Design Pattern

Design patterns are common architectural approaches.
Base- From a Book named Gang of four.

SOLID design principles:
Single Responsibility Principle(SRP):
A class should only have a single responsibility.

Open-Closed Principle(LSP):
Entities should be open for Extension but closed for modification.

Liskov Substitution Principle(LSP):
Objects should Be replaceable with instances of their subtypes without altering program correctness.

Interface Segregation Principle(ISP):
Many client specific interfaces are better than one general-purpose interface.

Dependency Inversion Principle(DIP):
Dependencies Should be Abstract rather than Concrete.

Creational :

1.Builder
2.Singleton
3.Factory
4.Prototype
5.Abstract

Structural :

1.Facade
2.Proxy
3.Decorator

Behavioural :

1.Iterator
2.visitor
3.Observer

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

1.Builder:

Builder is a creational design pattern, which allows constructing complex objects step by step.

Unlike other creational patterns, Builder doesn't require products to have a common interface. That makes it possible to produce different products using the same construction process.
Ex: API
Usage examples: The Builder pattern is a well-known pattern in C++ world. It's especially useful when you need to create an object with lots of possible configuration options.

Identification: The Builder pattern can be recognized in class, which has a single creation method and several methods to configure the resulting object. Builder methods often support chaining (for example, `someBuilder->setValueA(1)->setValueB(2)->create()`).

2.Singleton:

Definition- A component which is instantiated only once.

Note: According to the community it is not a good method.

Some of the Approaches:

- Global Variable
- Global Static variable.
- Completely static class.
- Non static class with some static members
- More Sophisticated methods

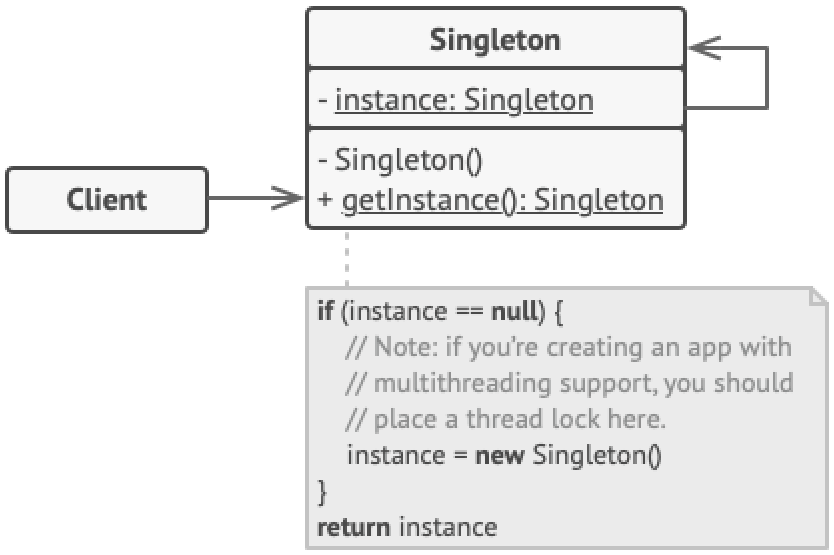
Major Con : Difficult for unit testing.

Implementation:

Naïve Singleton

It’s pretty easy to implement a sloppy Singleton. You just need to hide the constructor and implement a static creation method.

The same class behaves incorrectly in a multithreaded environment. Multiple threads can call the creation method simultaneously and get several instances of Singleton class.



```
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
```

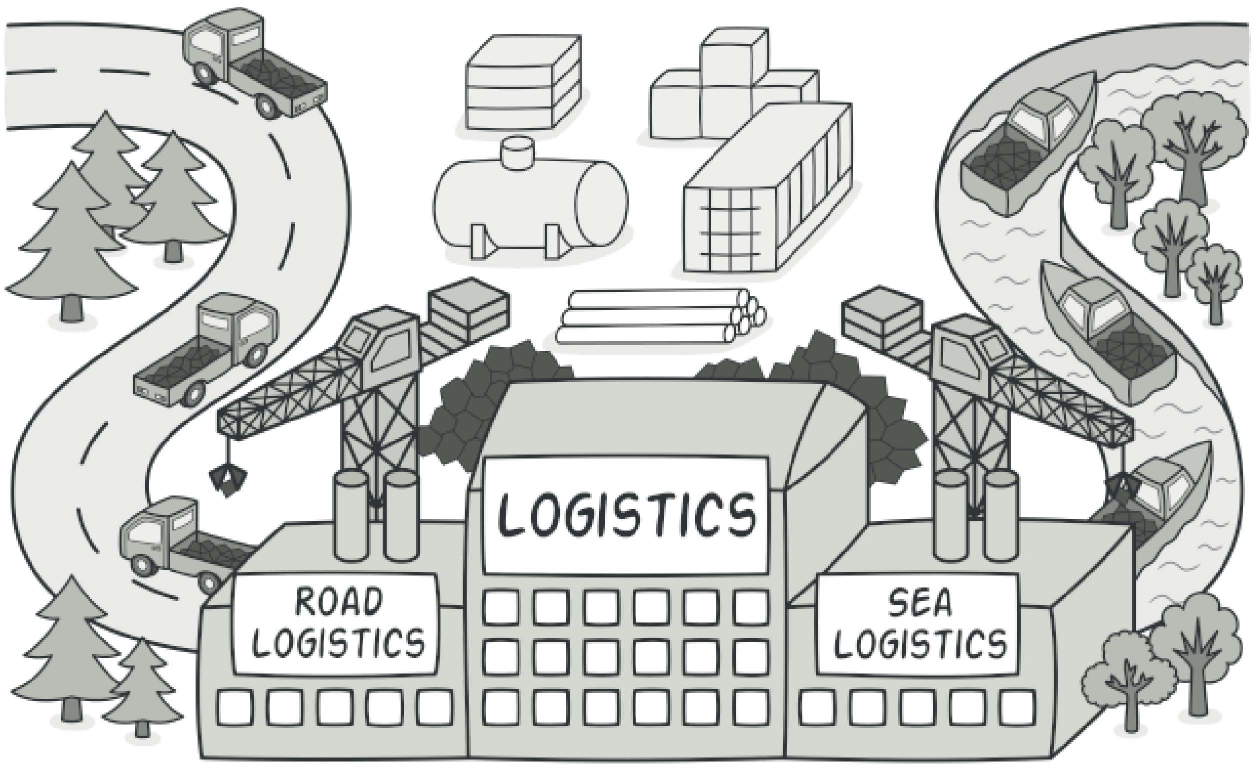
Thread-safe Singleton: To fix the problem, you have to synchronize threads during the first creation of the Singleton object.

