Module-3

Design Patterns

What is Design Pattern?

- Design patterns are typical solutions to commonly occurring problems in software design.
- They are like, pre-made blueprints that can be customized to solve a recurring design problem in your code.
- The pattern details and implementation of a solution that suits the realities of your own program.
- An analogy to an algorithm is a cooking recipe; both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint.

What does the pattern consist of?

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code** example in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Why Should We Learn Patterns?

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design.
- It is useful because it teaches how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a common language that can be used to communicate more efficiently within common team members.

Creational patterns

- Creational Patterns provides various object creation mechanism, which increase flexibility and reuse of existing code.
  - Factory Method
  - Abstract Factory Method
  - Builder
  - Lazy Initialization
  - Object Pool
  - Singleton

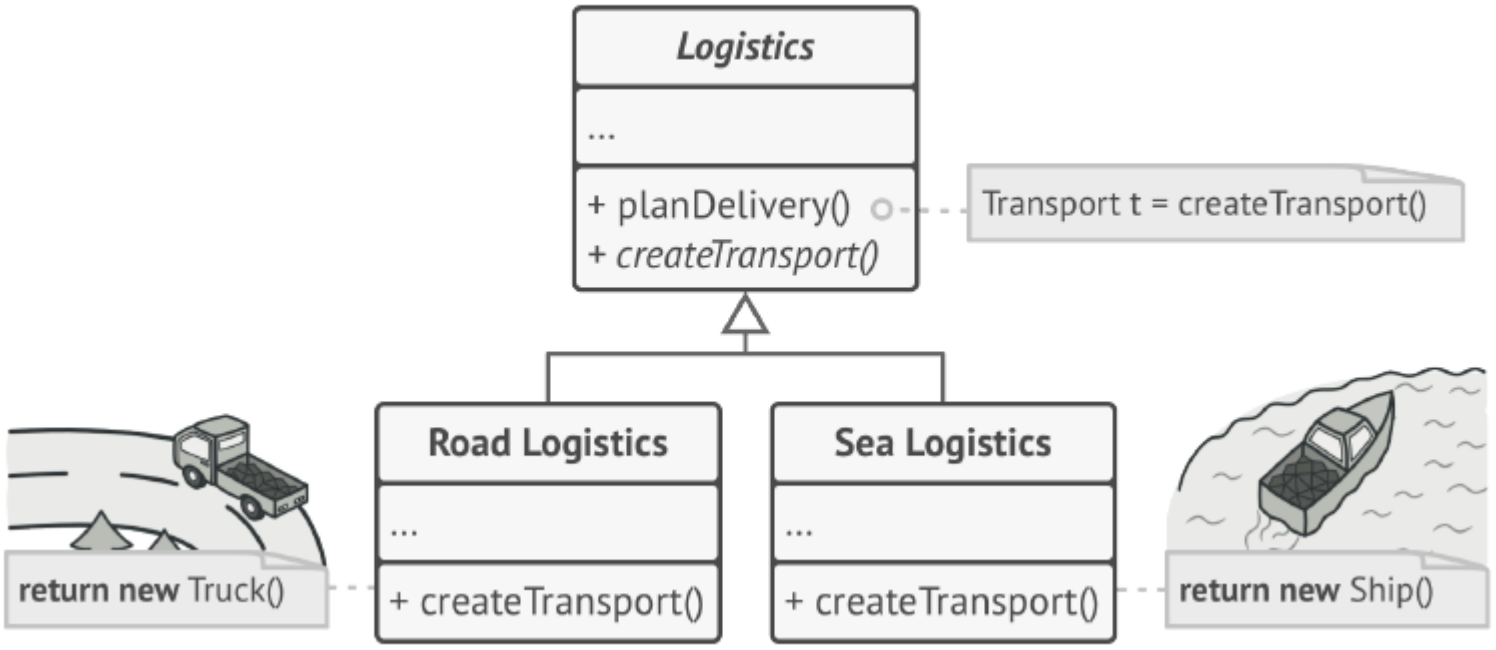
Factory Method

- **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.
- It is also known as: Virtual Constructor

