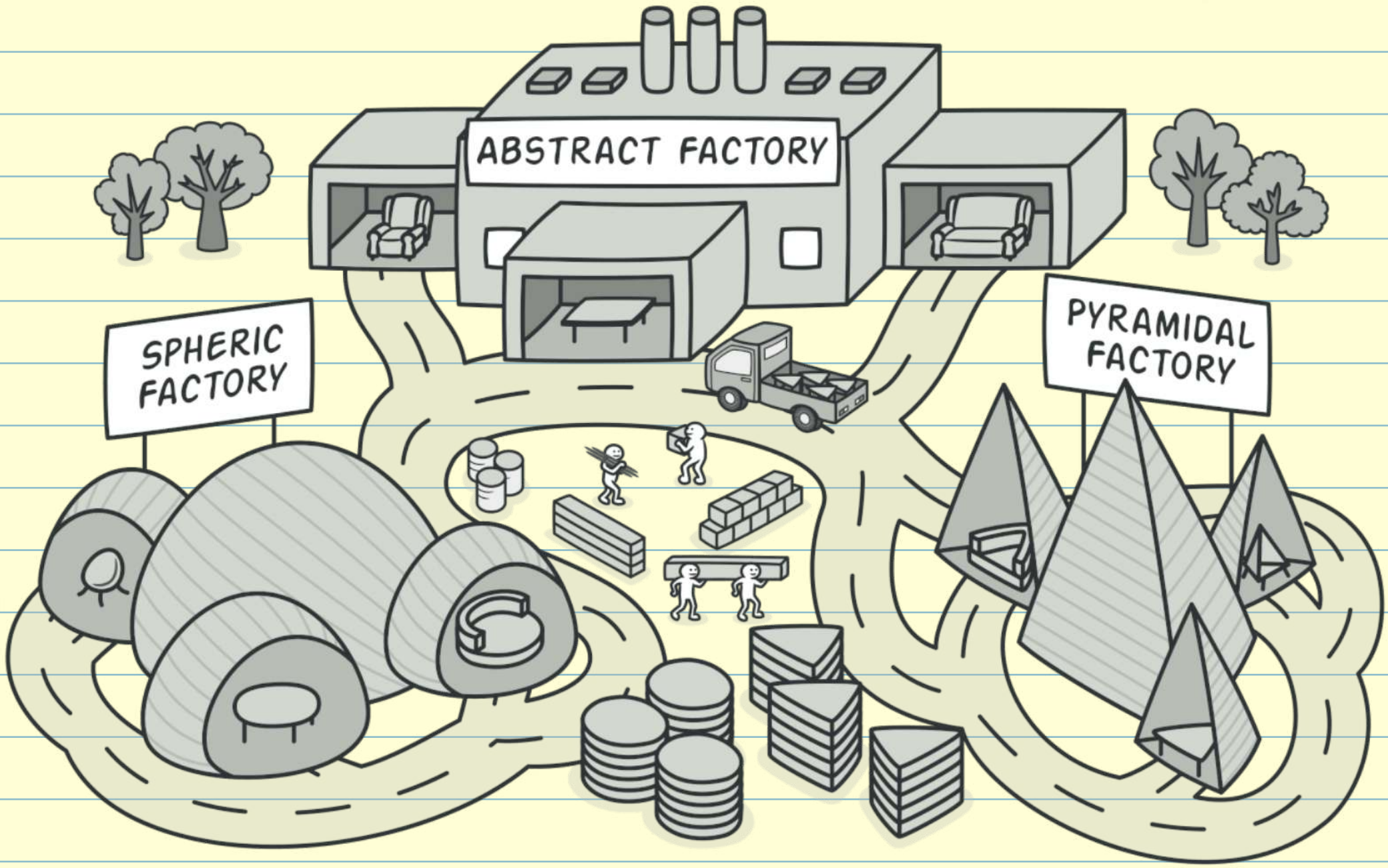


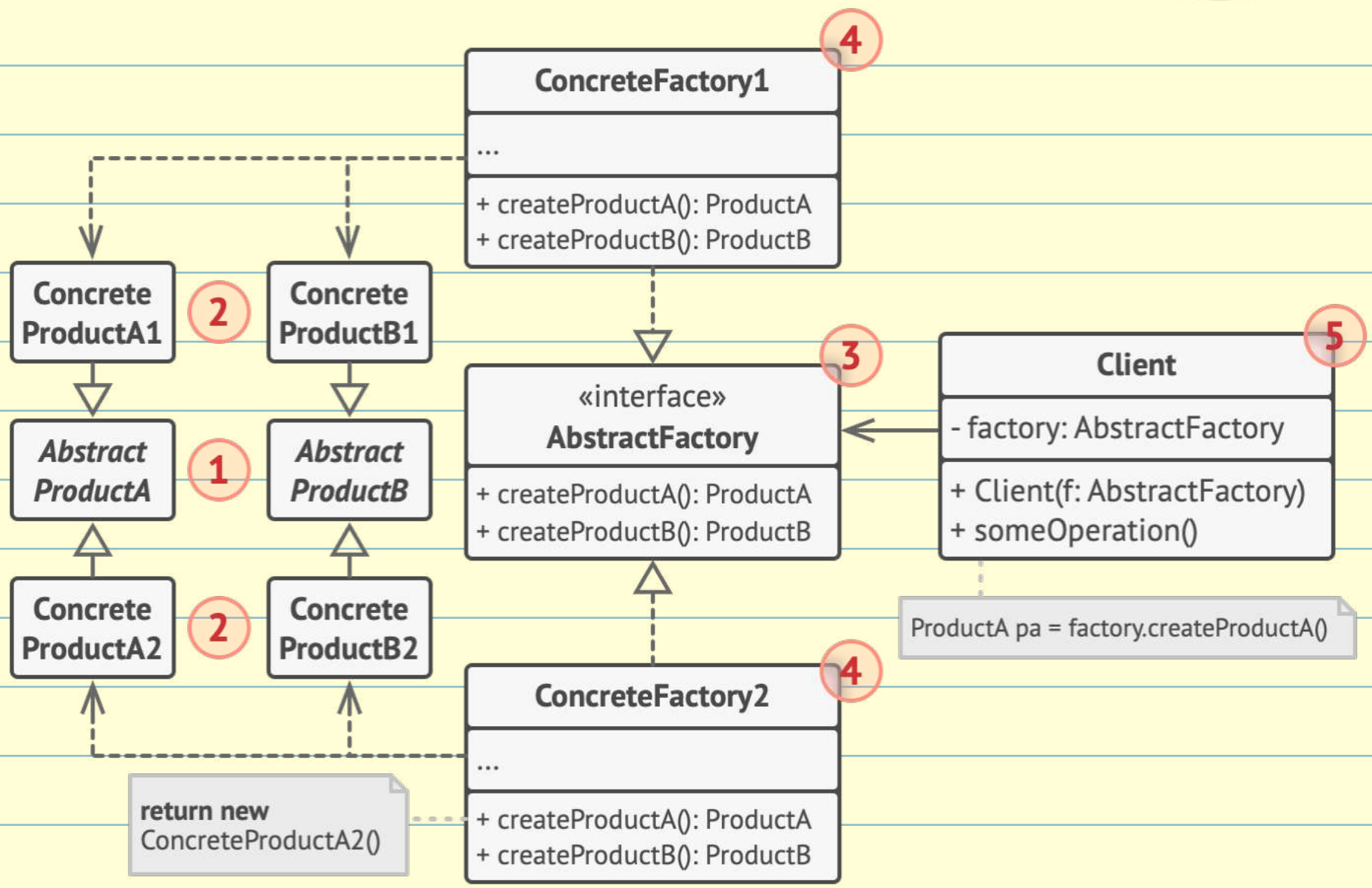
- Abstract factory is a creational design pattern that lets you produce families of related objects without specifying their concrete class.

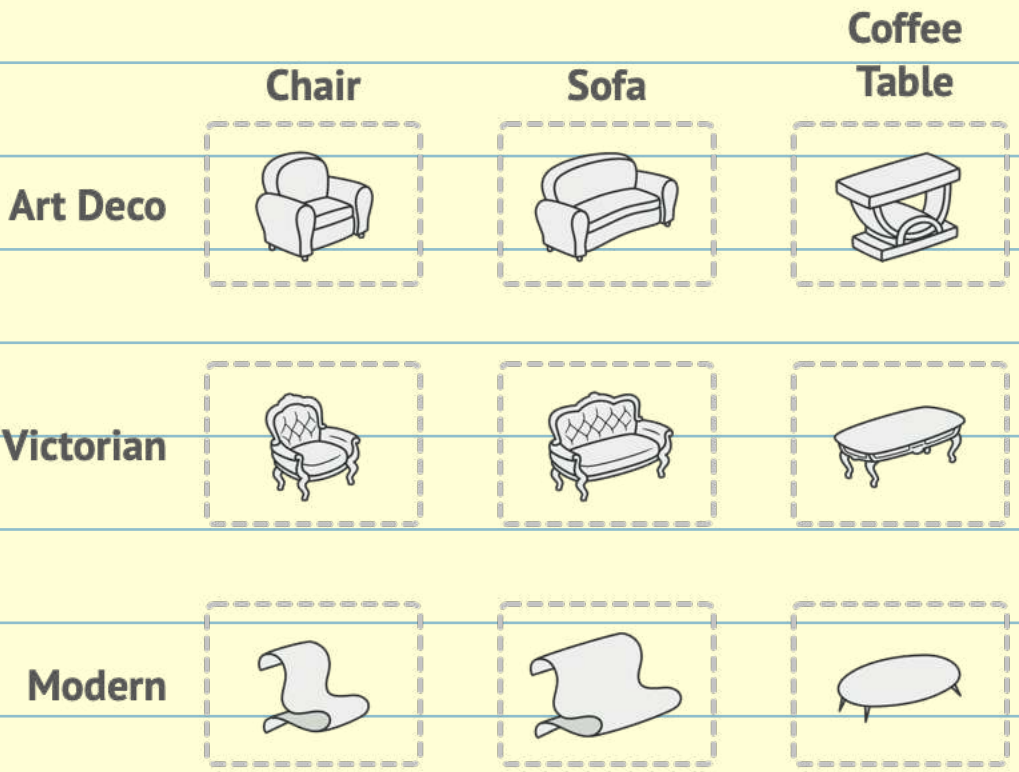
- use the abstract factory when you needs to work with various families of related products, but you don't want to depend on the concrete of those products - they might be unknown beforehand or you simply want to allow for future extensibility.

- Consider implementing the abstract factory when you have a class with a set of factory methods that blur its primary responsibility.

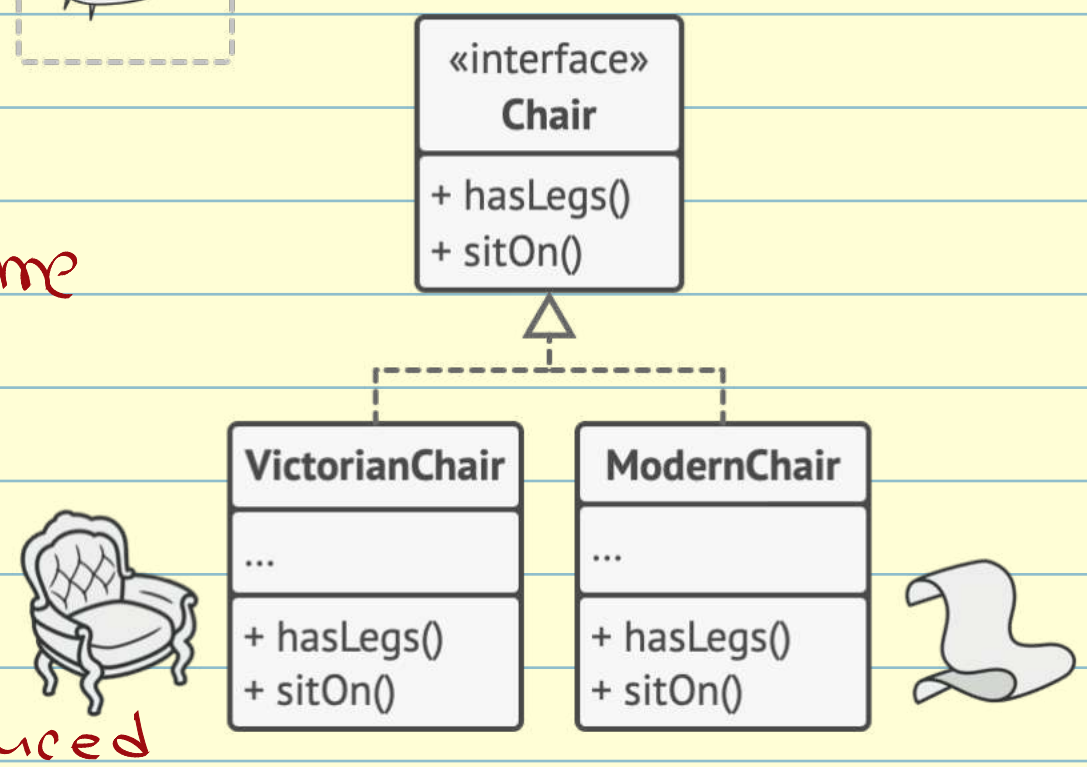
- avoid tight coupling between concrete products and client code.
- single responsibility principle.
- open close principle.

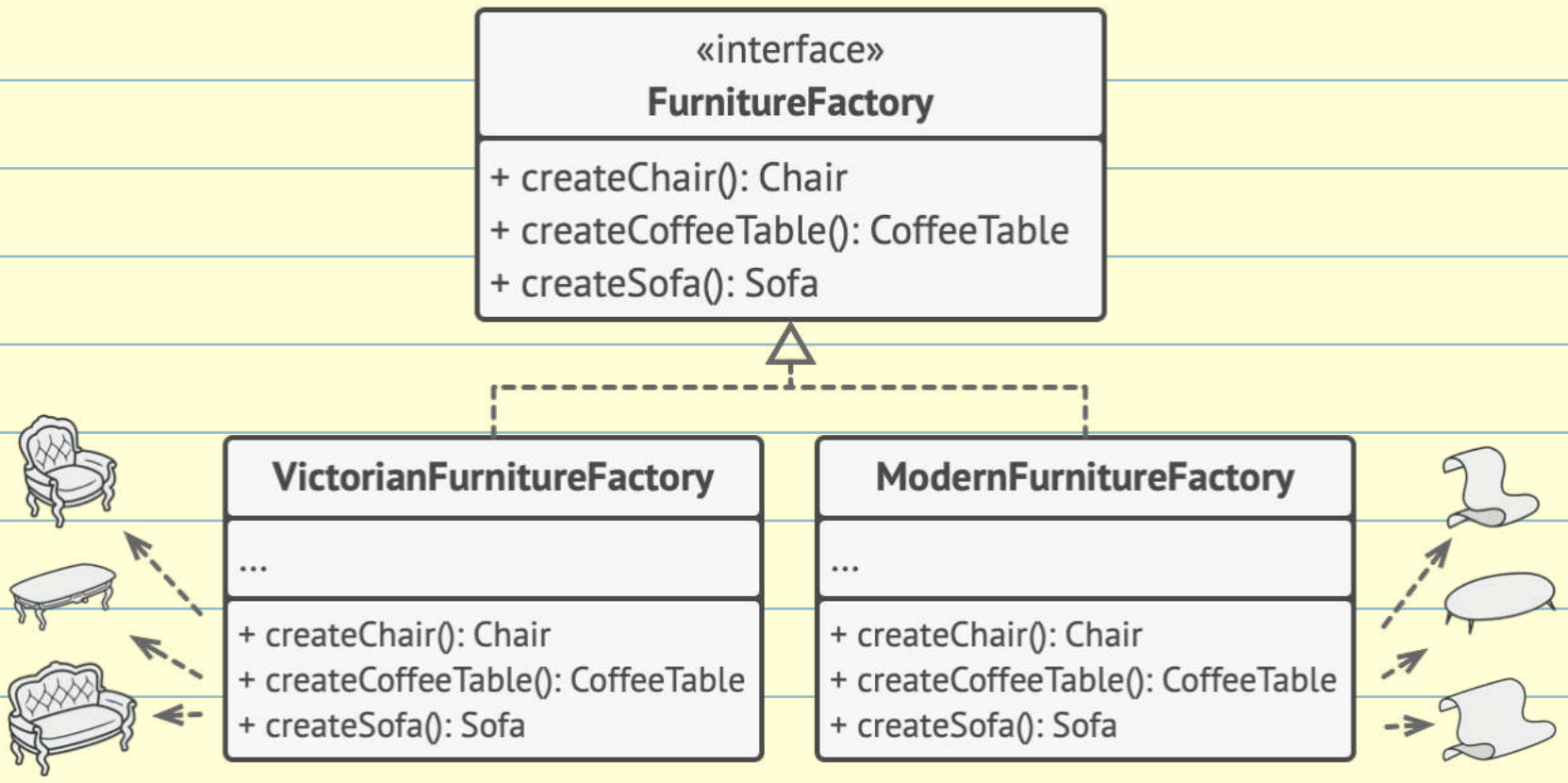

ABSTRACT FACTORY
SPHERIC FACTORY
PYRAMIDAL FACTORY


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
Concrete ProductA1
2
Concrete ProductB1
Abstract ProductA
1
Abstract ProductB
Concrete ProductA2
2
Concrete ProductB2
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
3
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
5
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
4
return new ConcreteProductA2()



