Factory Method

Phan Pham Thanh Tuyen

Primary Secondary

Intent

- Is a creational design pattern
- Provide an interface for creating objects in a superclass
- But allows subclasses to alter the type of objects that will be created

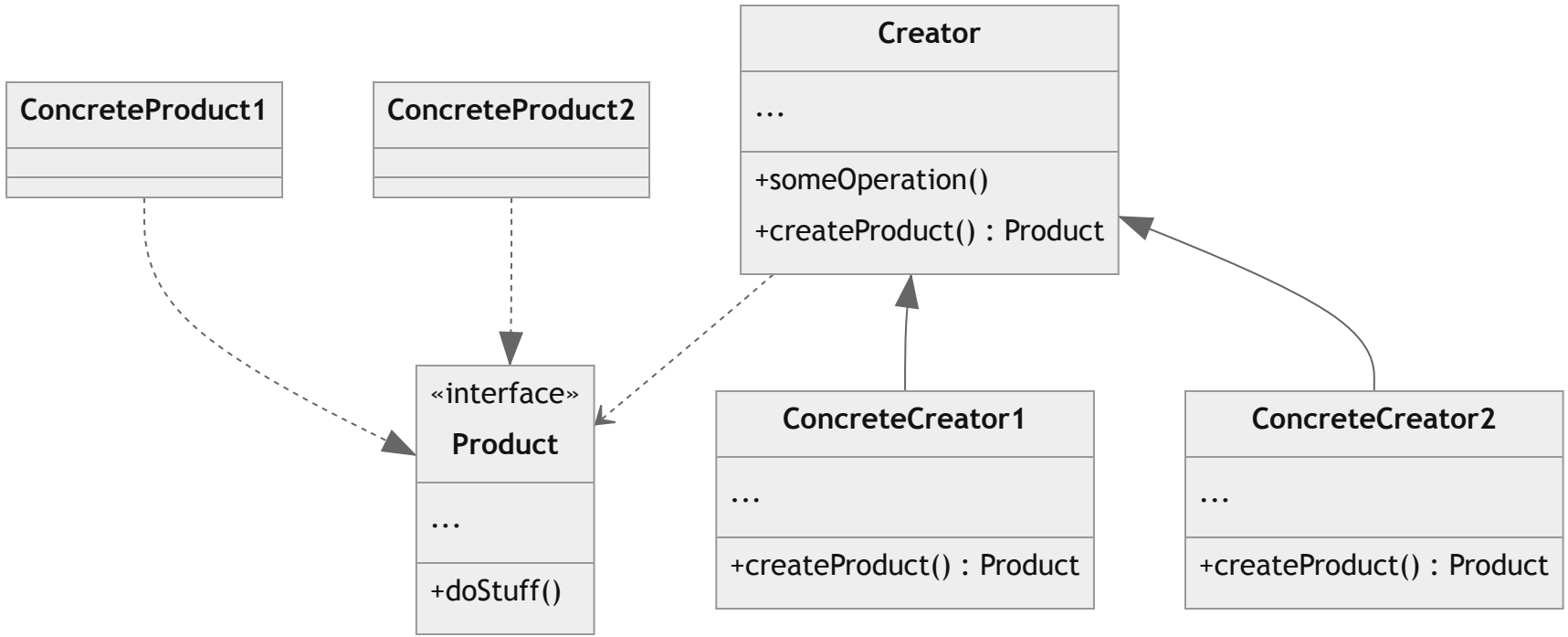
Problem

- 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.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.
- Great news, right? But how about the code? 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. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.
- As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

Solution

- Replace direct object construction calls with calls to a special factory method
- We can override the factory method in a subclass and change the class of products being created by the method
- Objects returned by a factory method are often referred to as products
- Subclasses may return different types of products only if these products have a common base class or interface
- Also, the factory method in the base class should have its return type declared as this interface.
- The code that uses the factory method (often called the client code) doesn't see a difference between the actual products returned by various subclasses

Class Diagram



Code Example 

```
abstract class Creator {
  public abstract factoryMethod(): Product;

  public someOperation(): string {
    const product = this.factoryMethod();
    return `Creator: The same creator's code has just worked with ${product.operation()}`;
  }
}

class ConcreteCreator1 extends Creator {
  public factoryMethod(): Product {
    return new ConcreteProduct1();
  }
}

class ConcreteCreator2 extends Creator {
  public factoryMethod(): Product {
    return new ConcreteProduct2();
  }
}
```

Code Example 

```
interface Product {
  operation(): string;
}

class ConcreteProduct1 implements Product {
  public operation(): string {
    return '{Result of the ConcreteProduct1}';
  }
}

class ConcreteProduct2 implements Product {
  public operation(): string {
    return '{Result of the ConcreteProduct2}';
  }
}
```

Code Example 

```
function clientCode(creator: Creator) {
  console.log(
    "Client: I'm not aware of the creator's class, but it still works."
  );
  console.log(creator.someOperation());
}

console.log('App: Launched with the ConcreteCreator1.');
clientCode(new ConcreteCreator1());
console.log('');

console.log('App: Launched with the ConcreteCreator2.');
clientCode(new ConcreteCreator2());

// App: Launched with the ConcreteCreator1.
// Client: I'm not aware of the creator's class, but it still works.
// Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

// App: Launched with the ConcreteCreator2.
// Client: I'm not aware of the creator's class, but it still works.
// Creator: The same creator's code has just worked with {Result of the ConcreteProduct2}
```

Applicability

- Use when do not know beforehand the exact types and dependencies of the objects
- Use when providing library or framework with a way to extend its internal components
- Use when saving system resources by reusing existing objects instead of rebuilding them each time

Pros and Cons

✓ Avoid tight coupling between the creator and the concrete products

✓ Single Responsibility Principle: move the product creation code into one place in the program, making the code easier to support

✓ Open/Closed Principle: introduce new types of products into the program without breaking existing client code

✕ The code may become more complicated since a lot of new subclasses are introduced to implement the pattern. The best case scenario is when introducing the pattern into an existing hierarchy of creator classes

References

- Slidev: https://github.com/slidevjs/slidev
- Refactoring Guru: https://refactoring.guru/design-patterns/factory-method