3. Factory Method

Also Known as Virtual Constructor.

Interview Question on OOPs

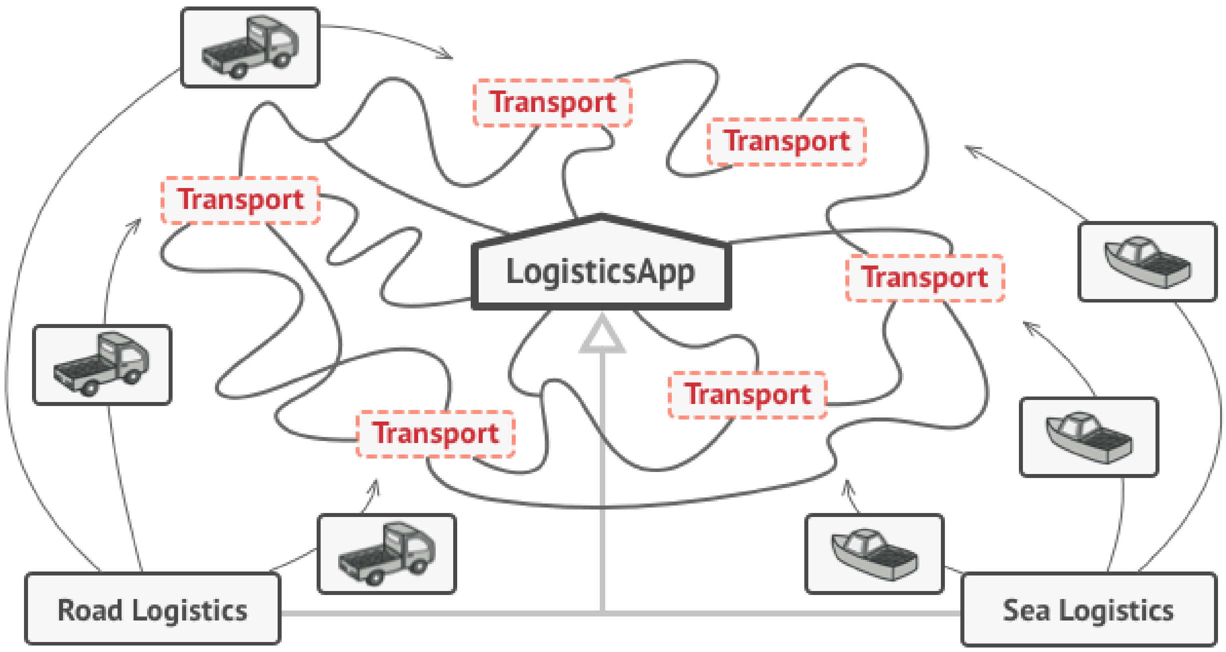
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Problem:

LogisticsApp

Solution:

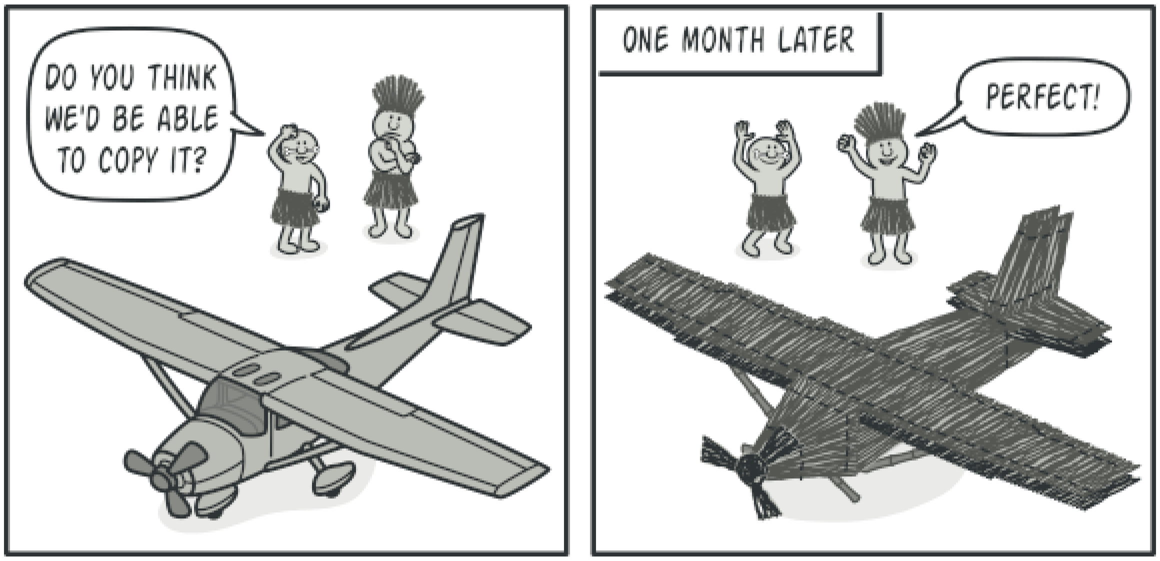


Nearest example to us Windows and Web Button

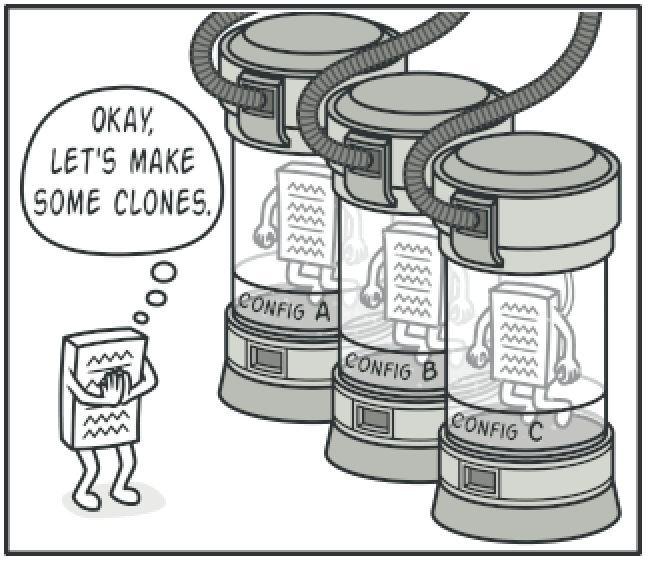
4. Prototype:

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

Problem:

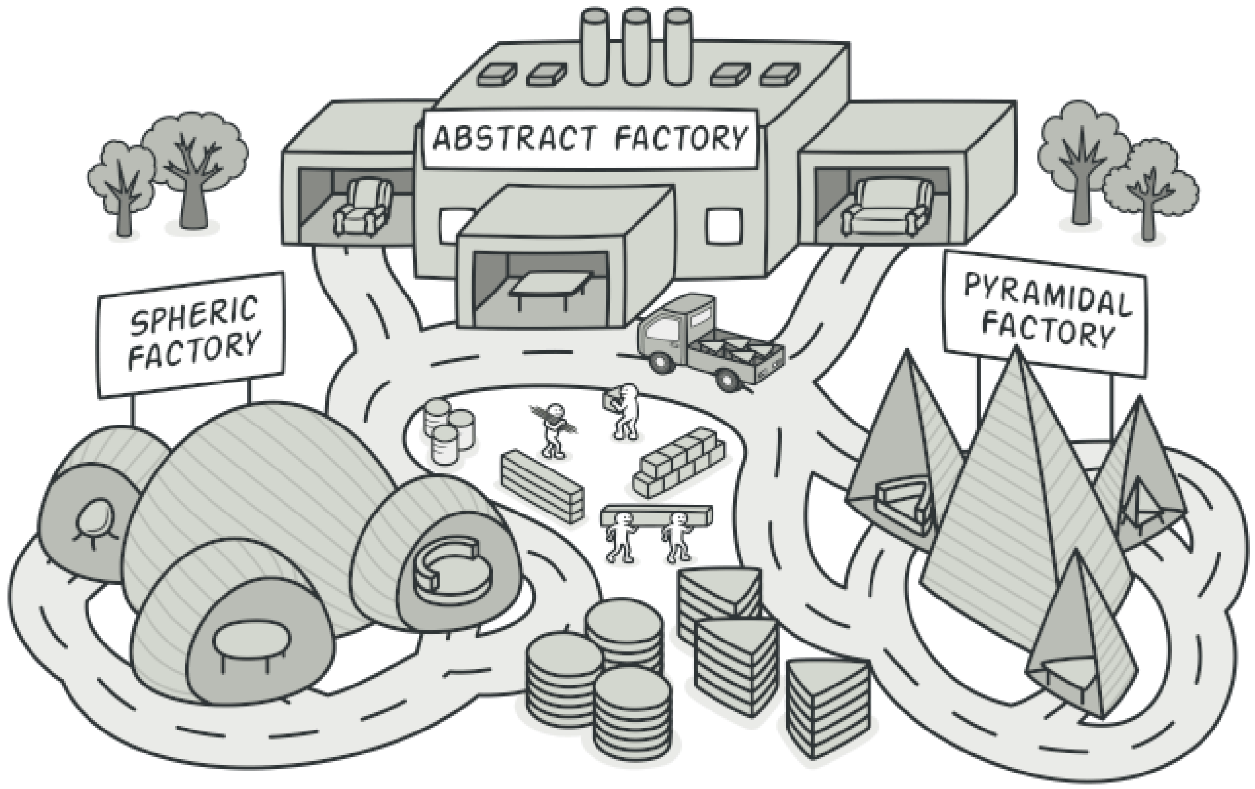


Solution:

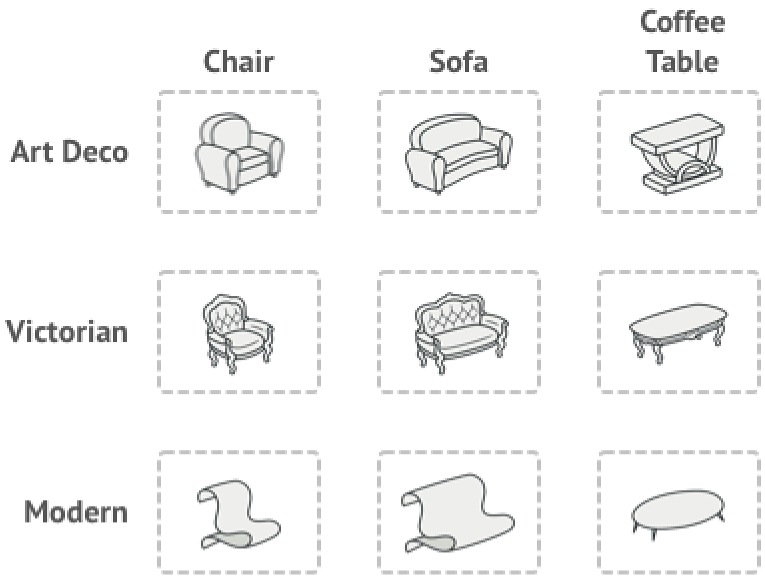


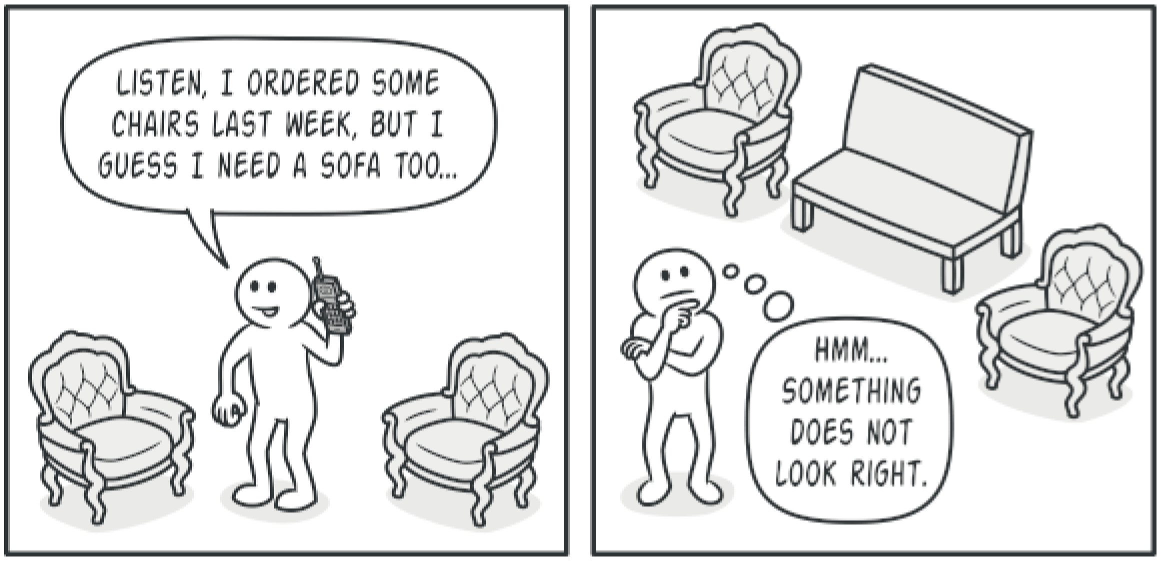
5. Abstract Factory Method:

Abstract Factory is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.

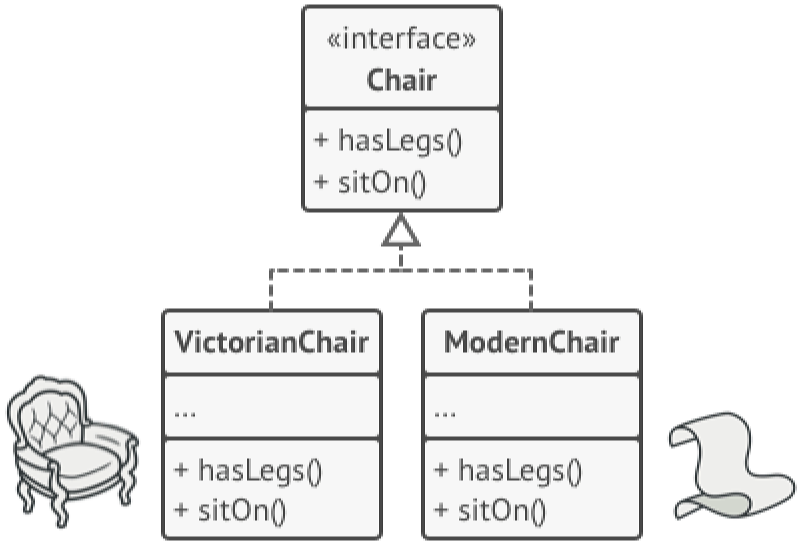

ABSTRACT FACTORY
SPHERIC FACTORY
PYRAMIDAL FACTORY

Problem:


Chair
Sofa
Coffee Table
Art Deco
Victorian
Modern


LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

Solution:

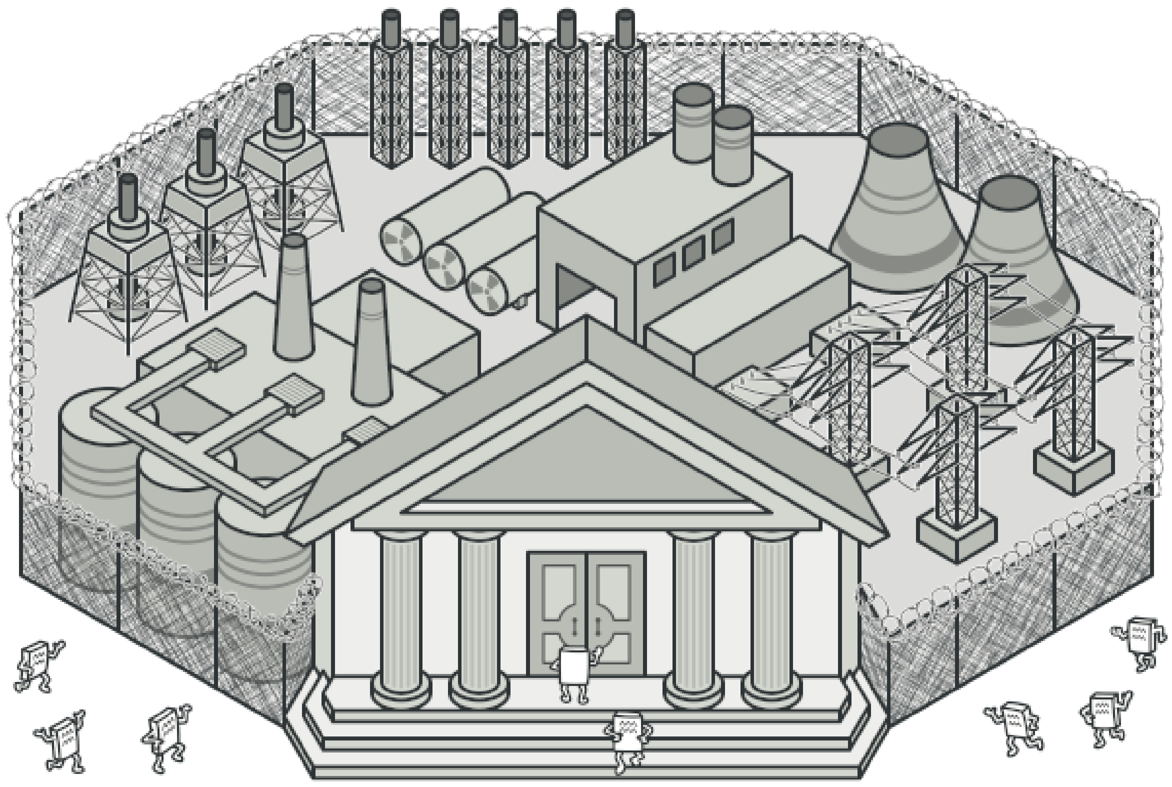

«interface»
Chair
+ hasLegs()
+ sitOn()
VictorianChair
...
+ hasLegs()
+ sitOn()
ModernChair
...
+ hasLegs()
+ sitOn()

