



# Creational Patterns

# Creational Patterns Introduction



- Creational design patterns abstract the instantiation process.
- They help make a system independent of how its objects are created, composed, and represented
  - ▣ They all encapsulate knowledge about which concrete classes the system uses
  - ▣ They hide how instances of these classes are created and put together

# Creational Patterns



- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

# Factory Method



# Challenge

- There are many kinds of houses in the game. Each of them have different looking.

<b>House</b>
JungleHouse
BeachHouse

# First Attempt



- ▣ use if-else to check which kind of house player is setting and then generate the object.

```
if (House == "JungleHouse"){  
    // processes to generate JungleHouse  
    return JungleHouse;  
}  
else if (House == "BeachHouse"){  
    // process to generate BeachHouse  
    return BeachHouse;  
}
```

# Factory Method



- Problem: We need to generate one kind of class (abstract class), but each concrete classes need different implementation.
- Think: what's the effort if you want to add or delete some types?
- Target: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

# Structure

## Product

defines the interface of objects created by factory method

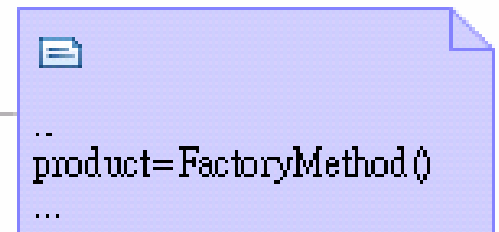
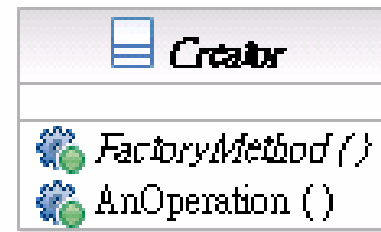


## ConcreteProduct

implements the Product interface

## Creator

declares the factory method returning an object of type Product



## ConcreteCreator

overrides the factory method to return an instance of a ConcreteProduct



# Participants



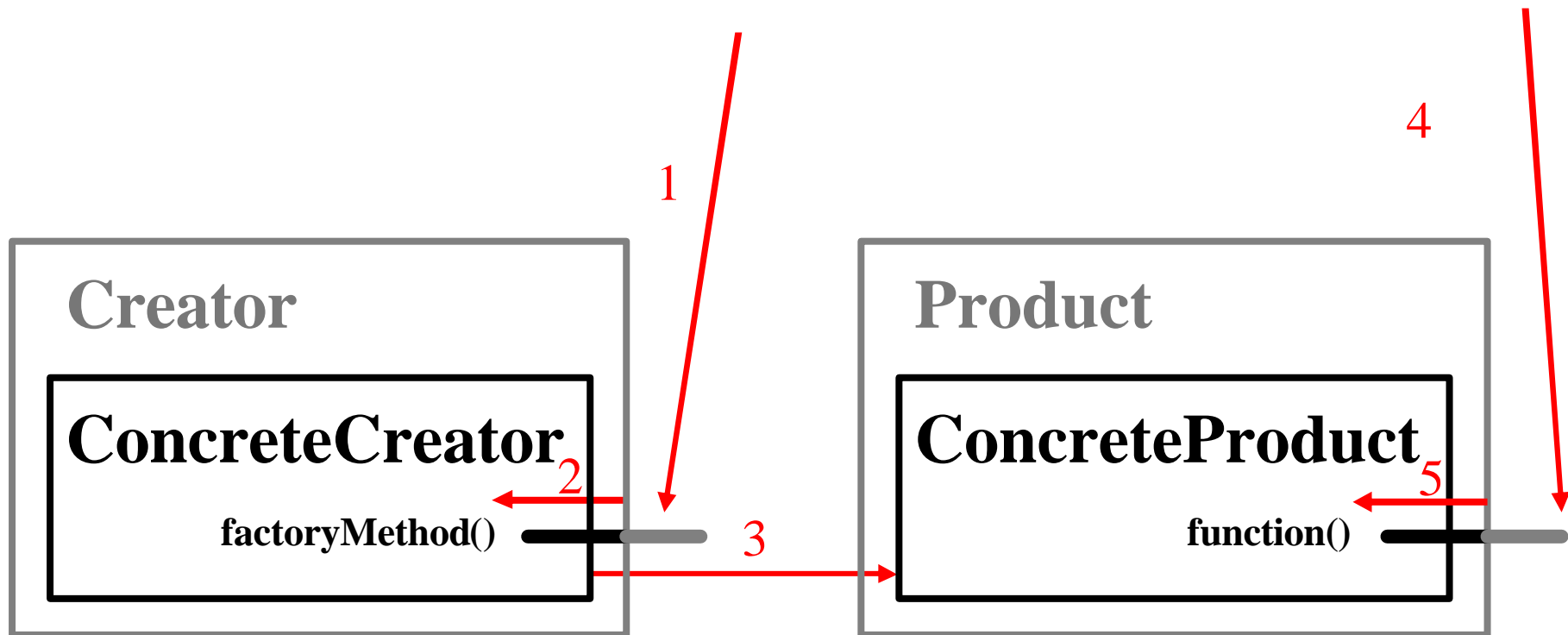
- ❑ Class **Product** defines the interface of objects created by factory method.
- ❑ Class **ConcreteProduct** implements the Product interface.
- ❑ Class **Creator** declares the factory method returning an object of type Product.
- ❑ Class **ConcreteCreator** overrides the factory method to return an instance of a ConcreteProduct.