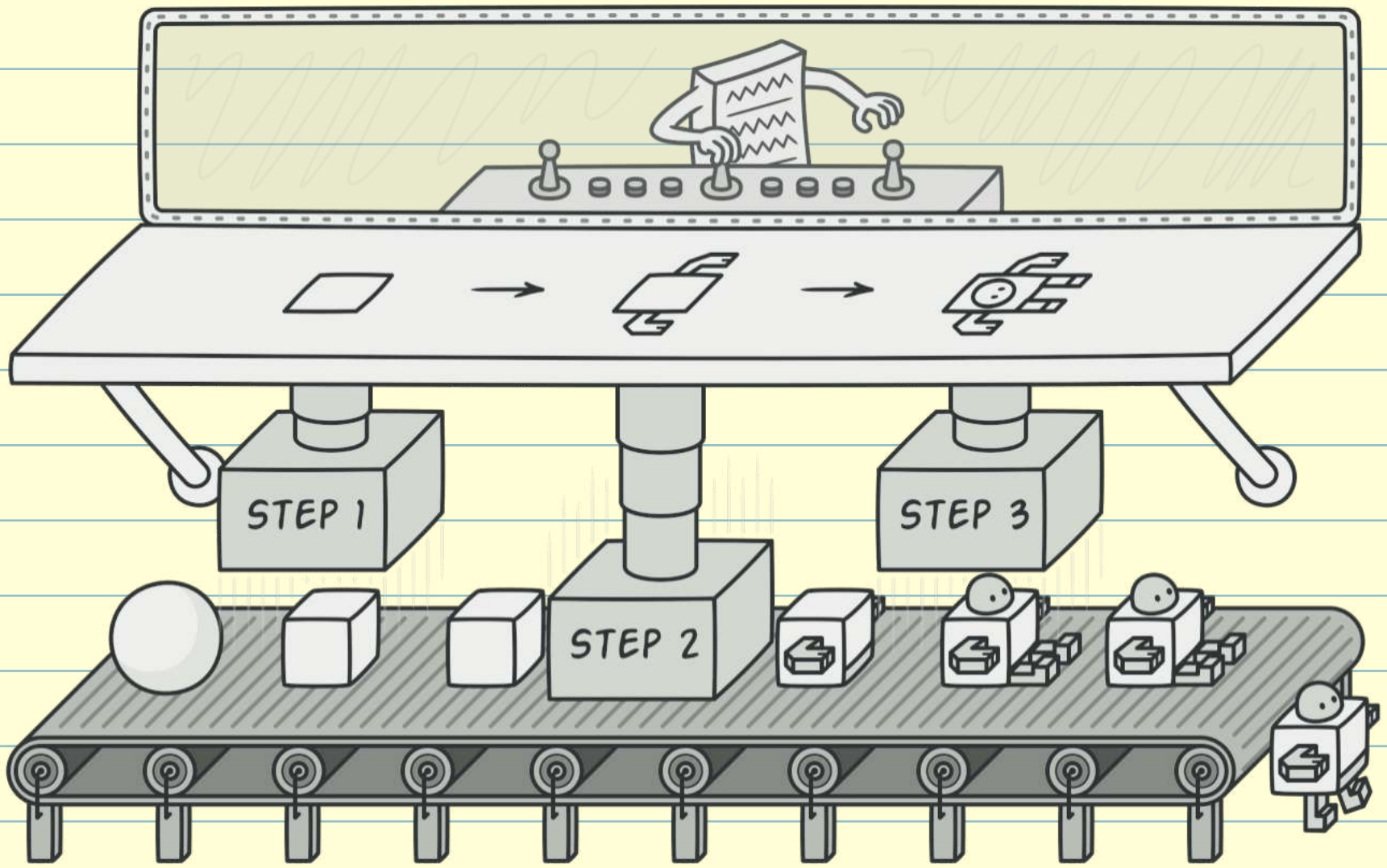
The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

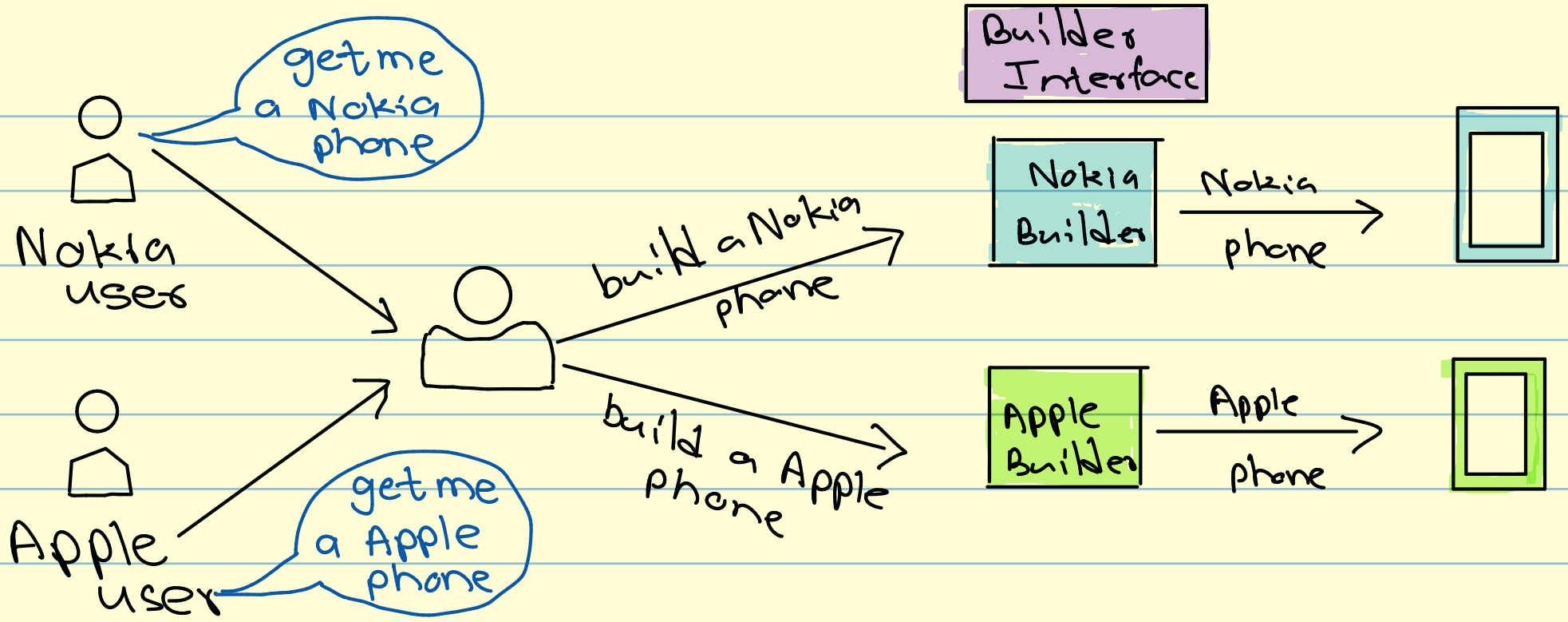




Builder

- Creational design pattern.
- lets you construct complex objects step by step.
  The pattern allows you to produce different types and representations of an objects using the same construction code.