Structural

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

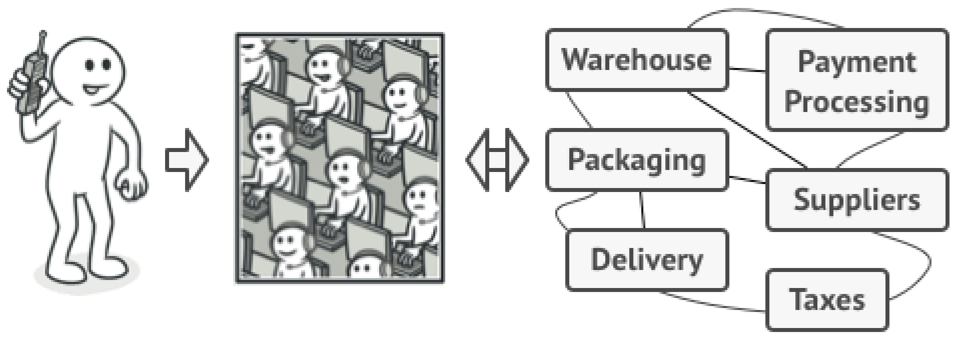
1.Facade
2.Decorator
3.Proxy

1.Facade: French word “Face”.



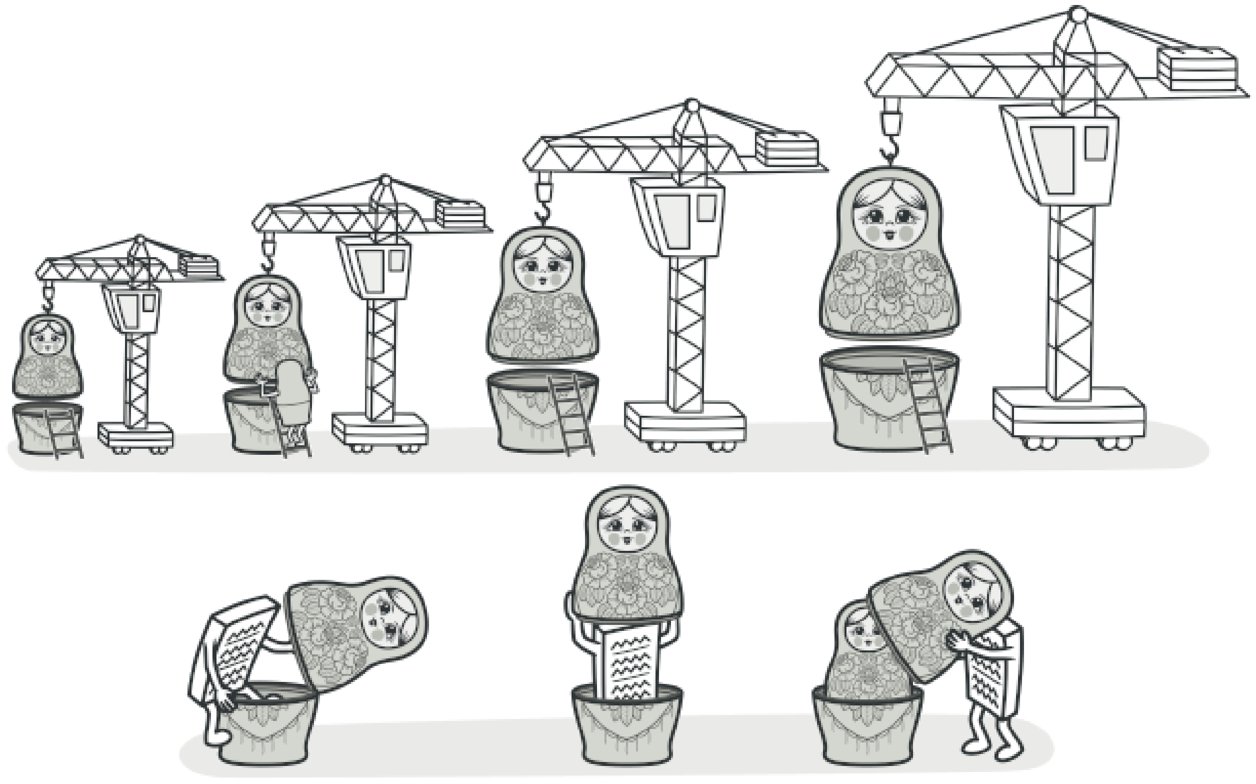
Problem:
Too many functions and dependencies.Which increase the complexity to clients.