# Interactions between Participants

---

1. Client calls **Creator**'s factory method
2. **ConcreteCreator** implements **Creator** interfaces, so it will be **ConcreteCreator** to perform factory method
3. **ConcreteCreator** generates and returns **ConcreteProduct** in factory method
4. Client calls **Product**'s function
5. **ConcreteProduct** implements **Product** interfaces, so Client's call will actually performed by **ConcreteProduct**

# Interaction Flow



# Applicability



- Use the Factory Method pattern when
  - ▣ a class cannot anticipate the class of objects it must create.
  - ▣ a class wants its subclasses to specify the objects it creates.
  - ▣ classes **delegate responsibility** to one of several helper **subclasses**, and you want to localize the knowledge of which helper subclass is the delegate.

# Consequences

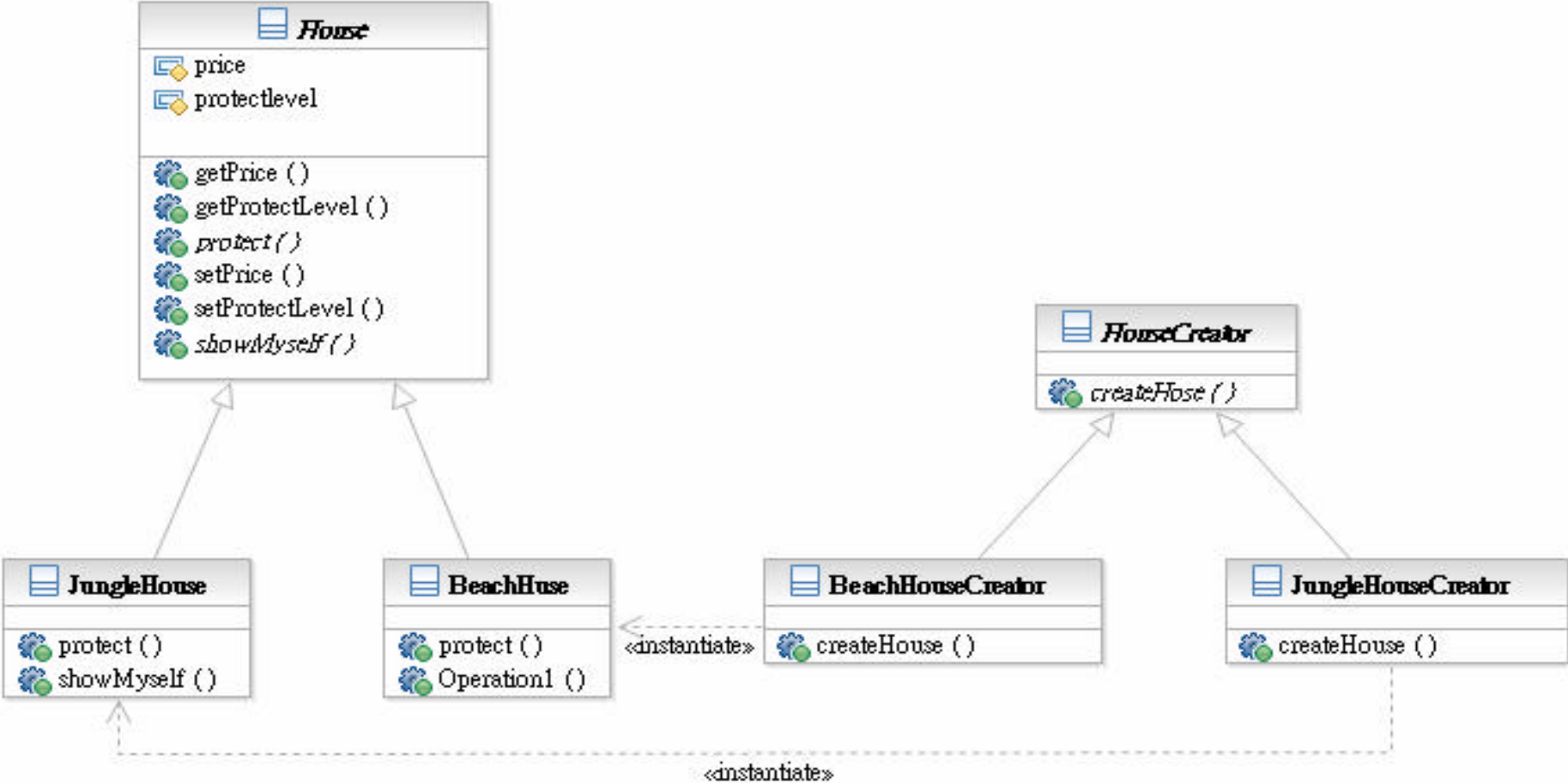


- **It eliminates the need to bind application-specific classes in your code.** The code only deals with Product interface.
- **It provides hooks for subclasses.** Factory Method gives subclasses a hook for providing an extended version of an object. .

# Apply Factory Method Pattern

<b>HouseCreator</b> <b>(Creator)</b>	<b>House</b> <b>(Product)</b>
JungleHouseCreator <b>(ConcreteCreatorA)</b>	JungleHouse <b>(ConcretetProductA)</b>
BeachHouseCreator <b>(ConcreteCreatorB)</b>	BeachHouse <b>(ConcreteProductB)</b>

# Structure of Sample



# Sample Code Flow

- Target: Generate JungleHouse and perform showMyself()
  1. main() calls **HouseCreator**'s createHouse() method to generate **House**