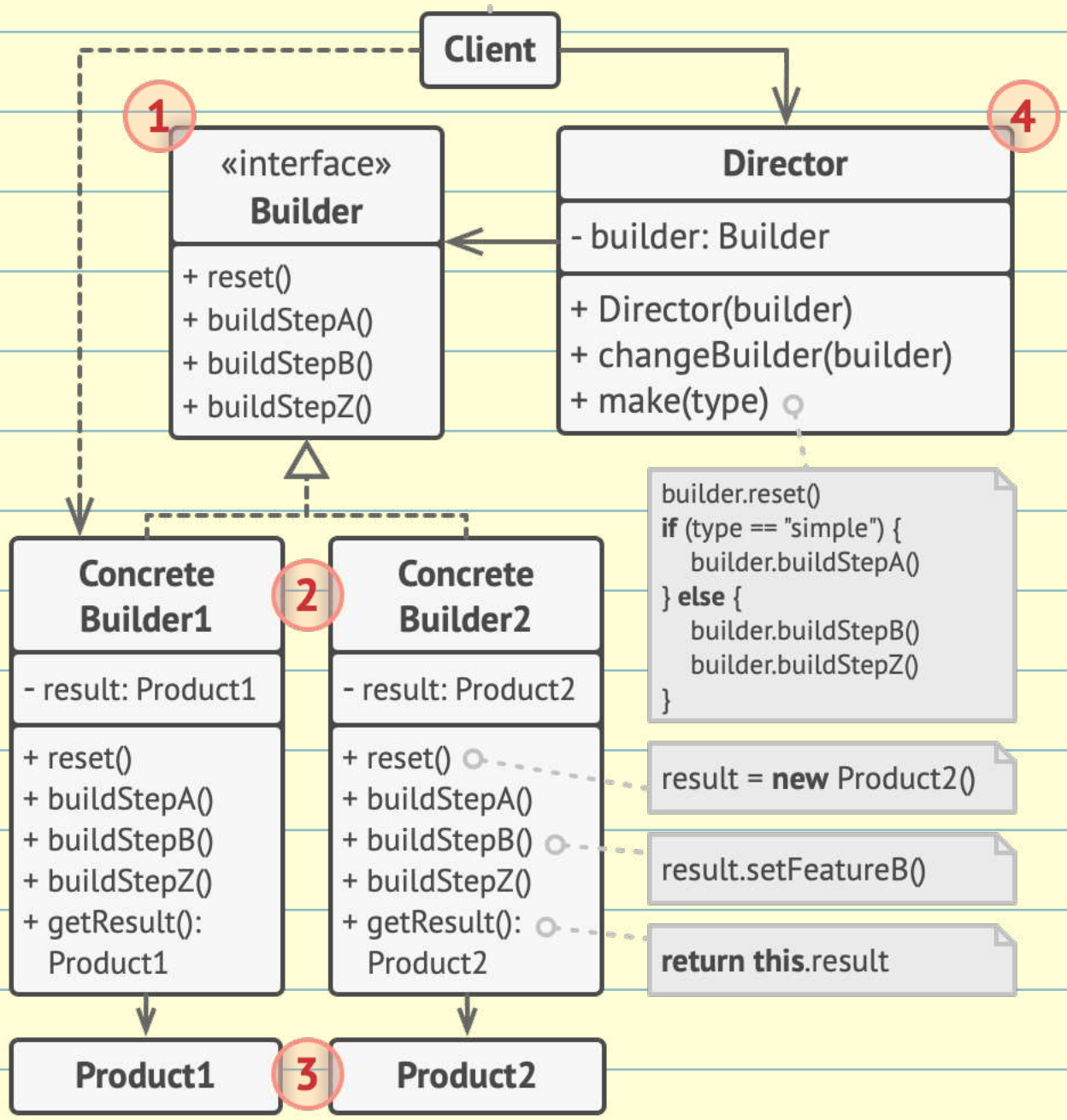


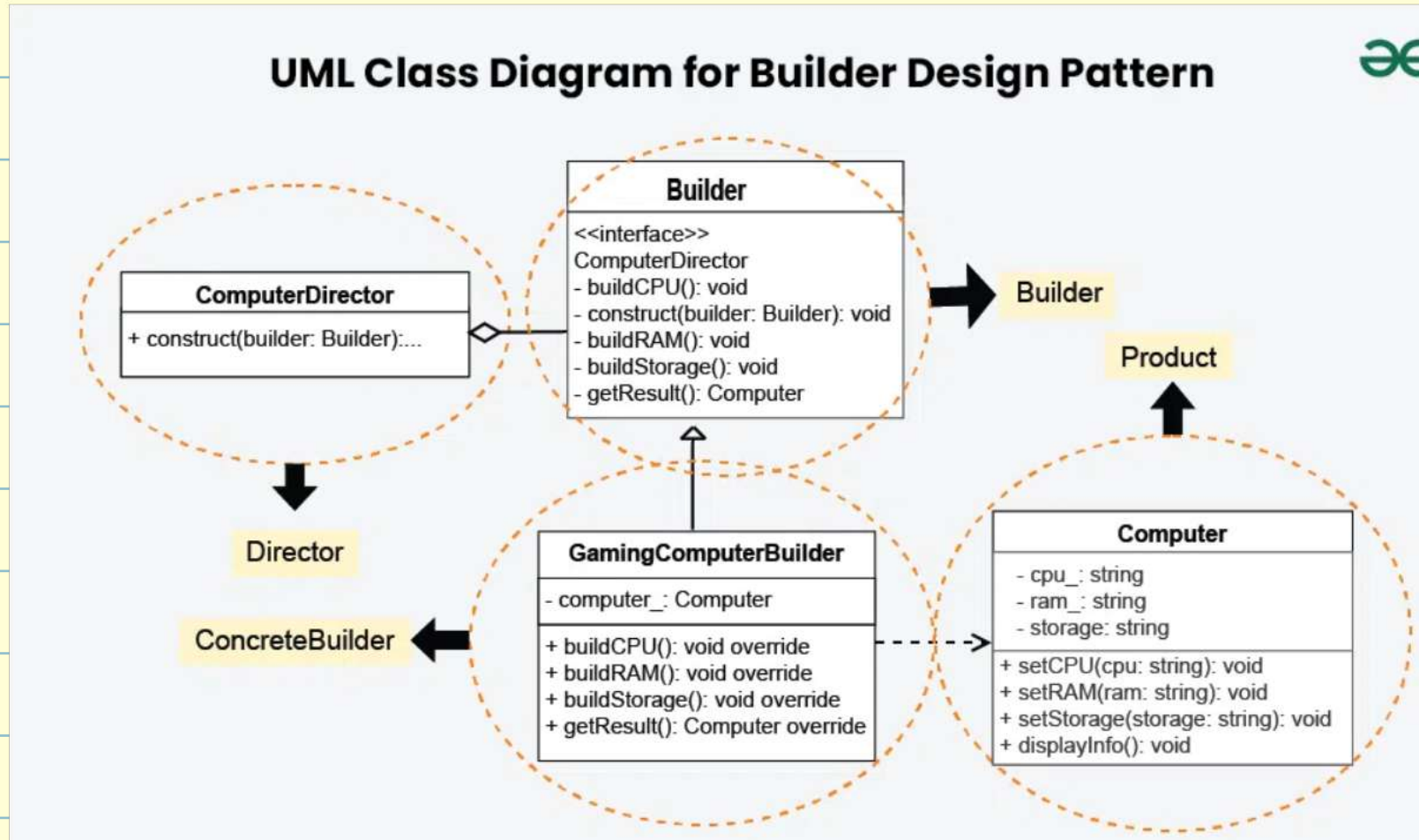


Client Code	Director code	Builder Code	Product

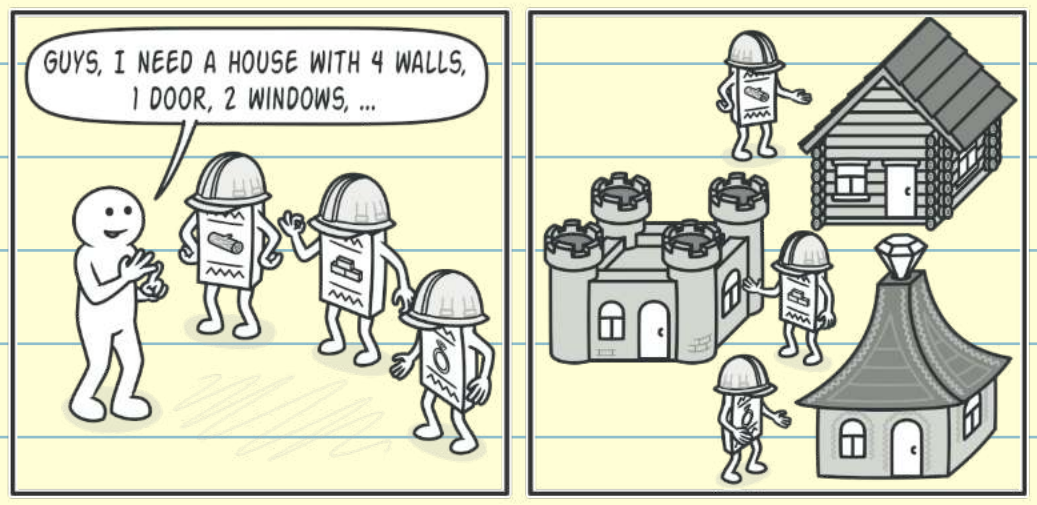
```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

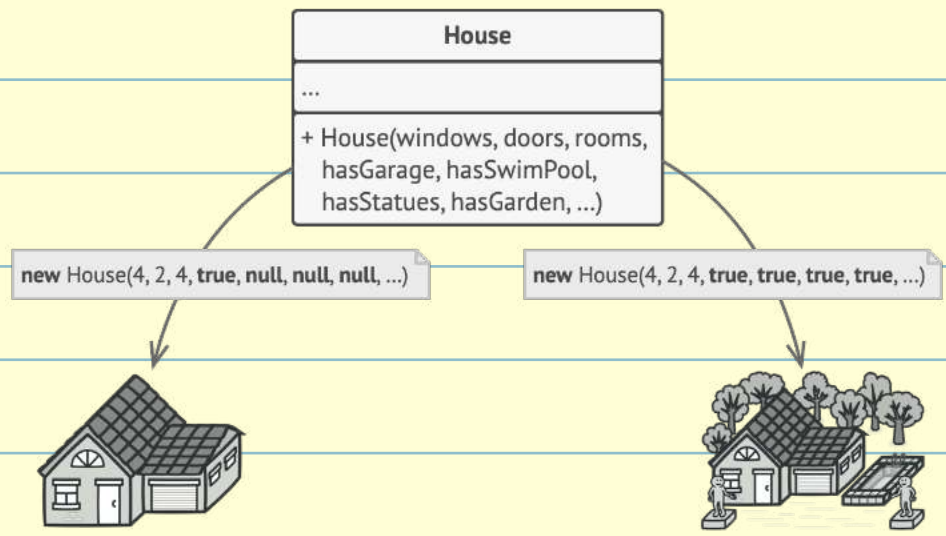
5





- use the builder pattern to get rid of a telescoping constructor.

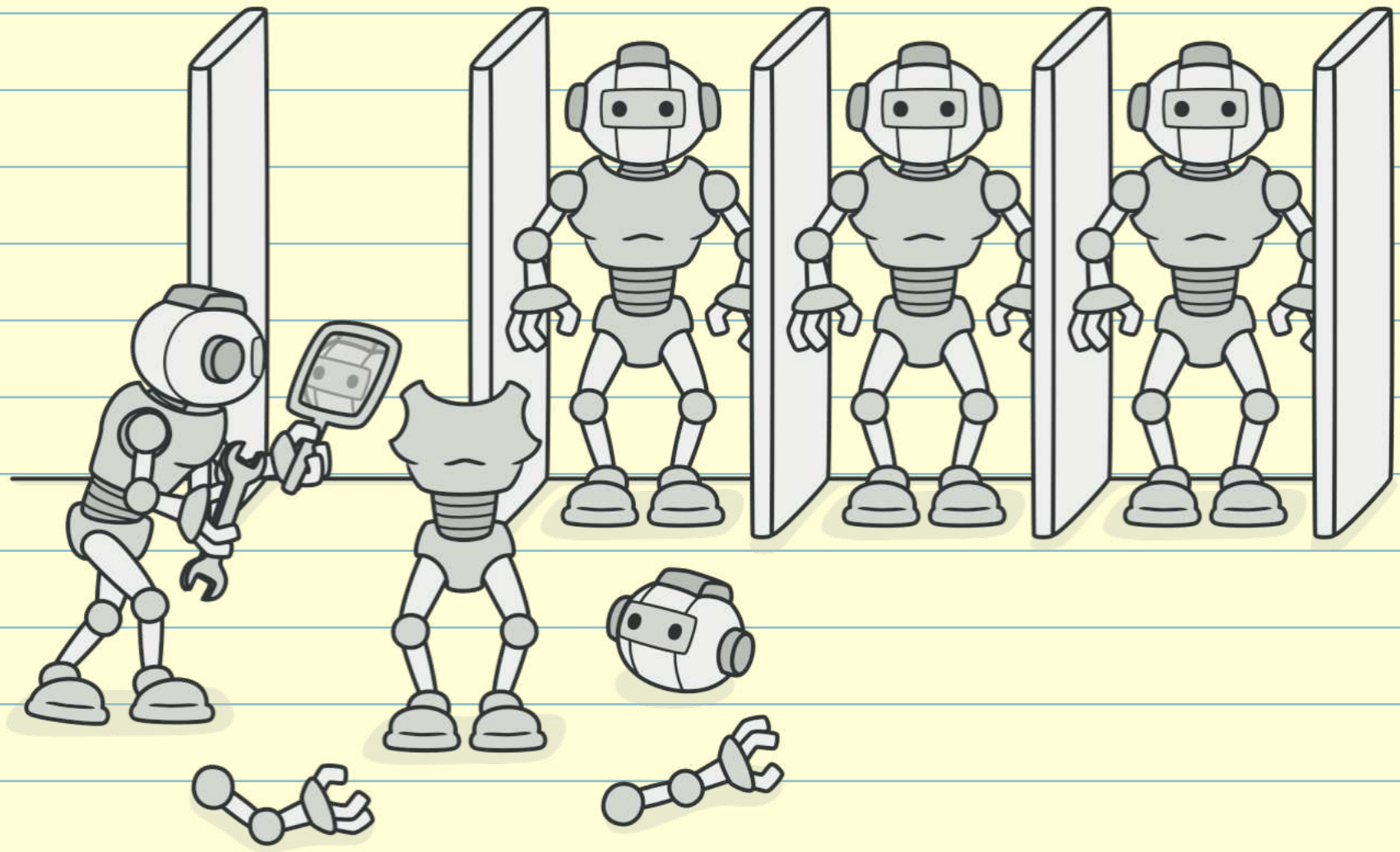




- use the builder pattern when you want your code to be able to create different representations of the same product.
- you can reuse the same construction code when building various representations of products.

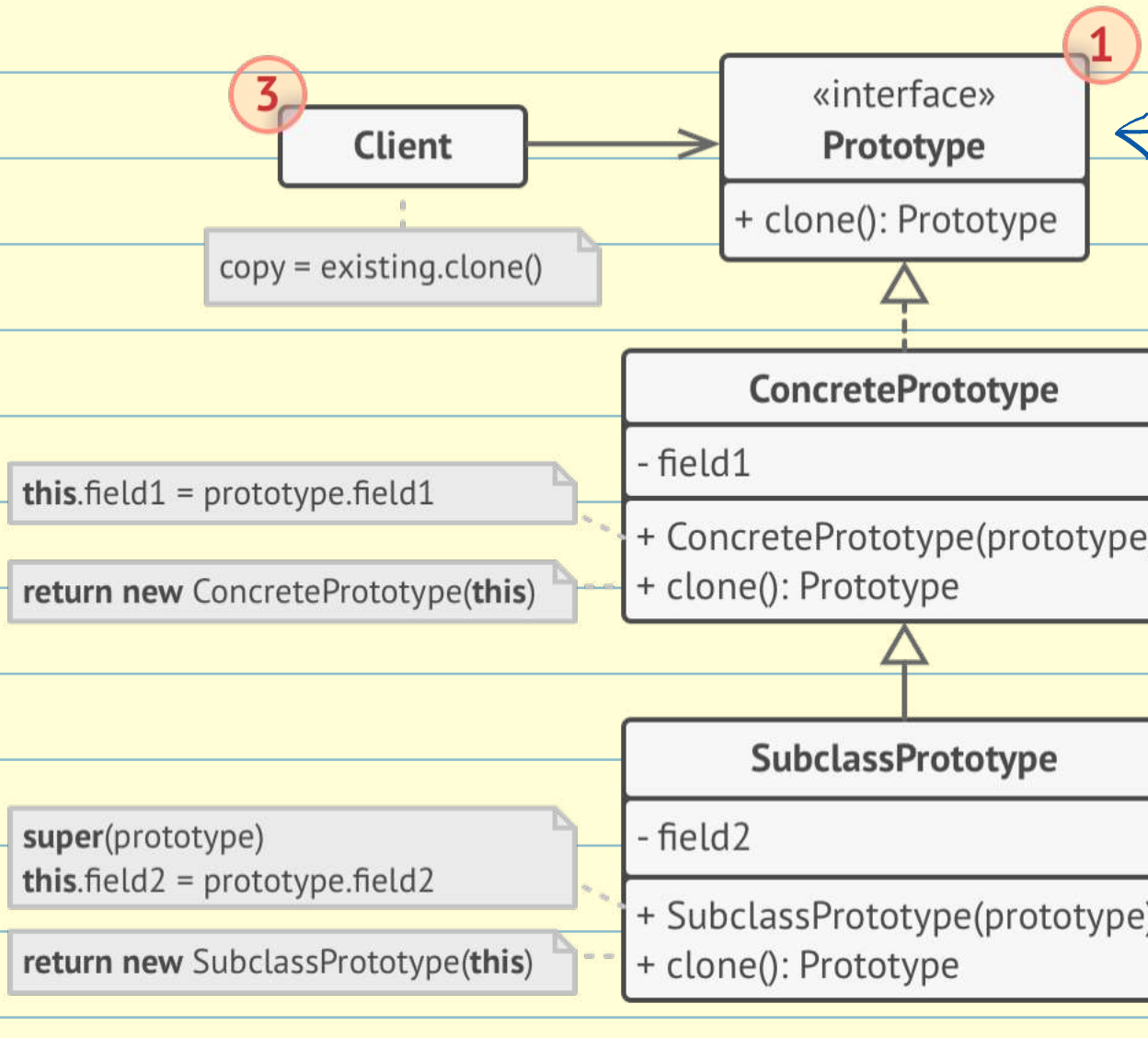
Prototype

- Creational design pattern
- allows cloning objects, even complex ones, without coupling to their specific classes.



- an object that support cloning is called a prototype.

- will save resources & time
- will get rid of repeated initialization code

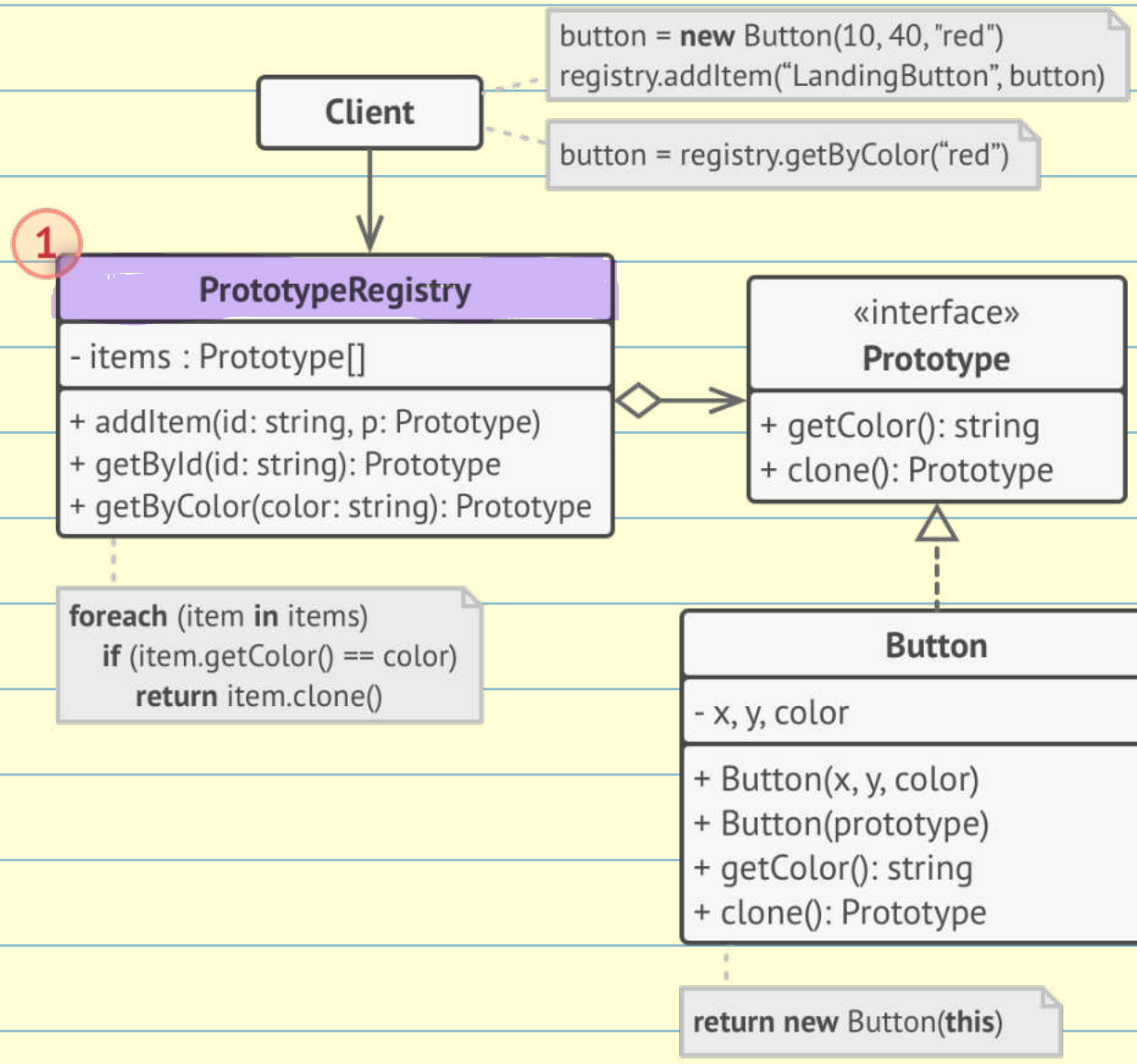


abstract class or interface

- declare the clone() method that all prototypes must implement.