Solution:
Face



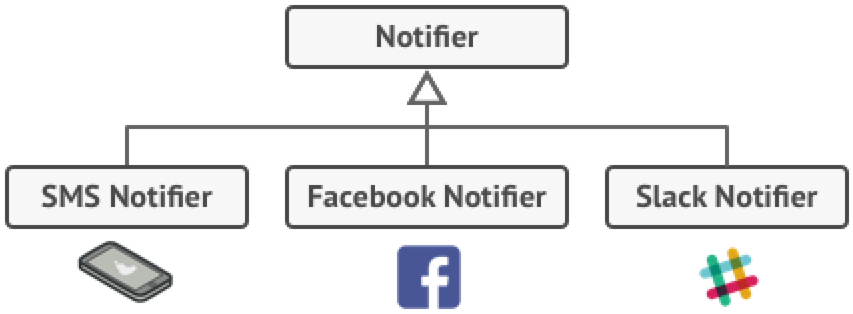
Decorator

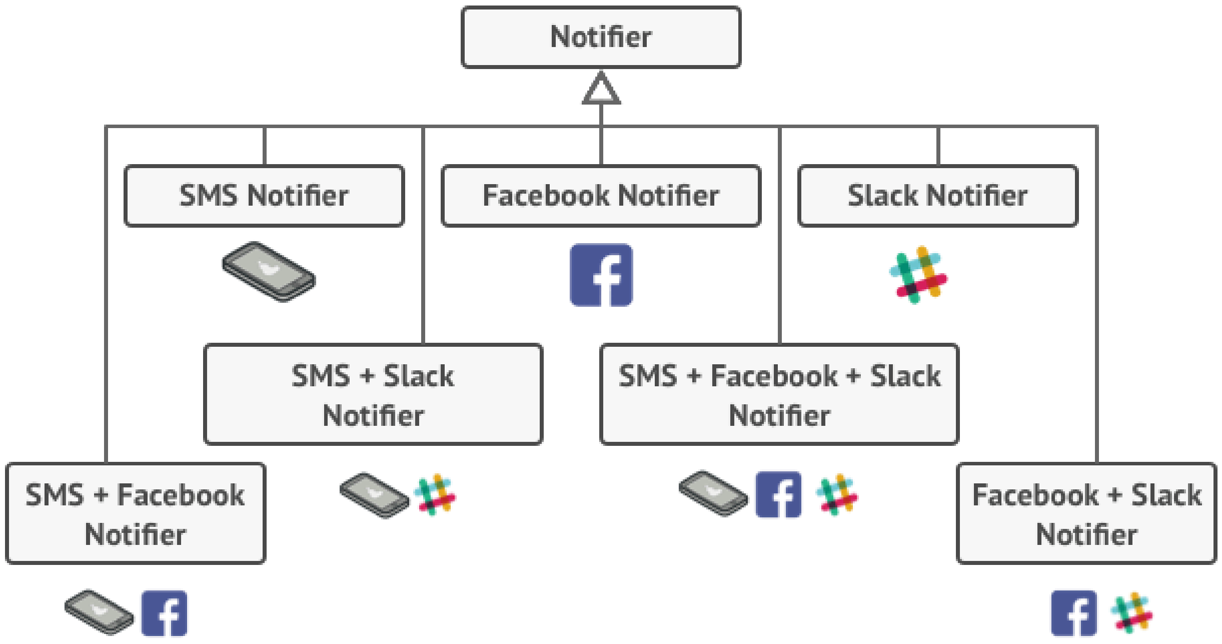
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



Problem:

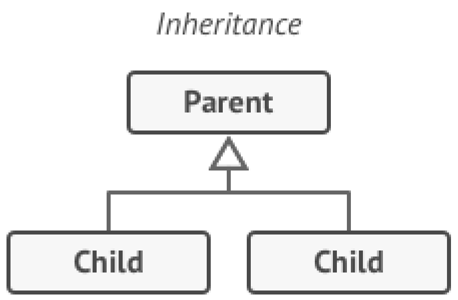
Example Notifier:

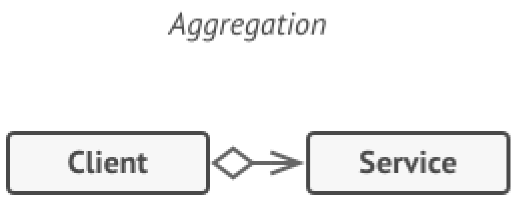




Solution:

One of the ways to overcome these caveats is by using *Aggregation* or *Composition* instead of *Inheritance*. Both of the alternatives work almost the same way: one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.

