SE 350 - Object-Oriented Software Development

Software Design Patterns: Factory Method

Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Factory Method
- Understand the pattern

GoF Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	**Factory Method**	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

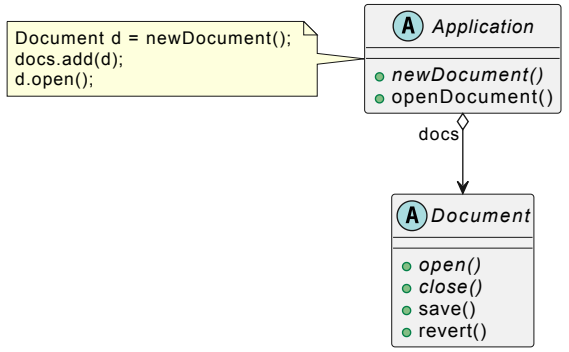
Introduction

Factory Method provides an interface for creating objects and defers instantiation to subclasses

- Creational design pattern
- For framework/library implementation
  - Framework phrased in terms of interfaces and abstract classes
  - Framework only knows *when* to instantiate, not *what*
  - Concrete subclasses of the framework decide *what* to instantiate

Problem illustration: `Application` and `Document` are a framework for implementing document-processing applications.

❓ What kind of specific `Document` should application create?



Overview

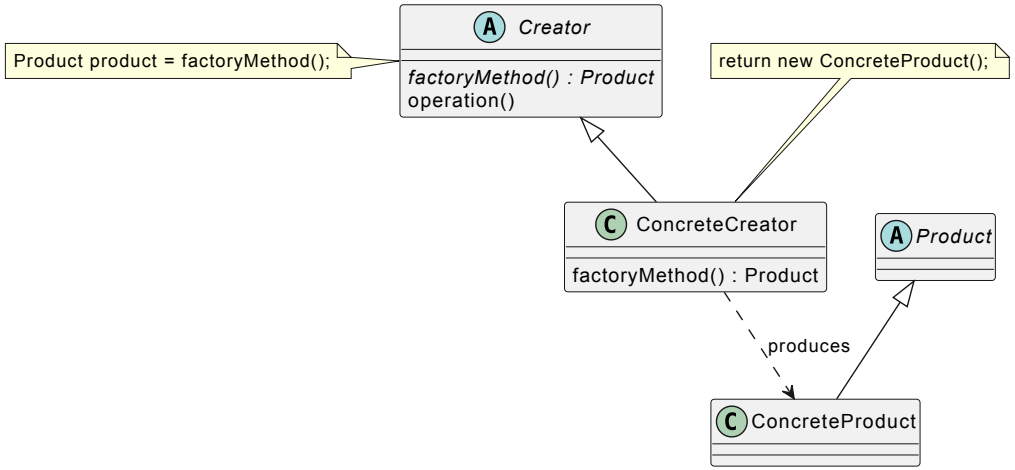
⚠ Problem

- Framework to standardize design for a variety of applications
- Allow individual applications to provide their own domain objects

Intent

- Interface for creating objects
- "Dynamically-dispatched" constructor
- Avoid hardcoded `new` creation

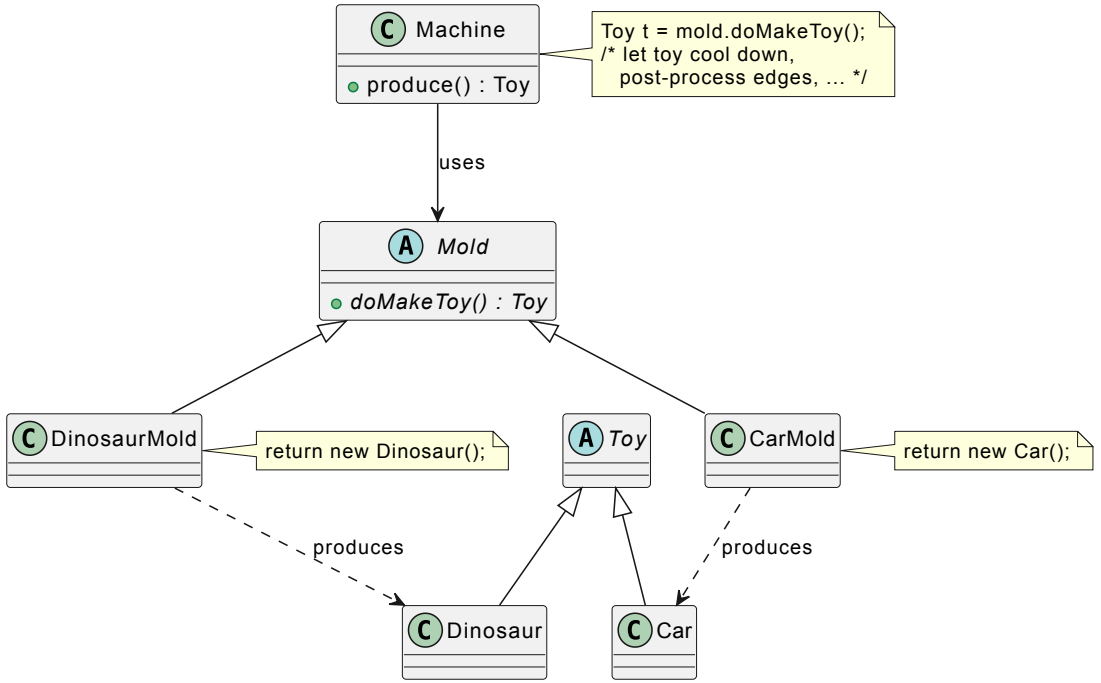
Structure



- Product interface or abstract class
- Concrete products
- Creator: provides standardized `operation` and uses `factoryMethod`
- Concrete creators: instantiate products