define how the cloning process should be caried out.

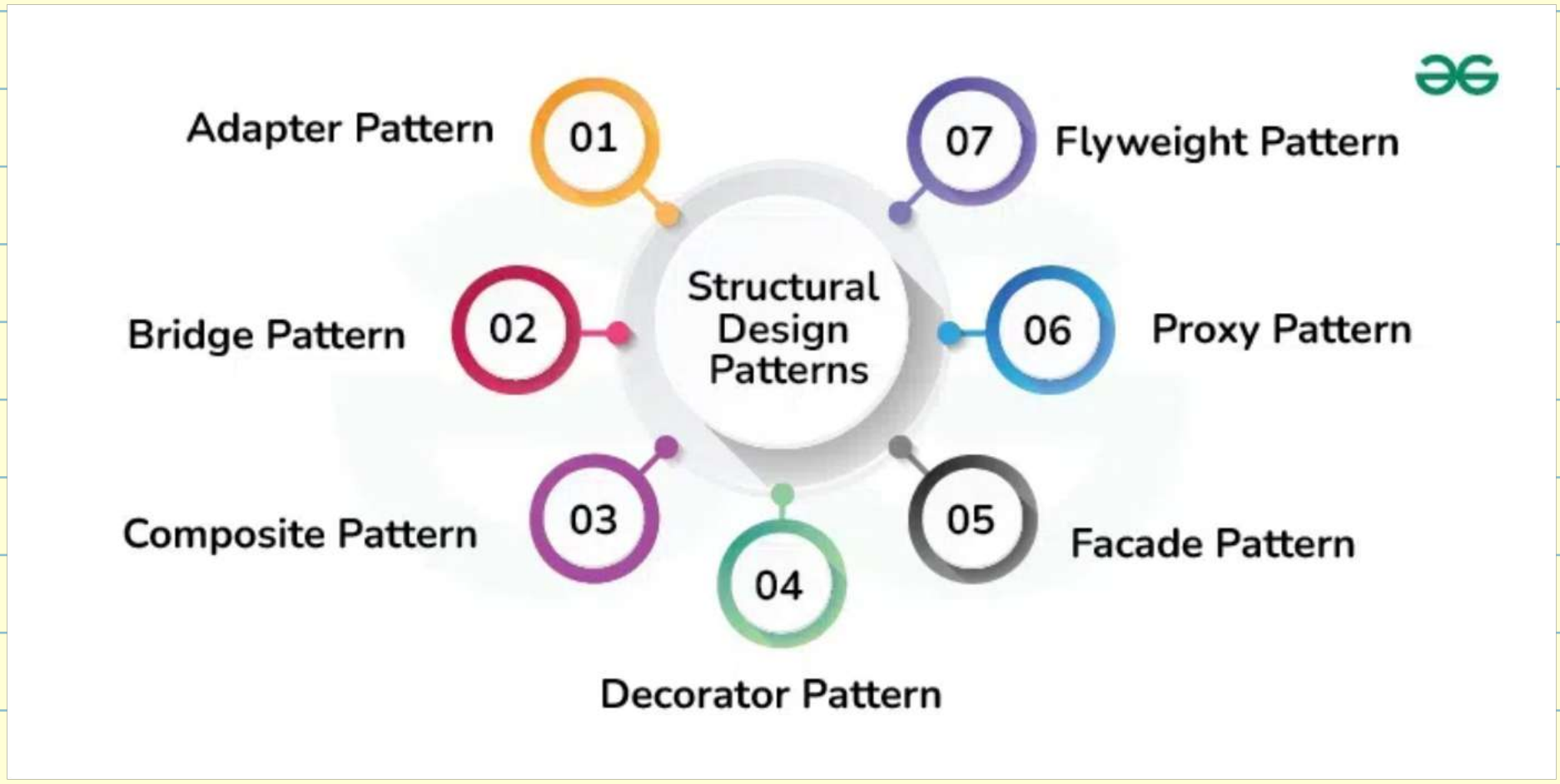
Copy → deep copy
↳ shallow copy



Prototype Registry provides an easy way to access frequently used prototypes.
The simplest prototype registy is a hash map

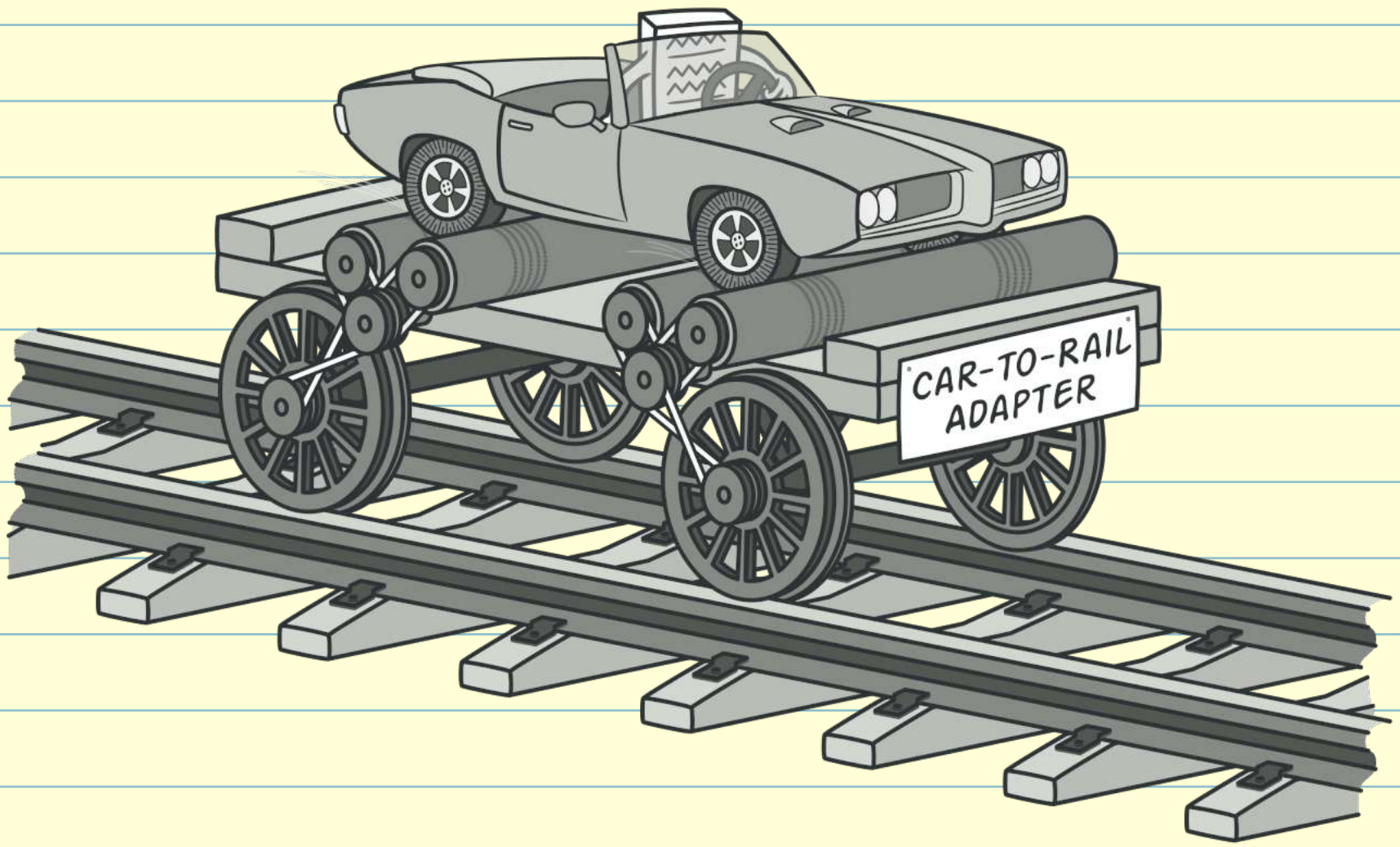
String → prototype

Structural Design Patterns

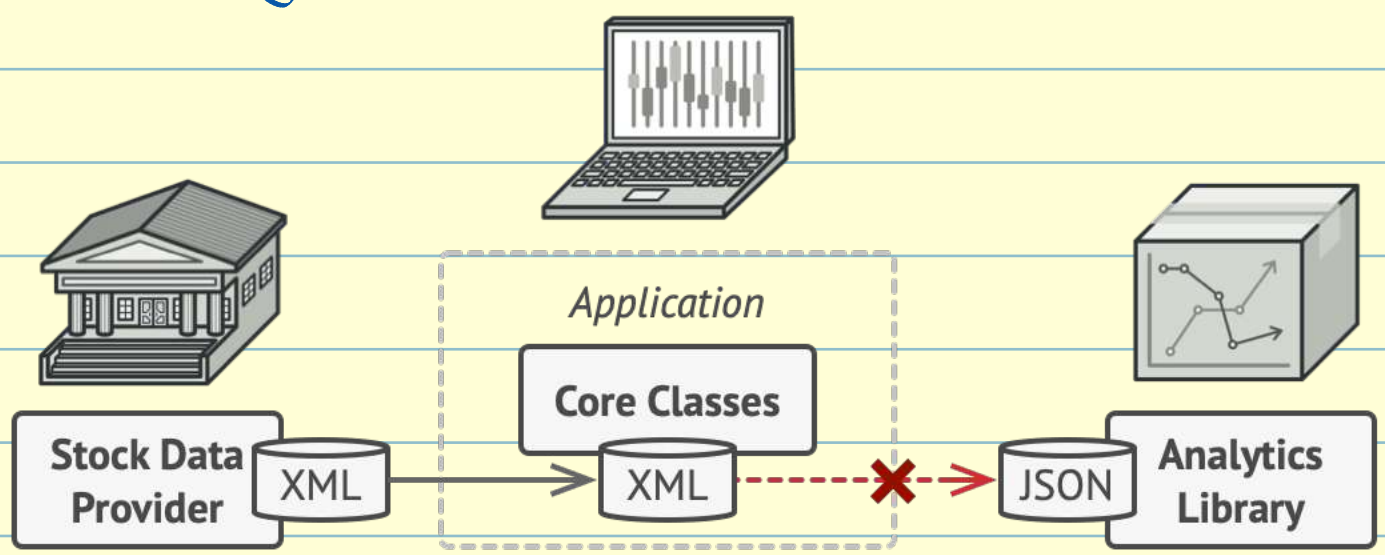


Adapter

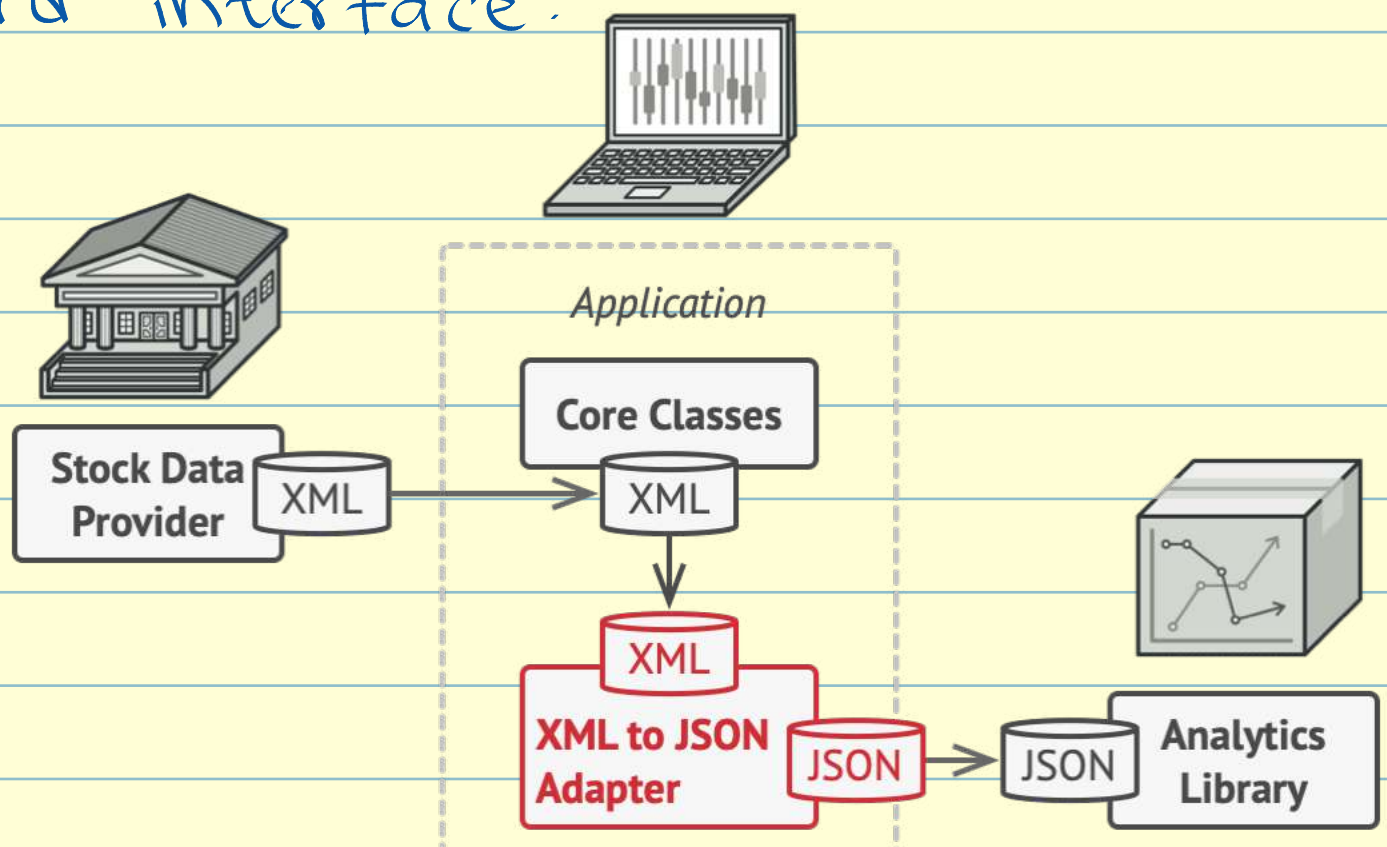
- also known as : Wrapper
- allows objects with incompatible interfaces.



- use the adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.



- The Adapter class let you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd party class or any other weird interface.

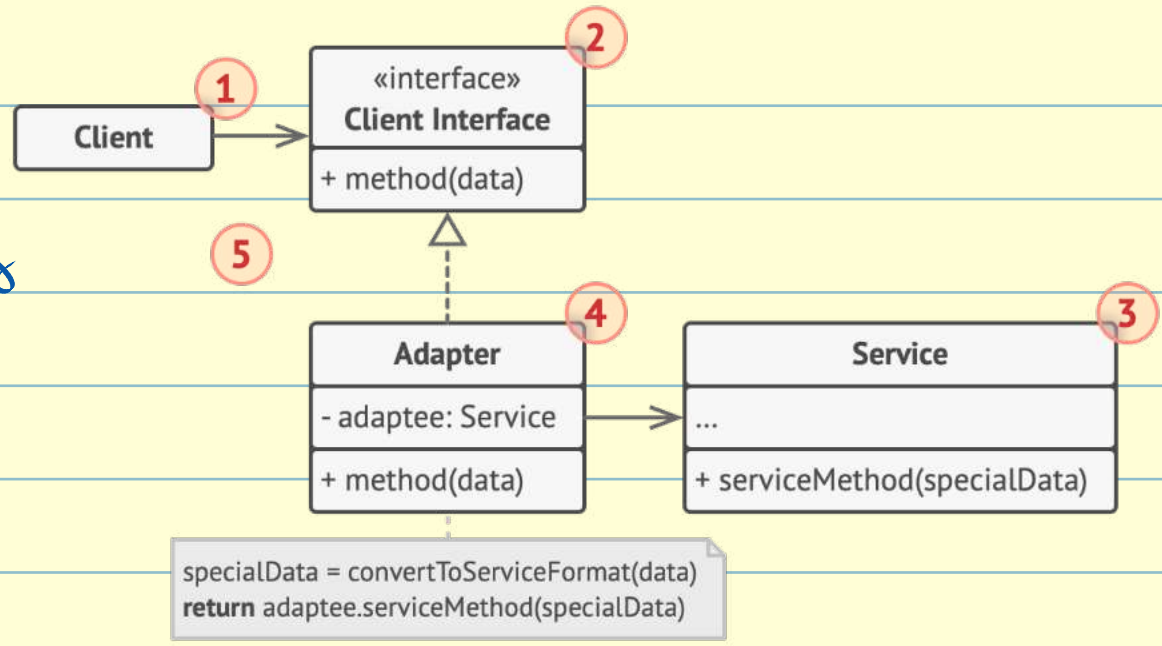


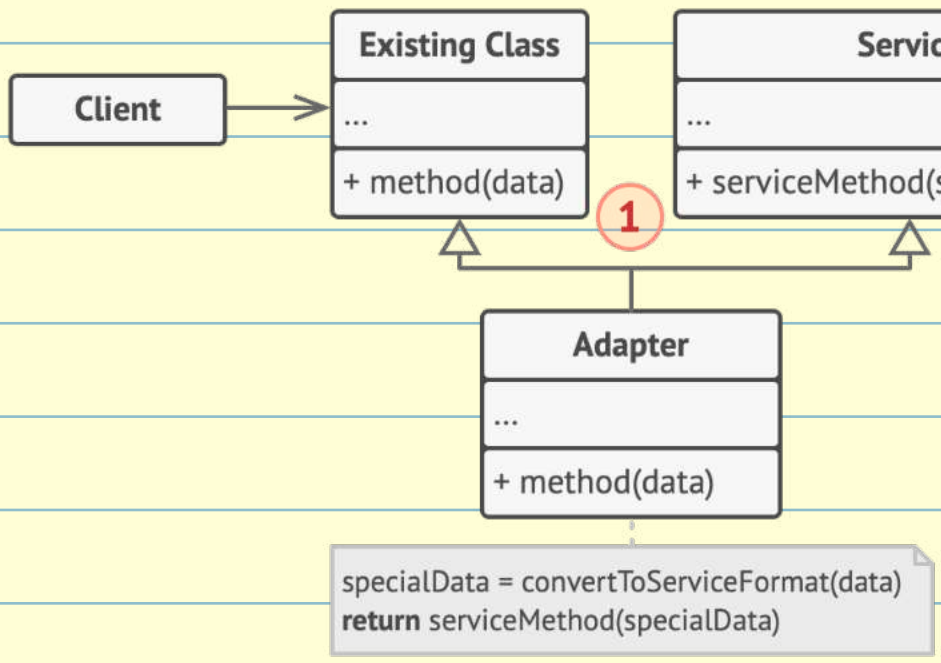
- single responsibility principle
  - can seperate the interface or data conversion code from the primary buisness logic of the program.

- open lclosed principle
  - can introduce new types of adapters intc the program without breaking existing client code, as long as they work with the adapter through the client interface.