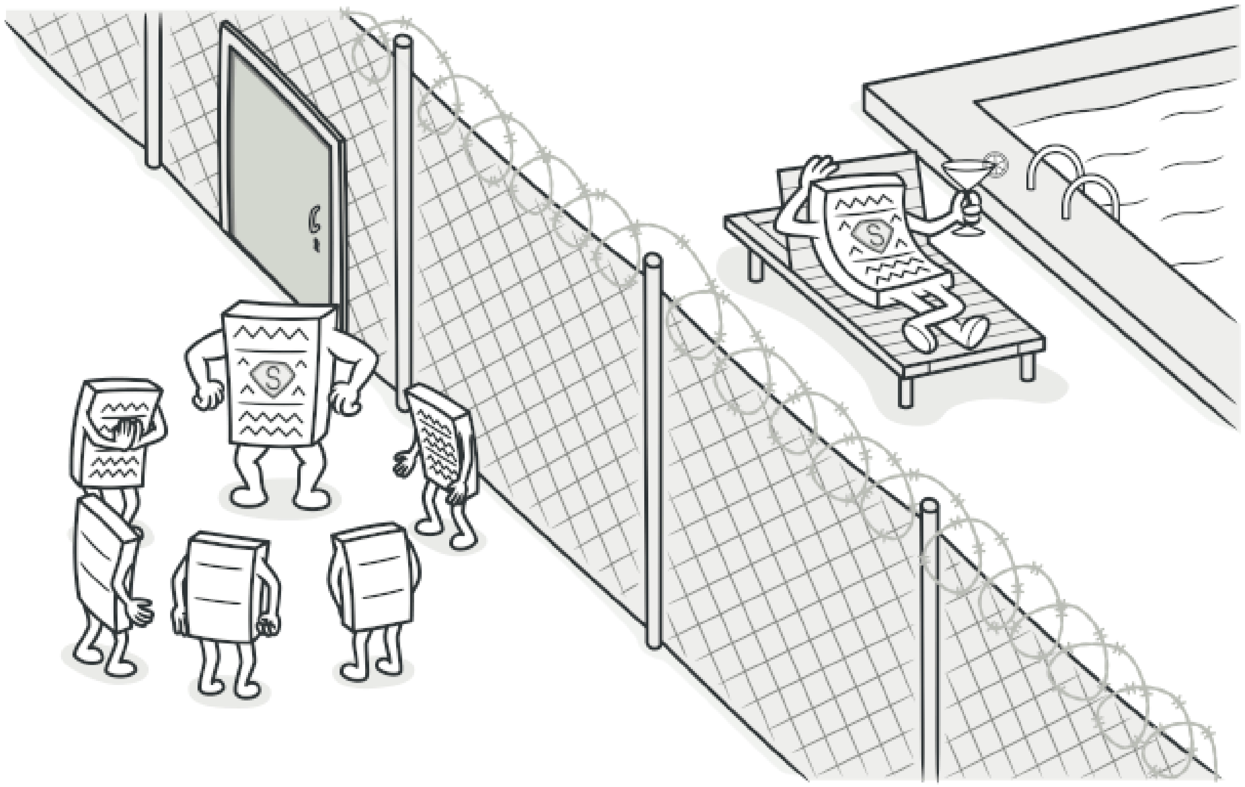




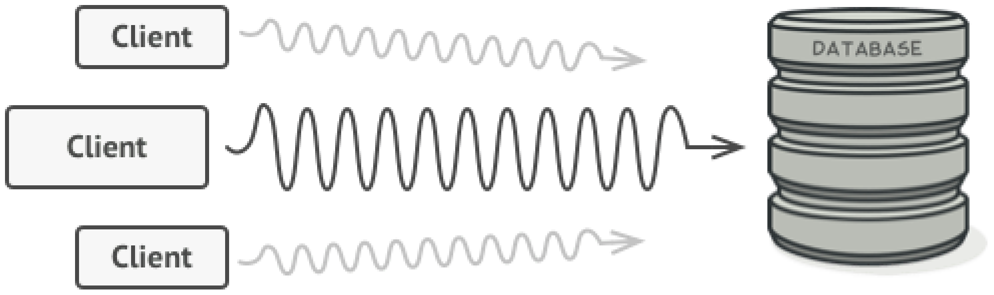
Coding Section

Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

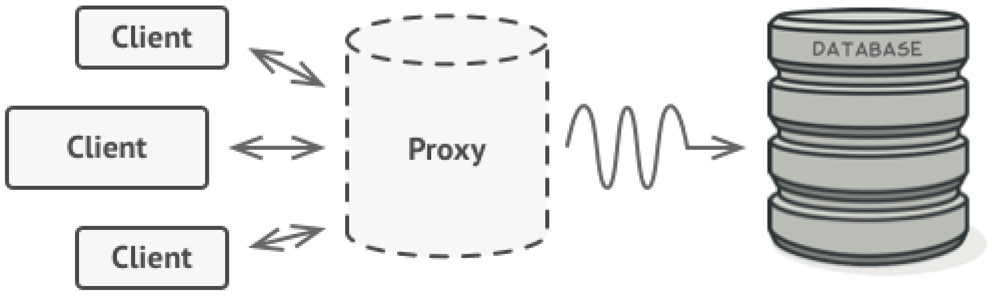


Problem:



You could implement lazy initialization: create this object only when it's actually needed. All of the objects clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

Solution:



The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

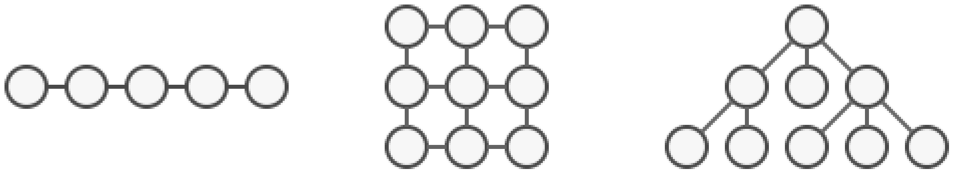
Behavioral Design Patterns

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

Iterator:

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Problem:
Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

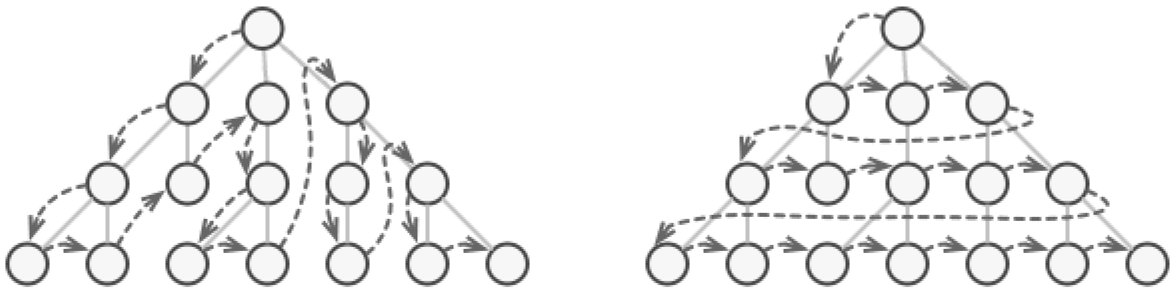


Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.



The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

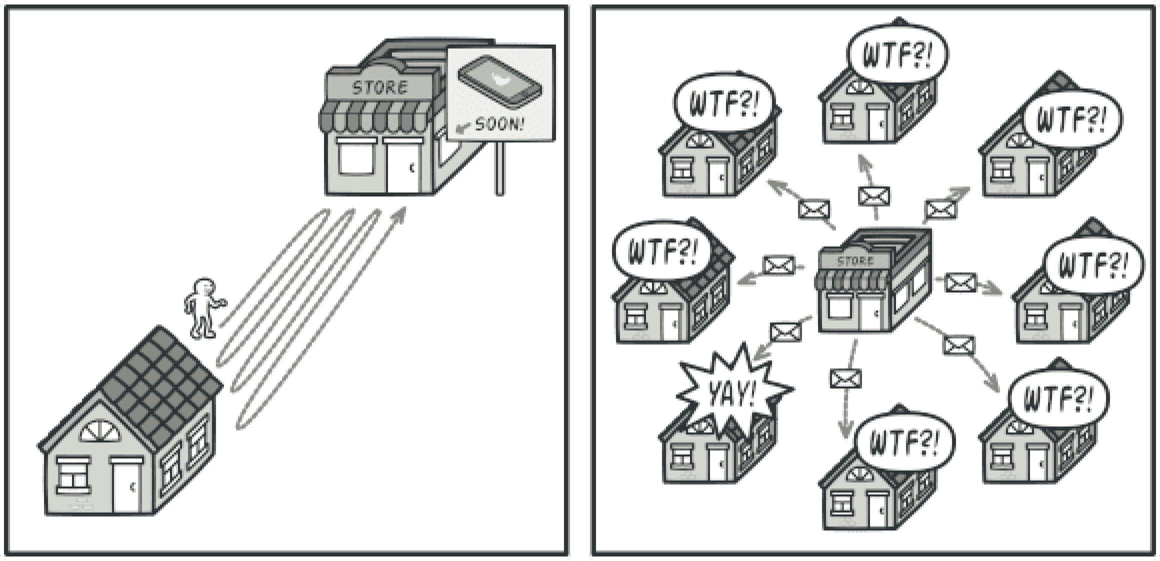
Observer:

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Problem:

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.