Problem

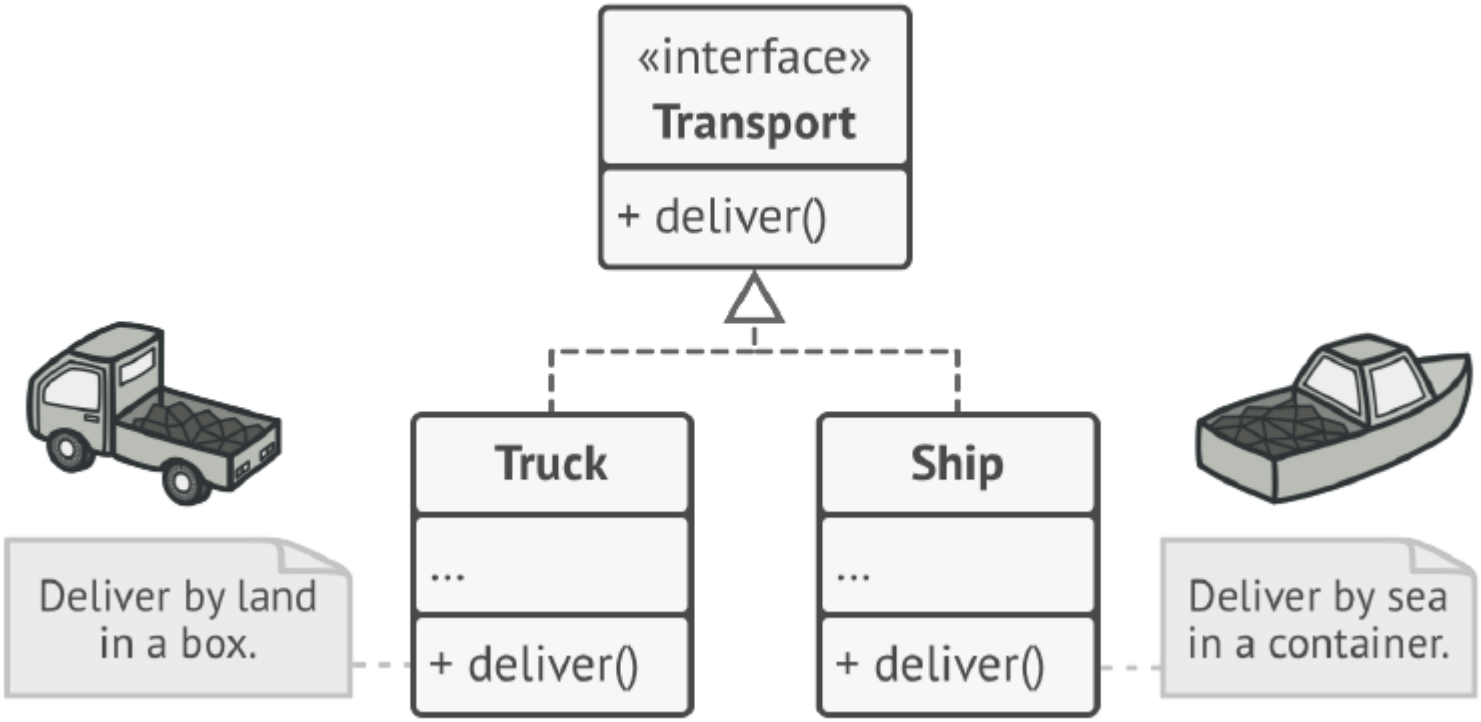
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Solution