Structure

object adapter

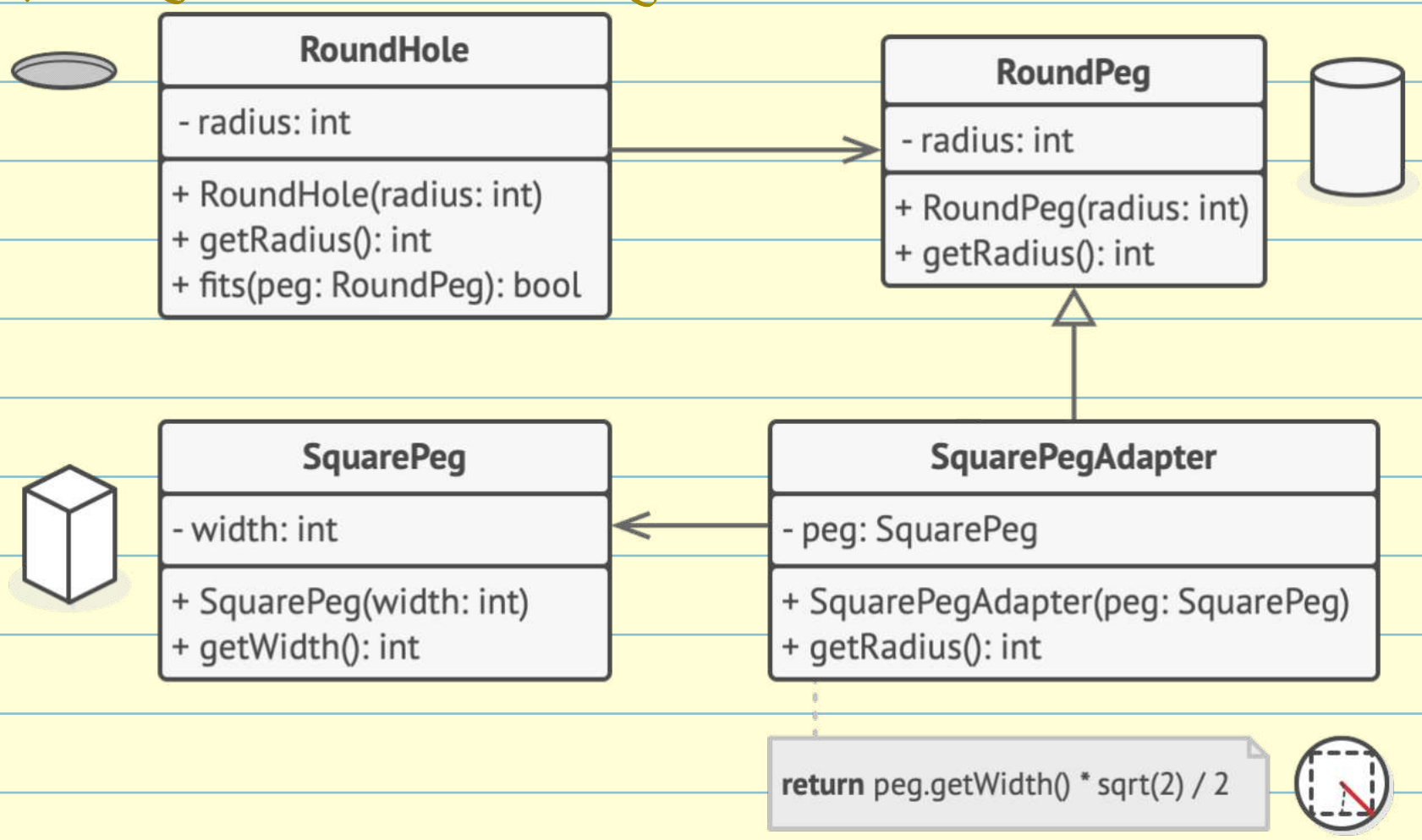




Class adapter

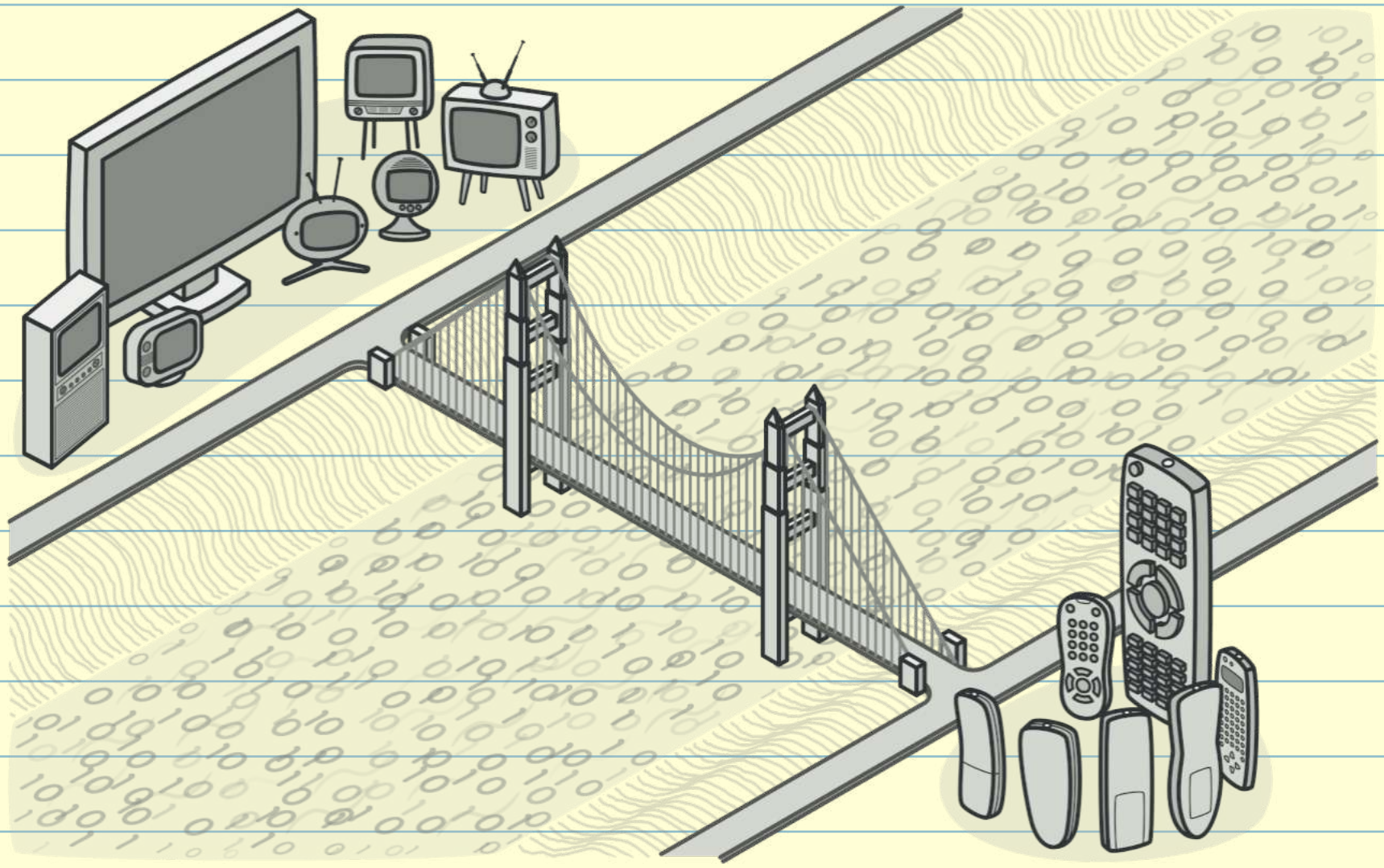
C can only be implemented in programming languages that support multiple inheritance, such as C++.

Adapting squre pegs to round holes



Bridge

- bridge is a structural design pattern that lets you splits a large class or a set of closely related classes into two seperate hierarchies - abstraction and implementation - which can be developed independently of eachother.

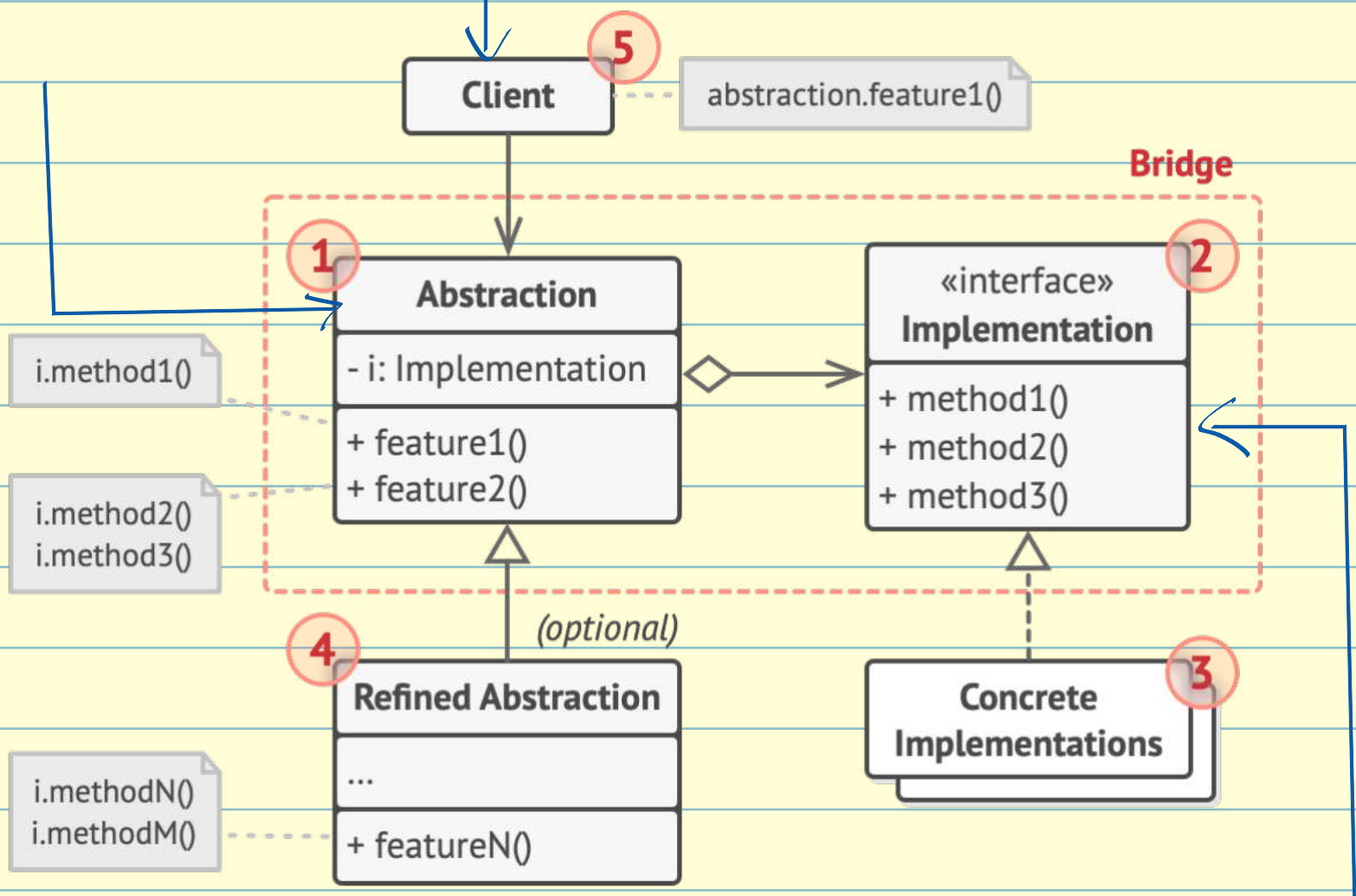


- allows you to seperate the abstraction from the implementation.
- There are 2 parts in Bridge design pattern
  1. Abstraction
  2. Implementation.

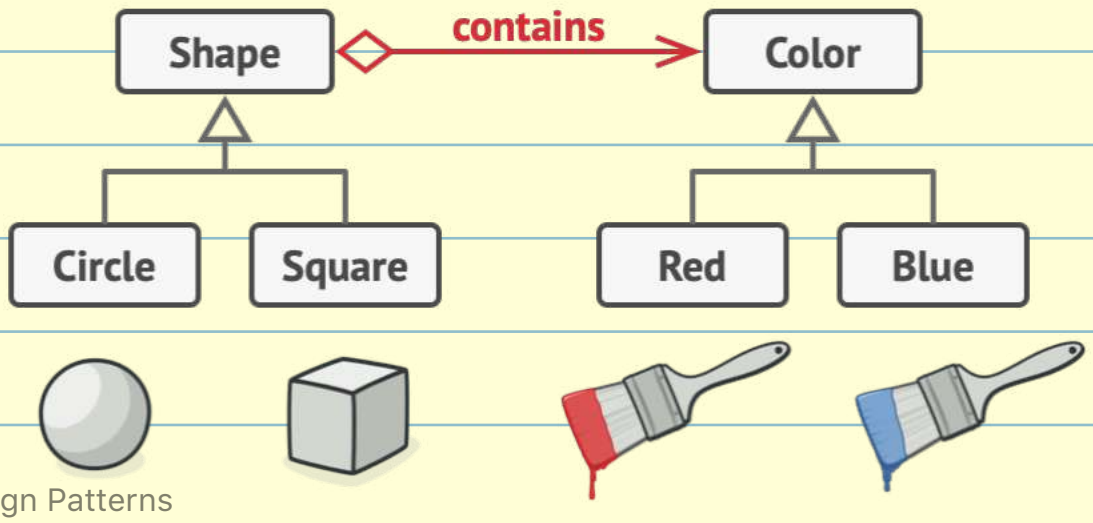
structure

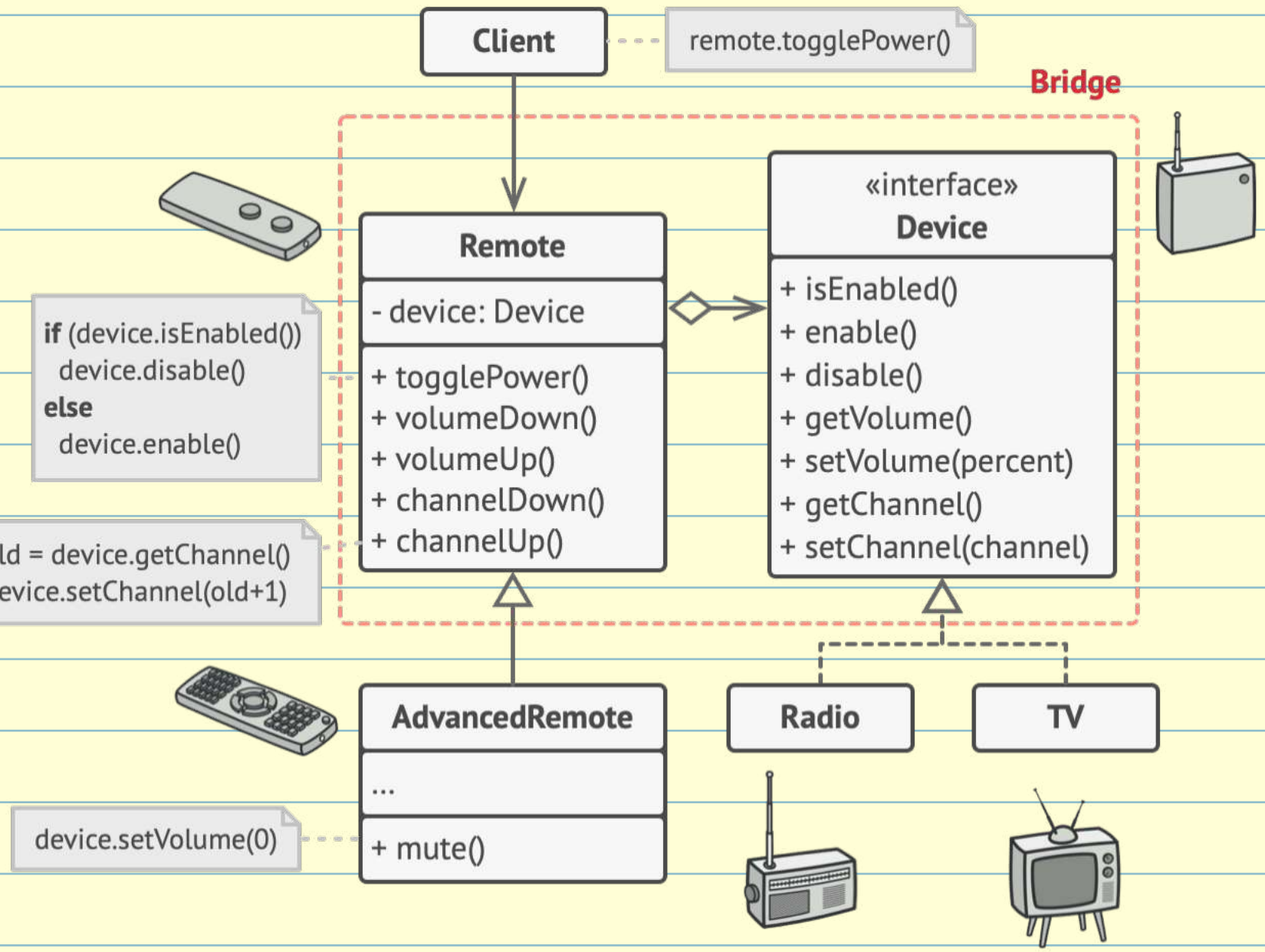
abstraction provides high-level control logic.

clients job is to link the abstraction object with one of implementation object.



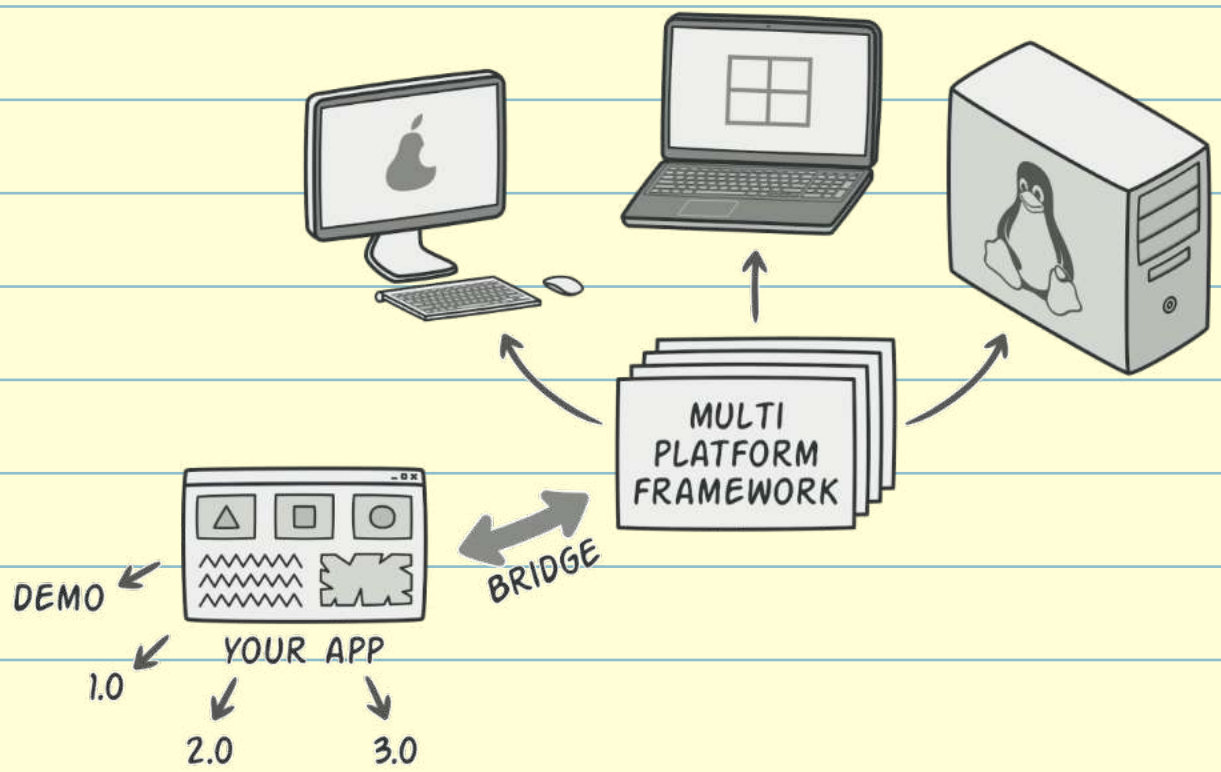
The implementation declares the interface that's common for all concrete implementations.