Factory Method Real-World Example

- Machine to manufacture plastic toys
- Molds define diferent shapes
- Machine injects plastic into molds of desired shape



Implementation

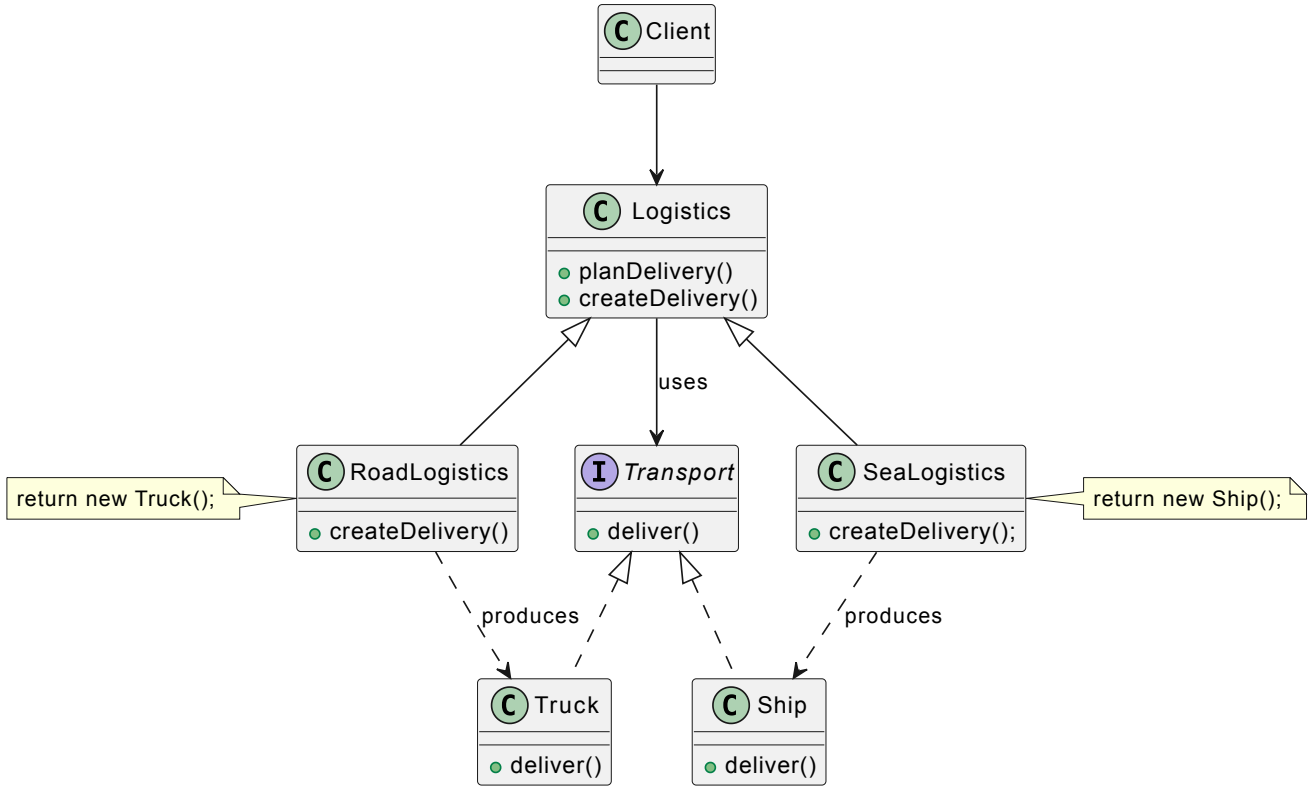
- All products share the same interface `Product`
- Implement a `Creator` with factory method (return type `Product`)
- Add one subclass of `Creator` for each subclass of `Product`
- Replace references to concrete product constructors with calls to factory method
- Naming convention: use a method name that suggests factory method, e.g., `Class doMakeClass`

```
public abstract class MazeGame {
  protected abstract Room doMakeRoom();
}
public abstract class Room {}

// Concrete creator and product: dungeon game
public class DungeonCrawler extends MazeGame {
  @Override protected Room doMakeRoom() {
    return new Chamber();
  }
}
public class Chamber extends Room {}

// Concrete creator and product: enchanted maze game
public class EnchantedMazeGame extends MazeGame {
  @Override protected Room doMakeRoom() {
    return new EnchantedRoom();
  }
}
public class EnchantedRoom extends Room {}
```