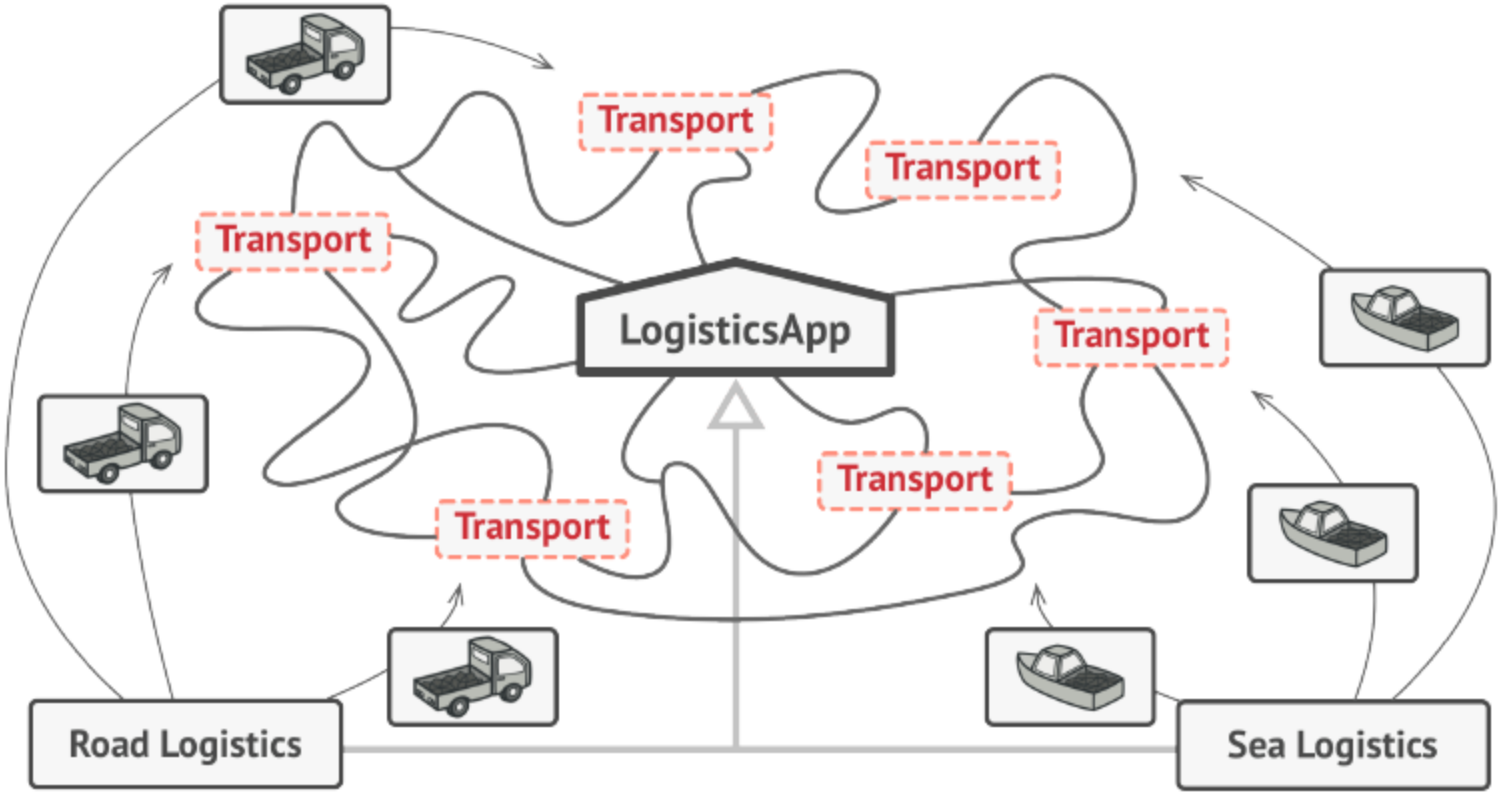
Subclasses can alter the class of objects being returned by the factory method.

Contd.