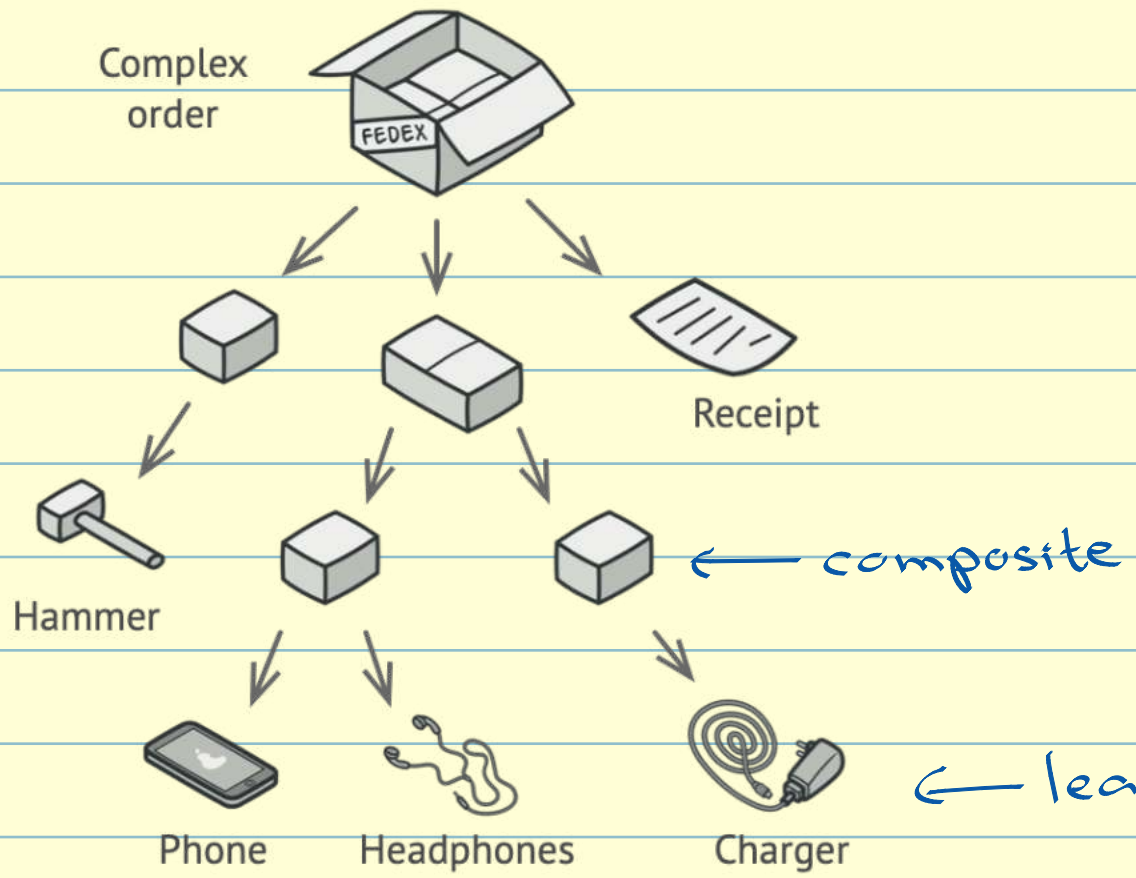
- Open / Closed principle
- Single Responsibility Principle.

- used mainly for implementing platform independence features.



Composite

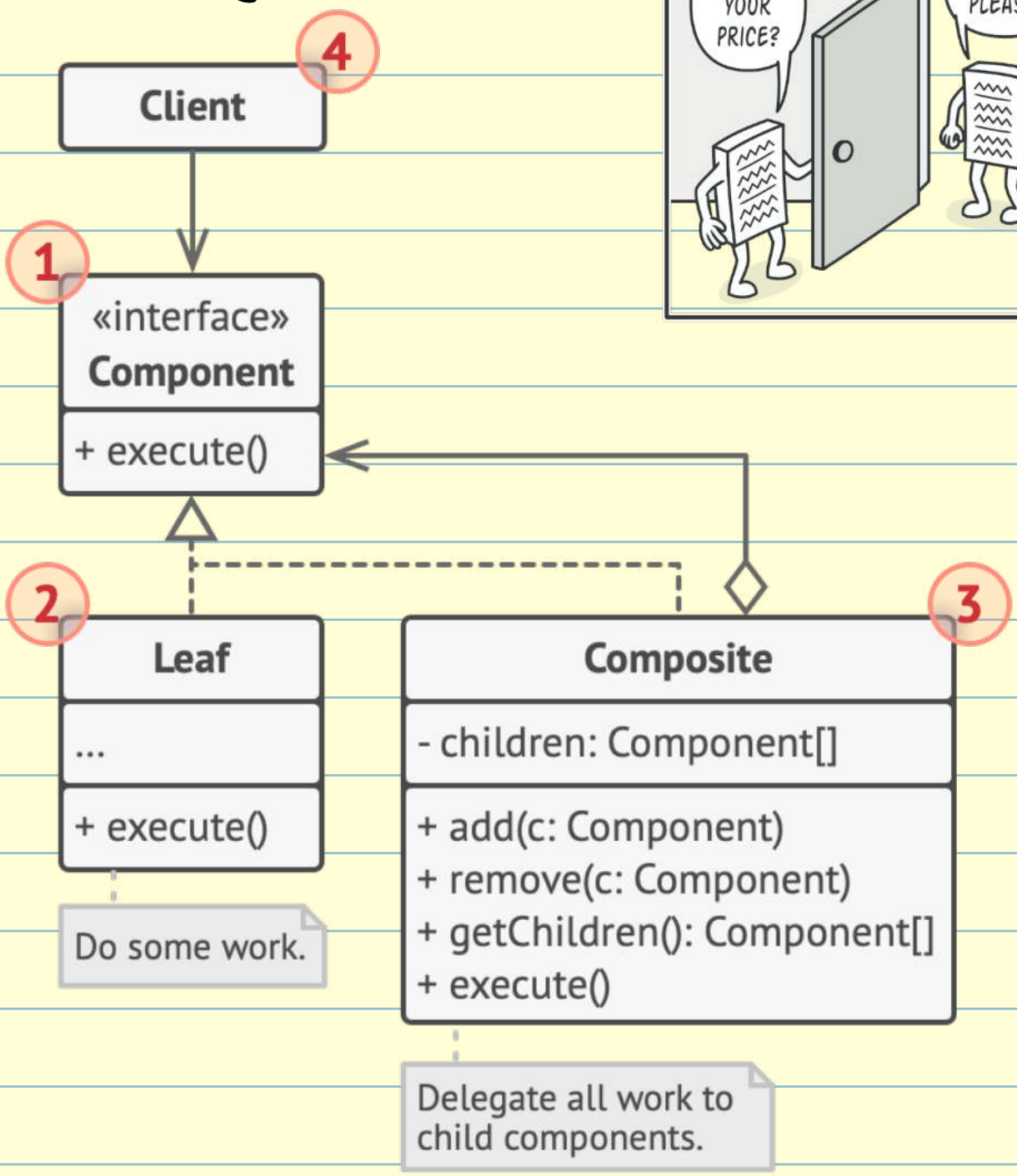
- lets you compose objects into tree structures and then work with these structures as if they were individual objects

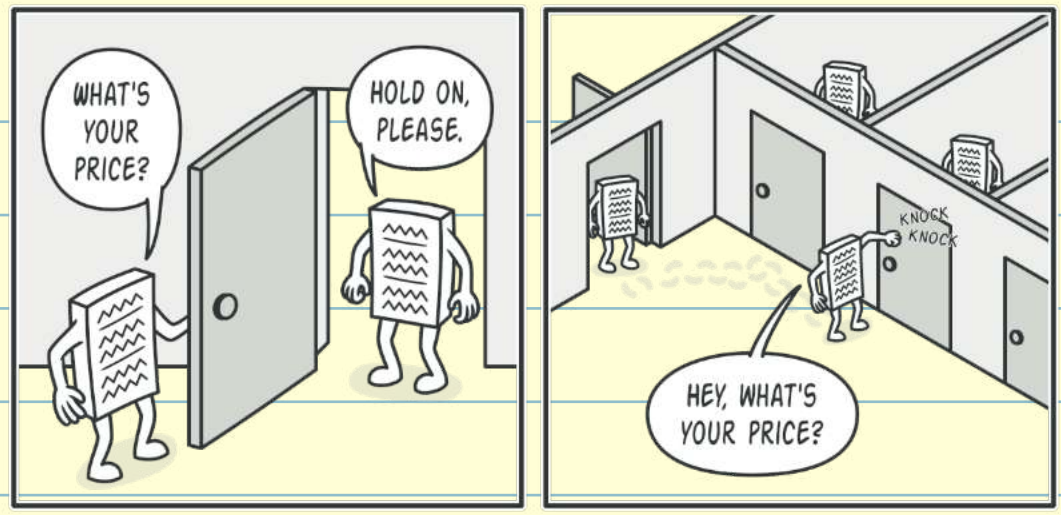


Key components in the composite pattern

1. Component - common interface for all objects.
   defines the common methods for both leaf and composite
2. Leaf
3. Composite - the container object that can hold leaf objects as well as other composite objects.
4. Client

Structure

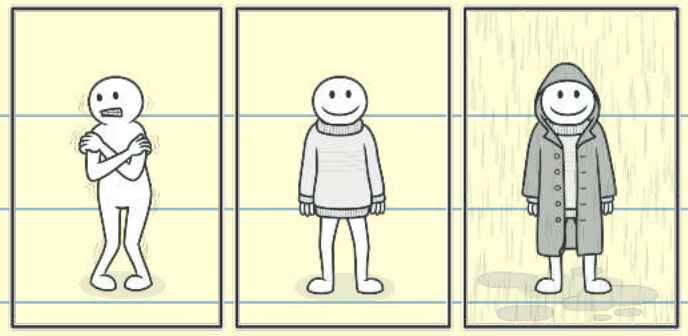




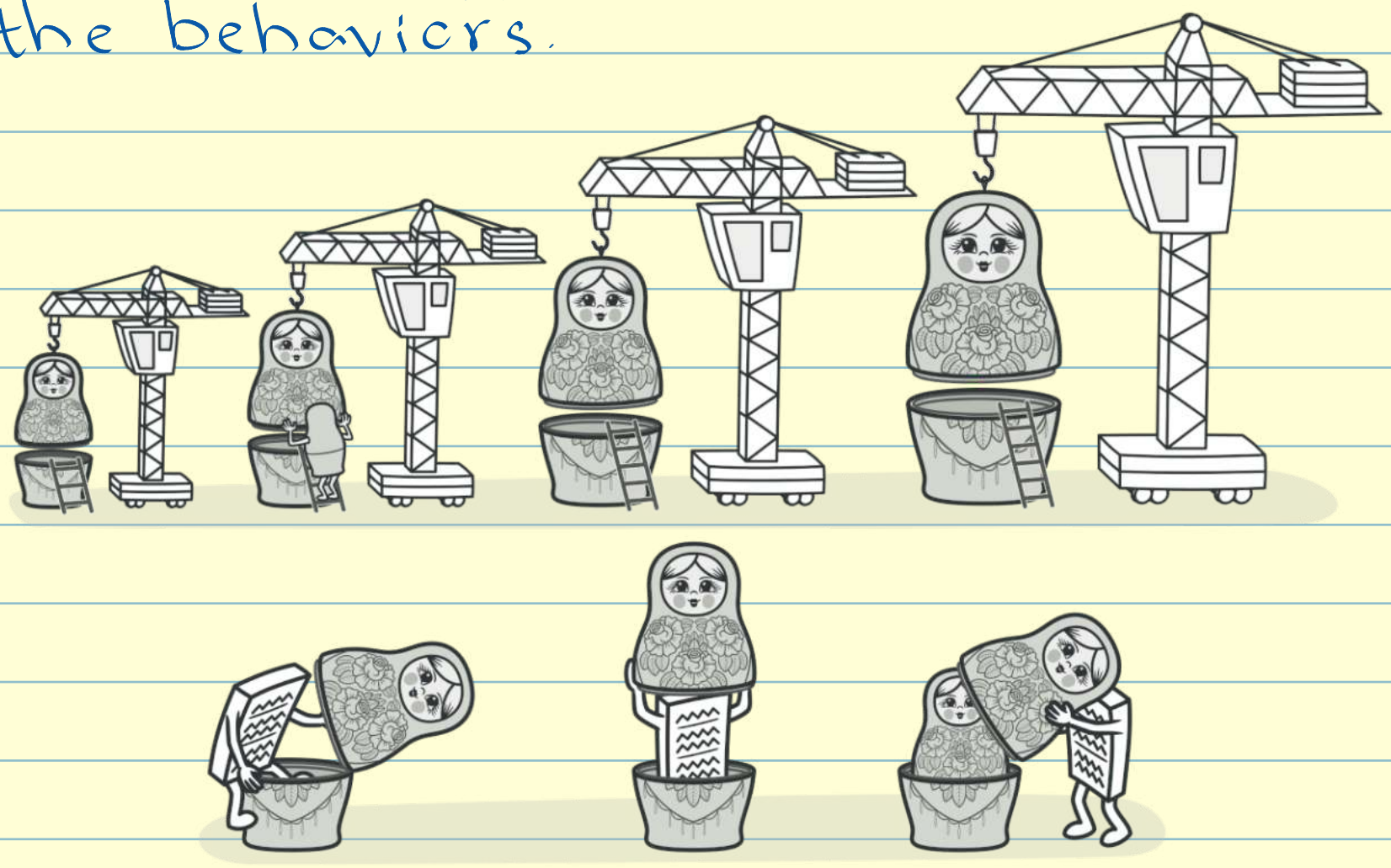
Advantages

- hierarchical structure
- simplified client code
- flexibility ; Callows add or remove objects without affecting the client code)
- Scalability
- Code reusability

Decorator



- structural design pattern
- lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



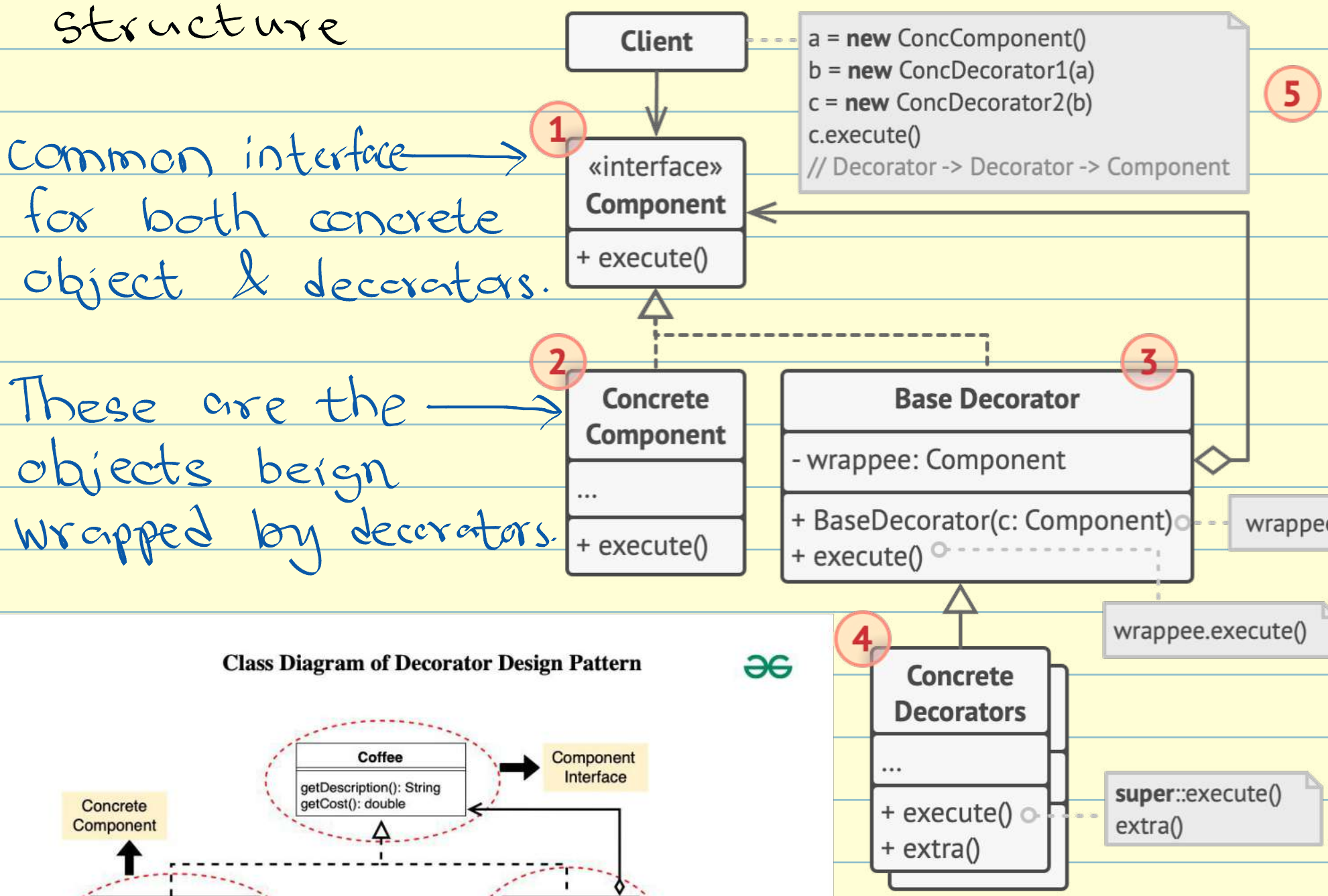