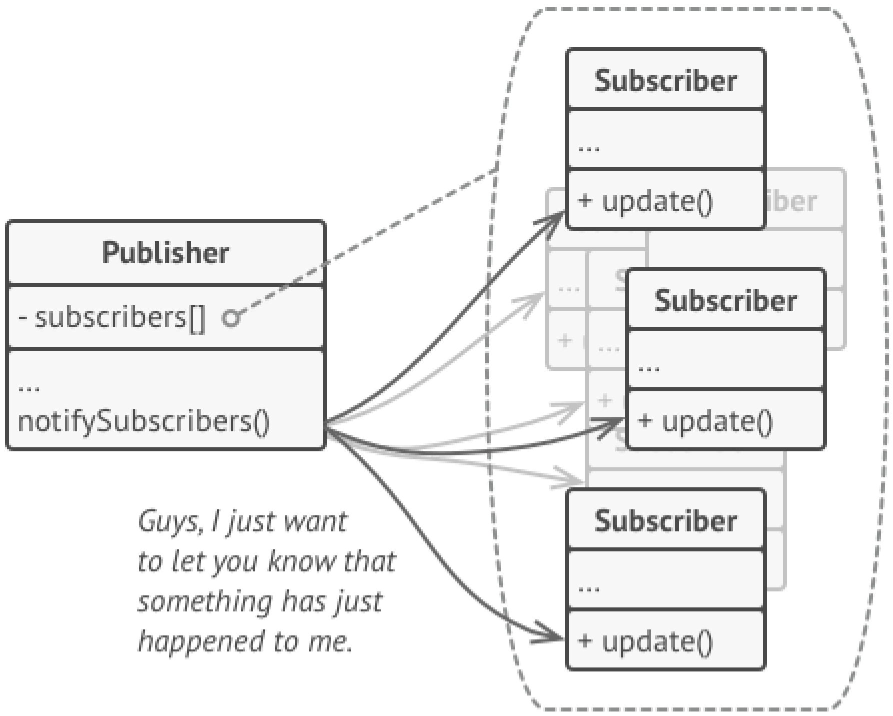
Visiting the store vs. sending spam

On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.

It looks like we've got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

Solution:

Subscription:

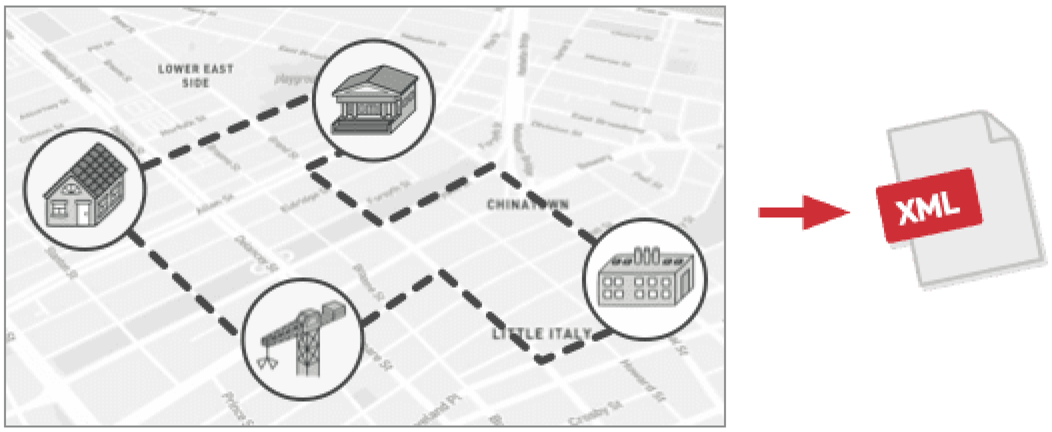


Visitor

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Visitor is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.

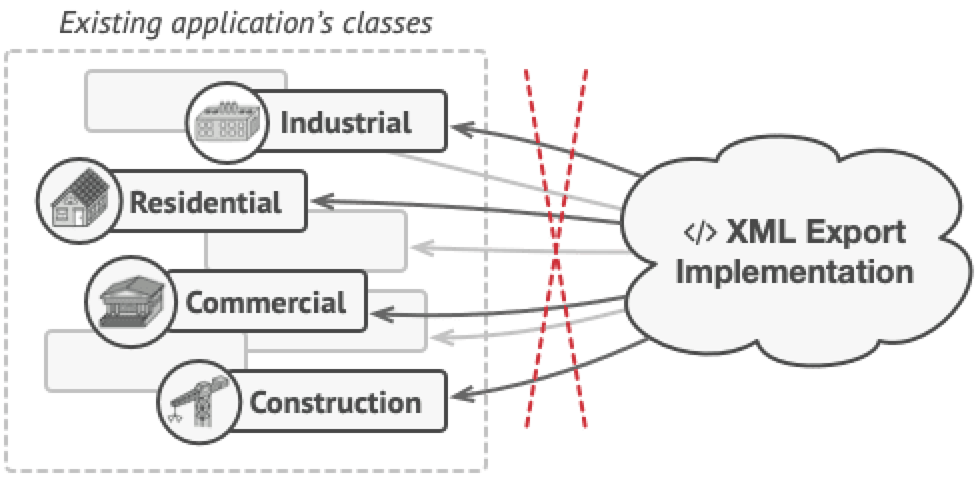
Imagine that your team develops an app which works with geographic information structured as one colossal graph. Each node of the graph may represent a complex entity such as a city, but also more granular things like industries, sightseeing areas, etc. The nodes are connected with others if there’s a road between the real objects that they represent. Under the hood, each node type is represented by its own class, while each specific node is an object.



Exporting the graph into XML.

At some point, you got a task to implement exporting the graph into XML format. At first, the job seemed pretty straightforward. You planned to add an export method to each node class and then leverage recursion to go over each node of the graph, executing the export method. The solution was simple and elegant: thanks to polymorphism, you weren't coupling the code which called the export method to concrete classes of nodes.

Unfortunately, the system architect refused to allow you to alter existing node classes. He said that the code was already in production and he didn't want to risk breaking it because of a potential bug in your changes.



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

Besides, he questioned whether it makes sense to have the XML export code within the node classes. The primary job of these classes was to work with geodata. The XML export behavior would look alien there.

There was another reason for the refusal. It was highly likely that after this feature was implemented, someone from the marketing department would ask you to provide the ability to export into a different format, or request some other weird stuff. This would force you to change those precious and fragile classes again.

Link for python: https://refactoring.guru/design-patterns/python