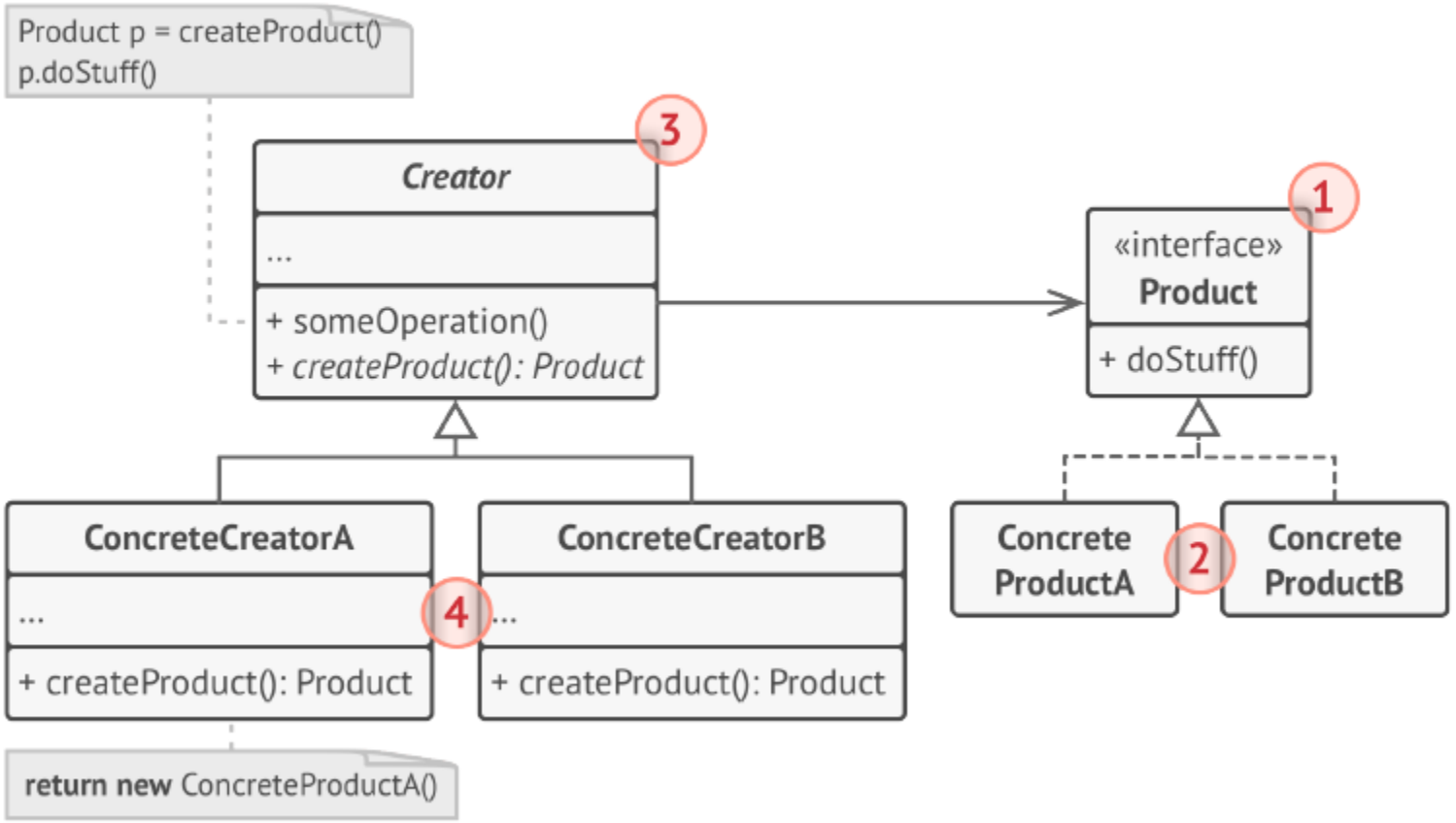
All products must follow the same interface.

Contd.