- use decorator pattern when you need to be able to assign extra behaviours to objects at runtime without breaking the code that uses these objects.

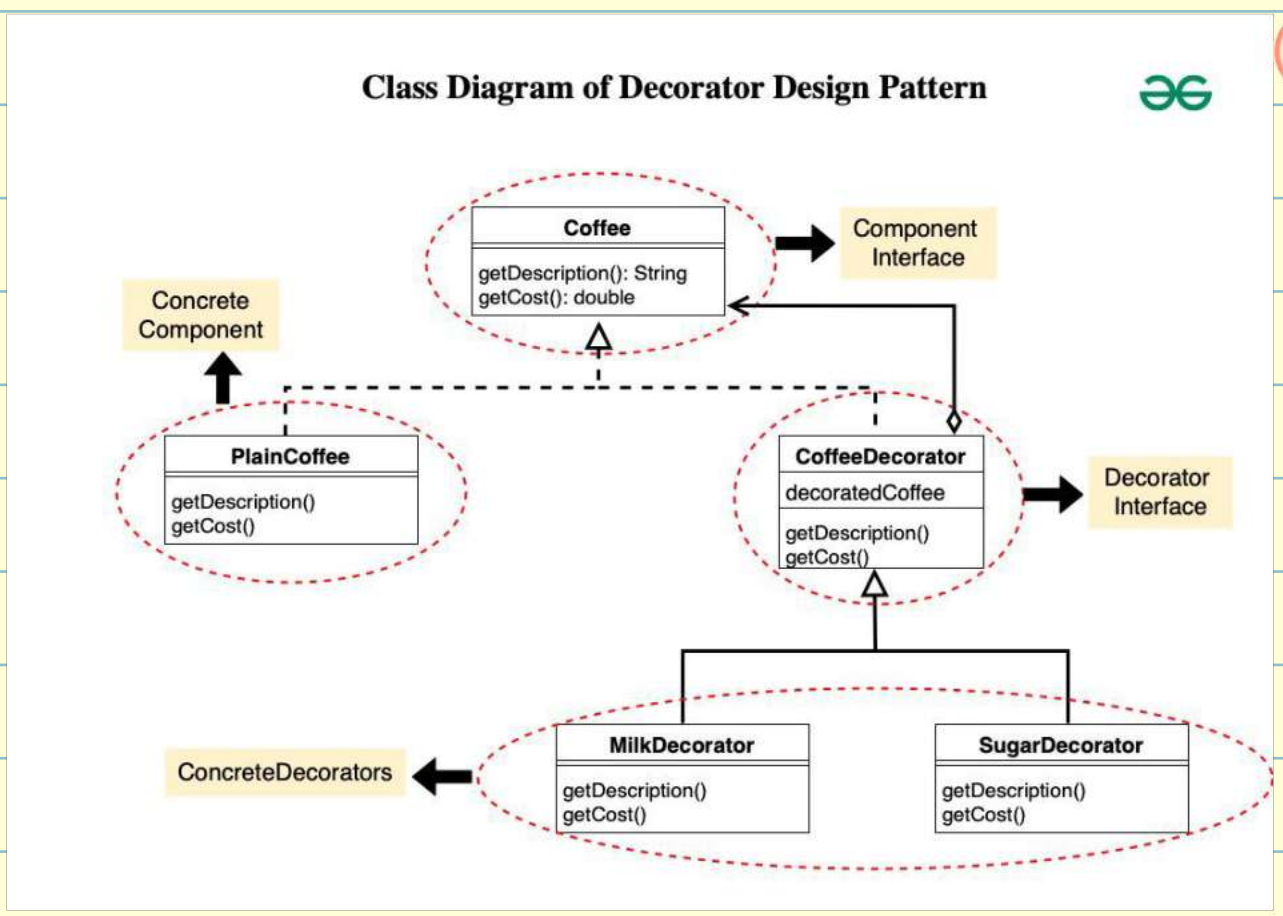
- commonly using in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner.

such as text formatting, graphical user interfaces, or customization of products like coffee or ice-cream.

* Decorators are commonly used in IO streams java.

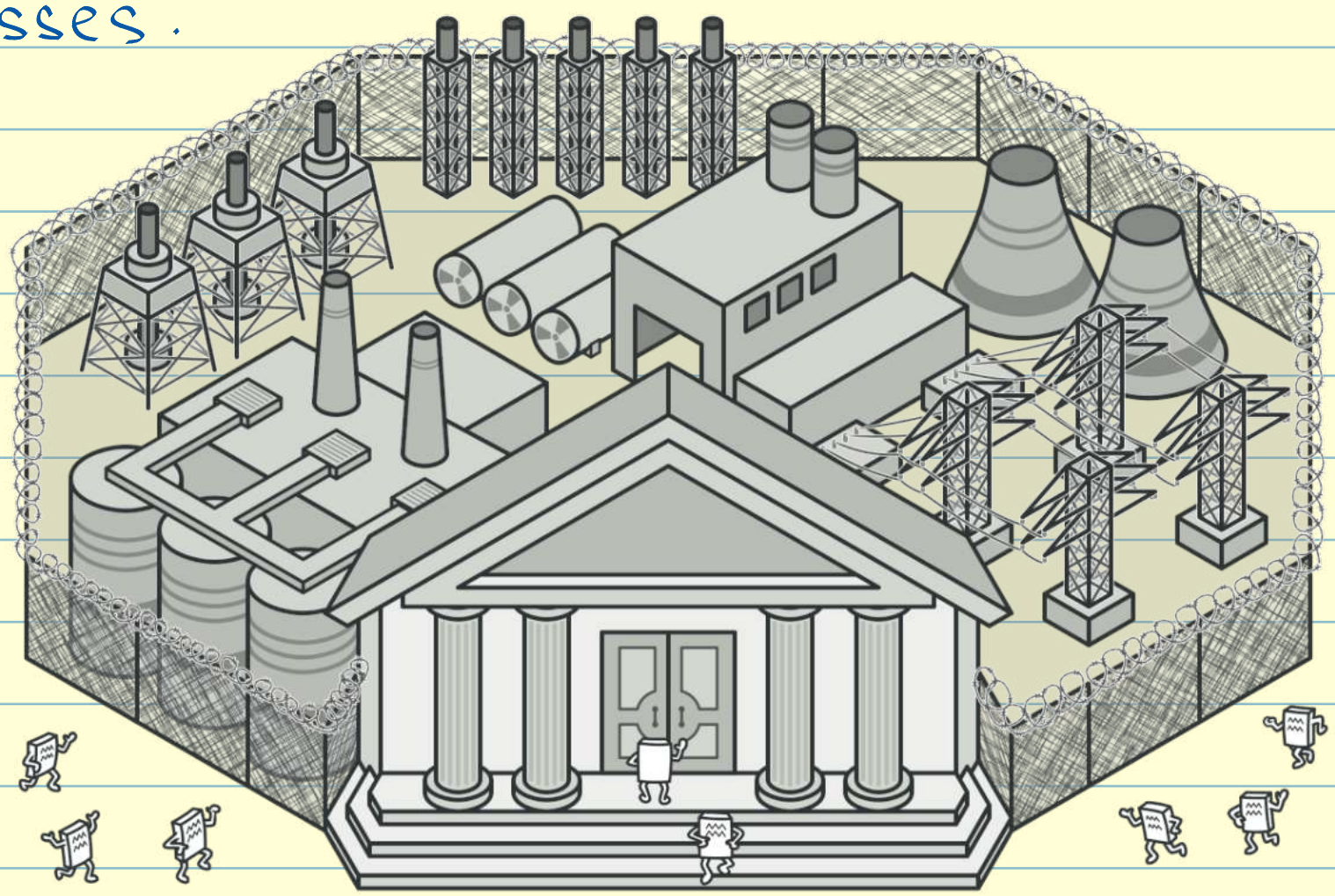
Structure





Facade

- structural design pattern that provides a simplified interface to a library, a framework or any other set of classes.



- in the other words, Facade describes a higher level interface that makes sub system easier to use.

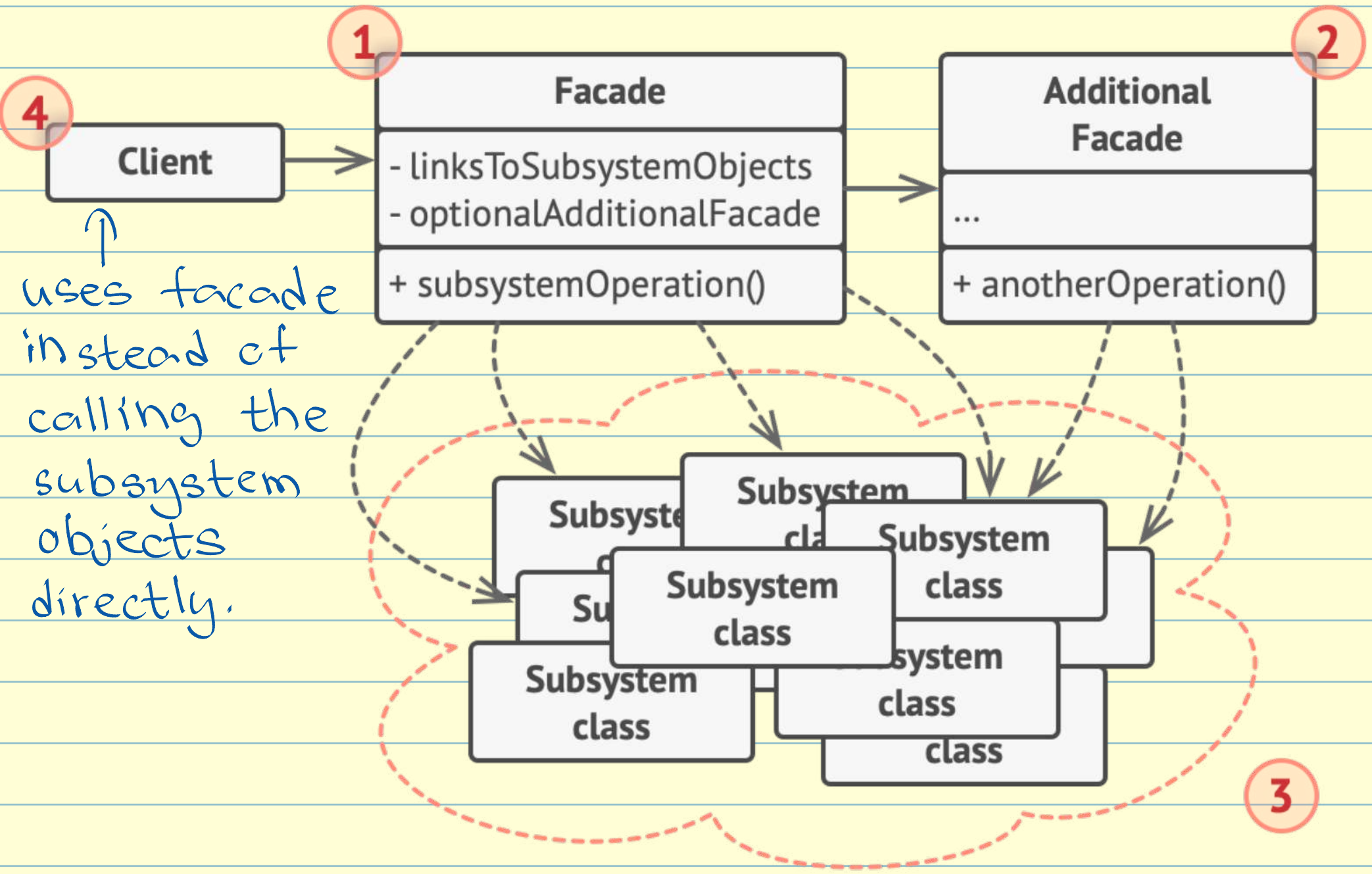
(every abstract factory is a type of facade)

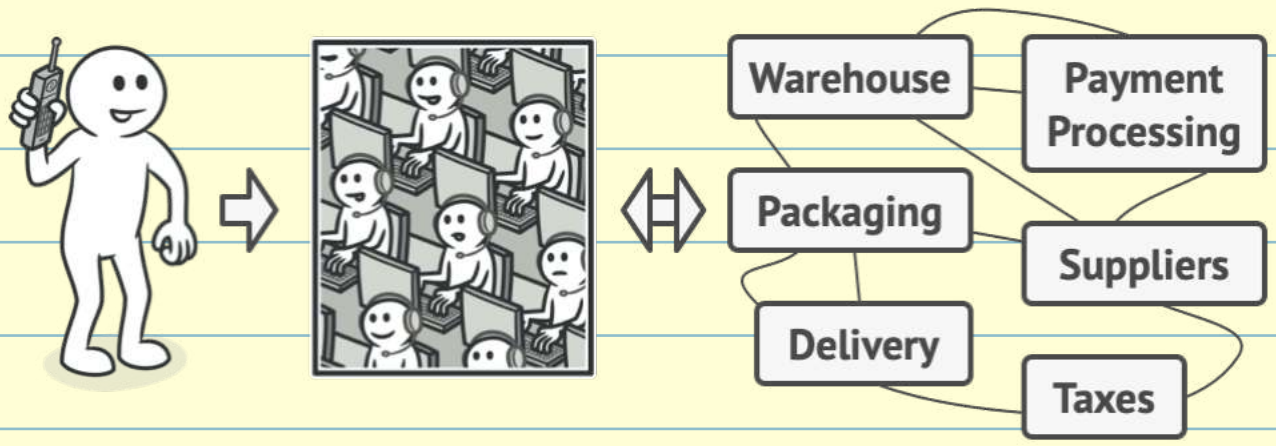
Advantag:-

- promote loose coupling betwee subsystem and its clients.
- sheild the clients from the

complexities of the subsystem components.

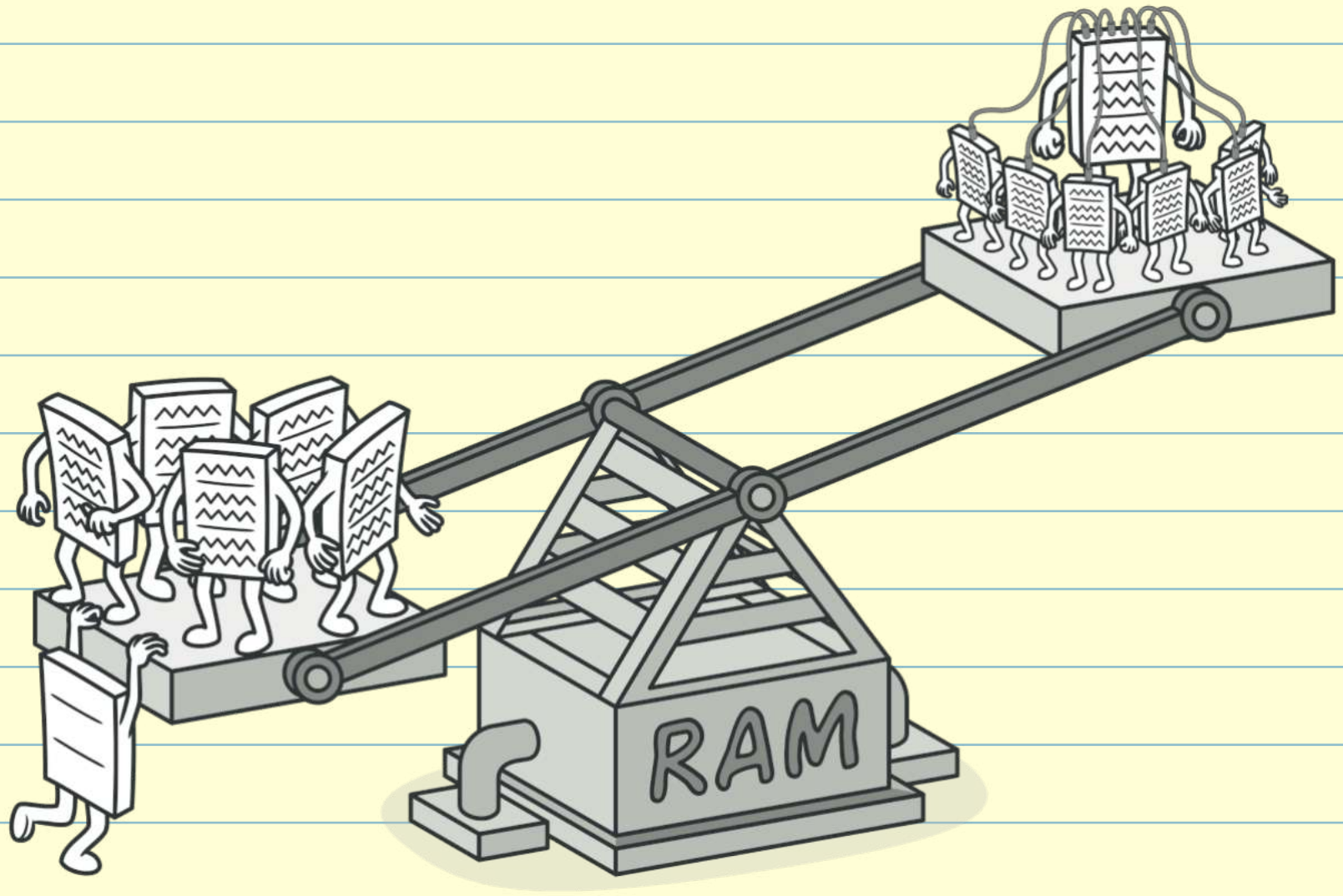
Structure





Flyweight