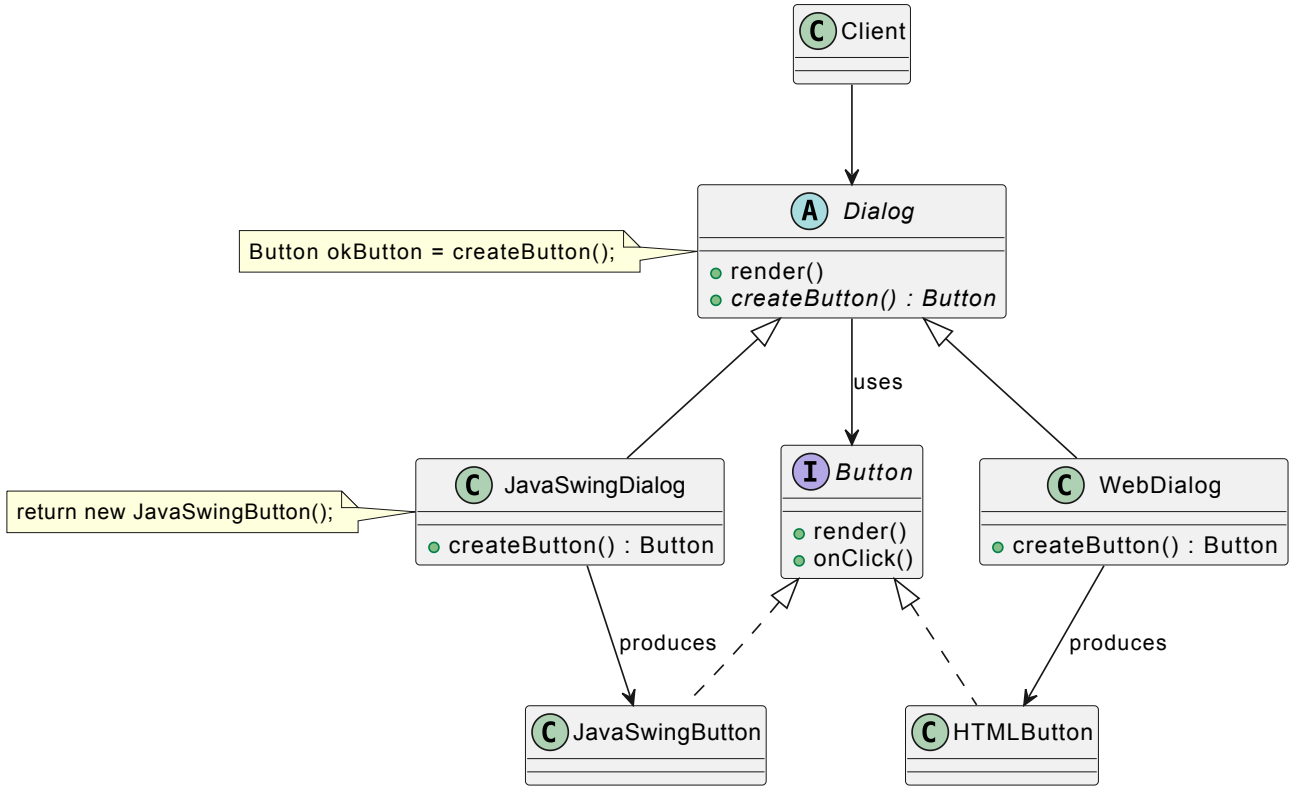
Factory Method Variations

- Abstract `Creator` vs. concrete `Creator` with default implementation of `factoryMethod`
- Parameterized factory methods: create multiple kinds of products with a single factory method
- Reflection: factory method returns `Class` to instantiate
- Type parameterization instead of subclassing: requires instantiable generics/templates (supported e.g., in C++, but not in Java)

Factory Method Example: Logistics



Factory Method Example: Cross-Platform UI



Refactoring to Factory Method

Code Smell: Classes in a hierarchy implement a method similarly, except object creation

Before: Two test cases are implemented similarly

```
public class XMLBuilderTests extends TestCase {
  public void testAddRootSibling() {
    XMLBuilder builder = new XMLBuilder("orders");
    builder.addBelow("orders", "order");
    try {
      builder.addSibling("orders", "customer");
      fail("Expected to fail: add sibling to root");
    } catch (InvalidArgumentException e) { /* expected exception */ }
  }
}

public class DOMBuilderTests extends TestCase {
  public void testAddRootSibling() {
    DOMBuilder builder = new DOMBuilder("orders");
    builder.addBelow("orders", "order");
    try {
      builder.addSibling("orders", "customer");
      fail("Expected to fail: add sibling to root");
    } catch (InvalidArgumentException e) { /* expected exception */ }
  }
}
```

▶ After Refactoring

Discussion

Benefits

- Reduces coupling
- Promotes the Single Responsibility Principle
- Promotes the Open/Closed Principle

Drawbacks

- May introduce many subclasses of `Creator`
- When using parameterized factory methods to avoid subclassing, may violate Open/Closed Principle