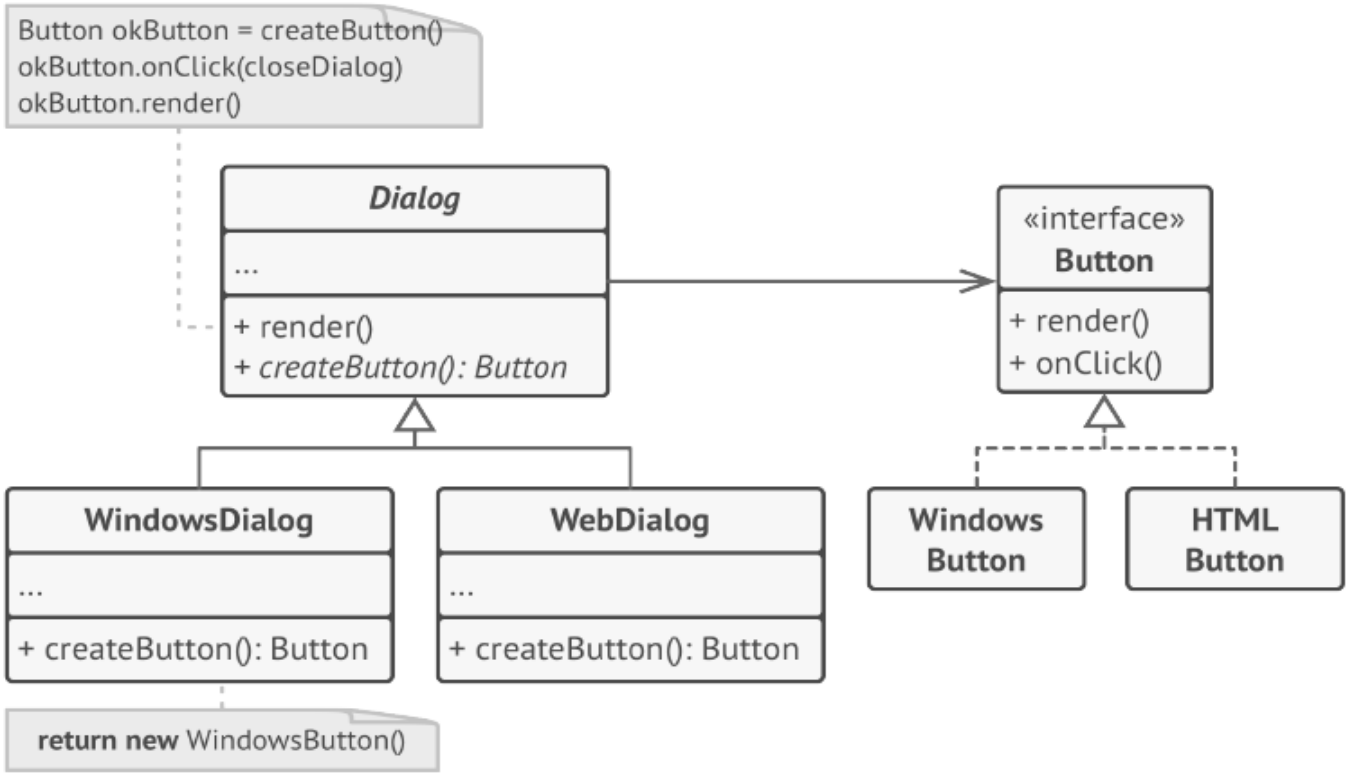


As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

Structure


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

Implementation



This example illustrates how the Factory Method can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes.

Applicability

- Use the Factory Method when we don’t know beforehand the exact types and dependencies of the objects our code should work with.
  - The Factory Method separates product construction code from the code that actually uses the product. Therefore it’s easier to extend the product construction code independently from the rest of the code.
  - For example, to add a new product type to the app, we’ll only need to create a new creator subclass and override the factory method in it.

Contd.

- Use the Factory Method when we want to provide users of our library or framework with a way to extend its internal components.
  - Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that the subclass should be used instead of a standard component?
  - The solution is to reduce the code that constructs components across the framework into a single factory method and let anyone override this method in addition to extending the component itself.

Pros and Cons

- Avoid tight coupling between the creator and the concrete products.
- *Single Responsibility Principle*. One can move the product creation code into one place in the program, making the code easier to support.
- *Open/Closed Principle*. One can introduce new types of products into the program without breaking existing client code.
- The code may become more complicated since one need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when one is introducing the pattern into an existing hierarchy of creator classes.