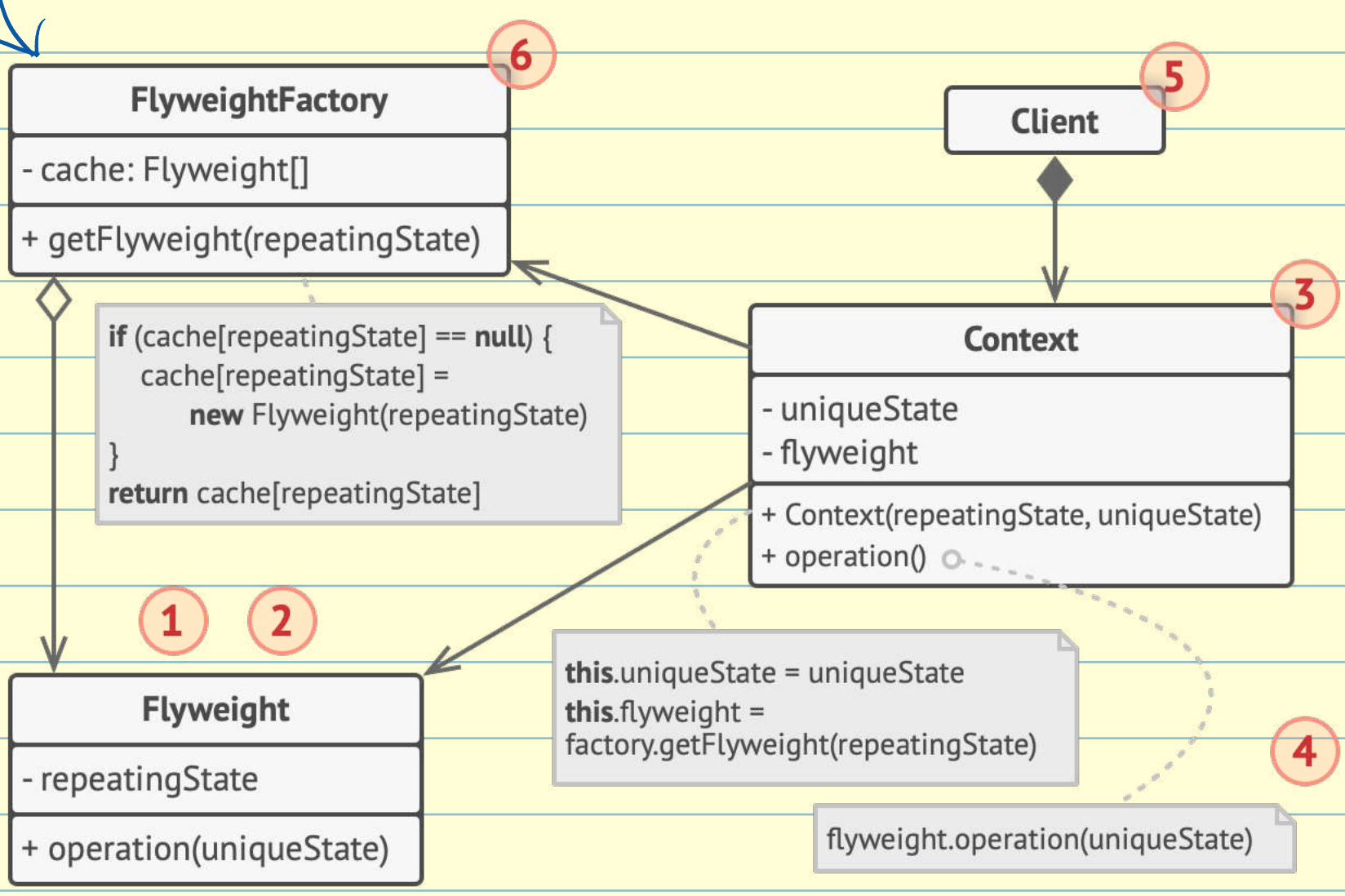
- structural design pattern
- lets you fit more objects into the availabe amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

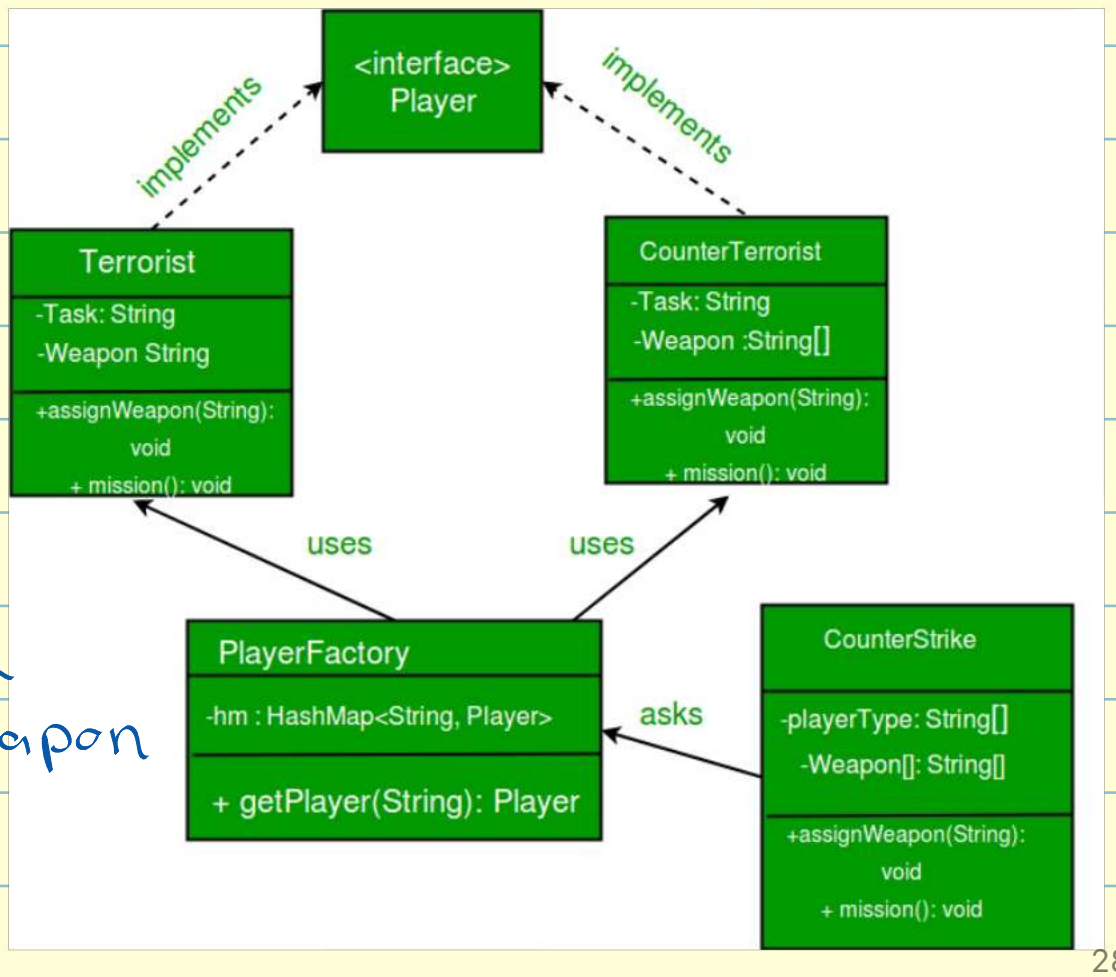


Advantage :-

- reduces the number of objects.
- reduces the amount of memory and storage devices required if the objects are presisted.

manages a pool of existing flyweight.



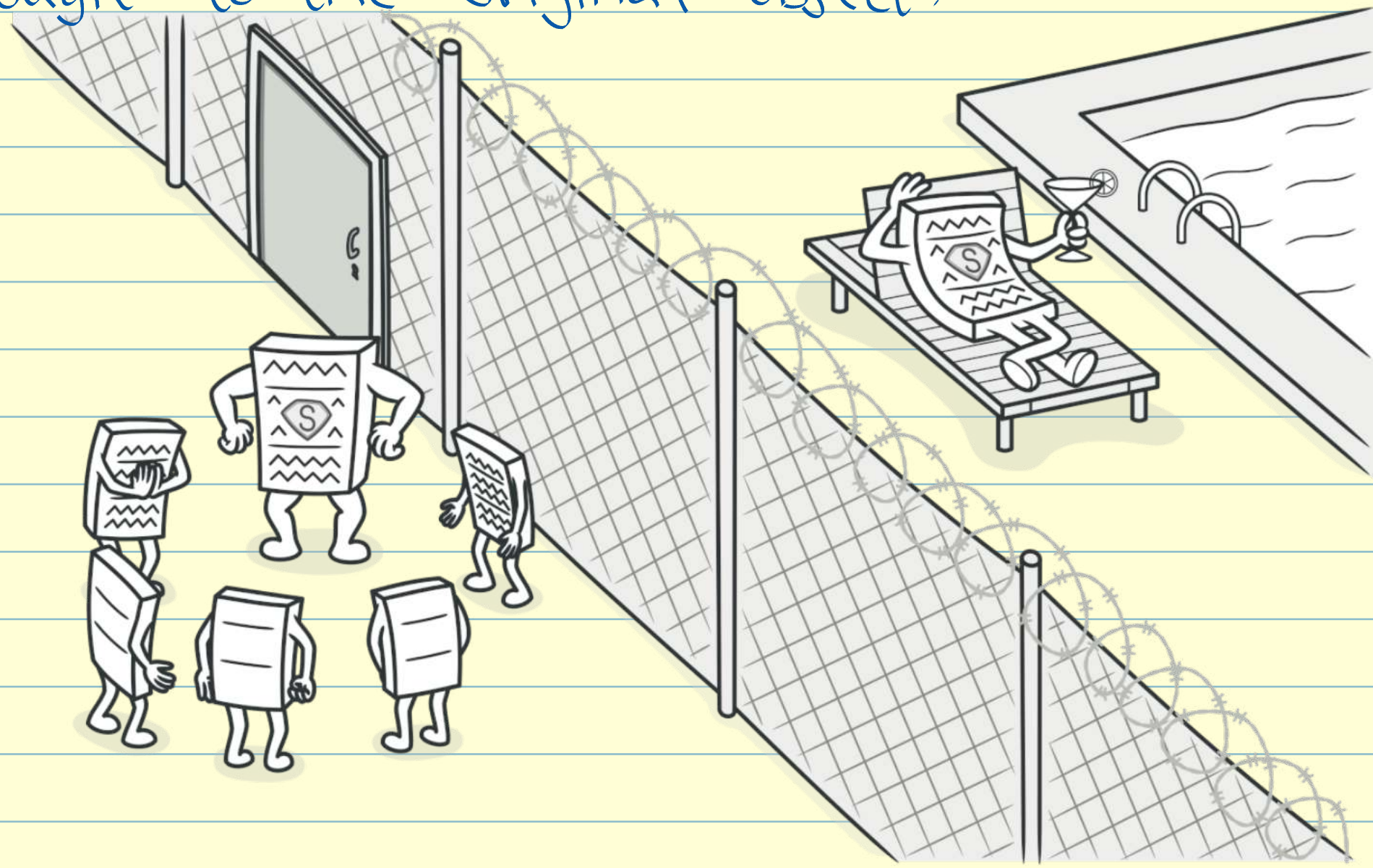


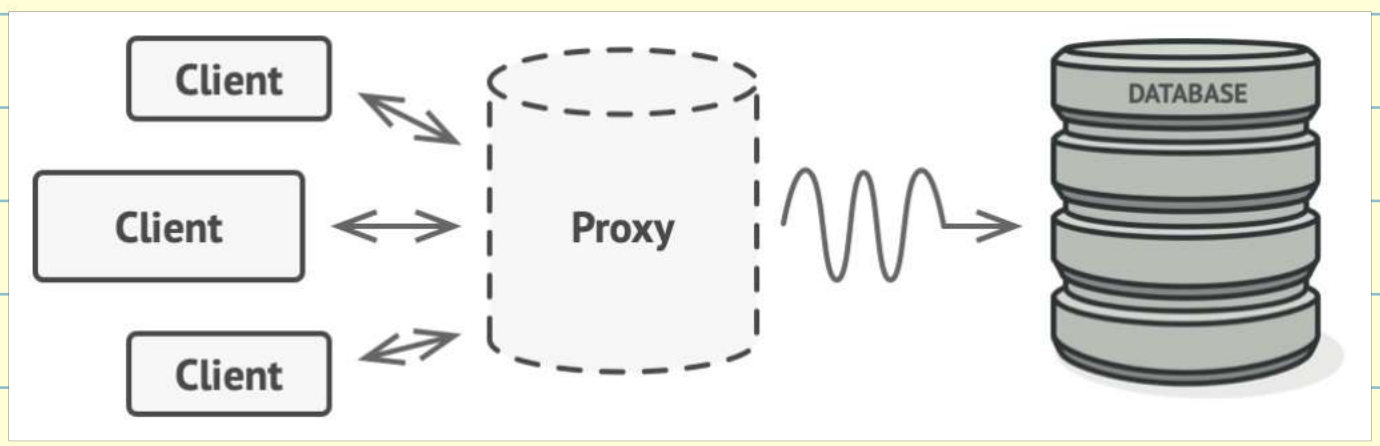
intrinsic state → Task

extrinsic state → weapon

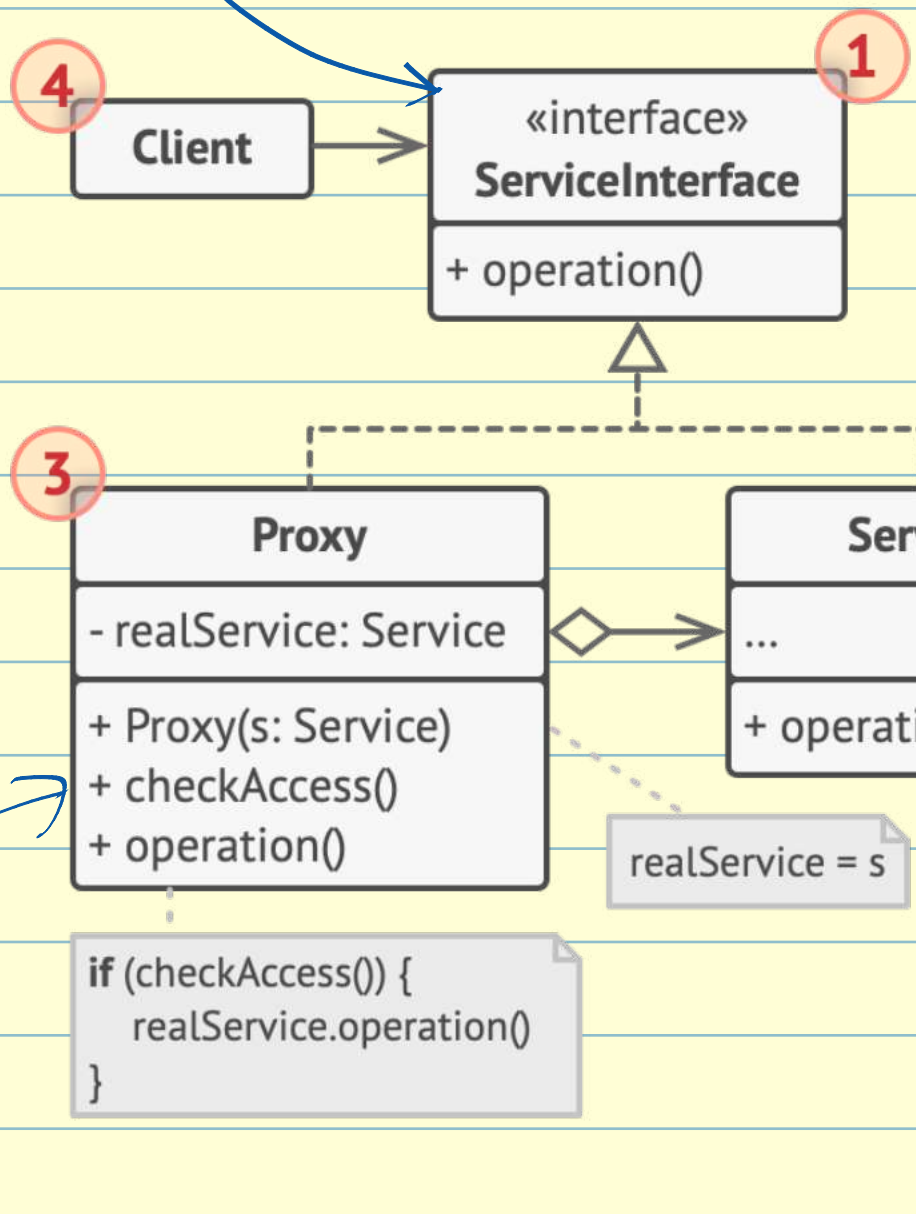
Proxy

- lets you provide a substitute or placeholder for another object. A proxy control the access to the original object, allowing you to perform something either before or after the request gets through to the original object.





common interface for both real object and proxy.



(subject)

(Real subject)

maintains a refference to the real sybject. control access to the real subject, adding additional logic if necessary.

represents the real resource or object that the proxy controls access to.