  2. **JungleHouseCreator** implements **HouseCreator** interfaces, so it will be **JungleHouseCreator** to perform createHouse()
  3. **JungleHouseCreator** generates **JungleHouse** using its own implementation in createHouse(). Then, return JungleHouse
  4. Client calls House's showMyself() function
  5. **JungleHouse** implements **House**'s interfaces, so Client's call will actually performed by **JungleHouse**'s ()



# Related Patterns



- **Abstract Factory** is often implemented with factory methods
- Factory methods are usually called within **Template Methods**

# Abstract Factory



# Challenge

- There are many decorations to setup a farm and each decoration have some “functions”. There are many different styles of decorations and the function of decoration will vary accordingly.
- The fence of different decoration style has different protection way

<b>Decoration Style</b>	<b>House</b>	<b>Fence</b>
Jungle	JungleHouse	JungleFence
Beach	BeachHouse	BeachFence

# First Attempt

- use if-else statement to cover all possible style and in each style, we new all kinds of decorations.

```
if (DecorationStyle == "Jungle"){
    if ( Decoration == "House"){
        return JungleHouse;
    }
    else if (Decoration = "Fence"){
        return JungleFence;
    }
}
else if (DecorationStyle == "Beach"){
    ...
}
```

# Abstract Factory



- Problem: We need to support multiple families of products. How can clients manipulate those products without know which family they generate?
- Think: what's the effort if we need to add one more family of products.
- Target: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

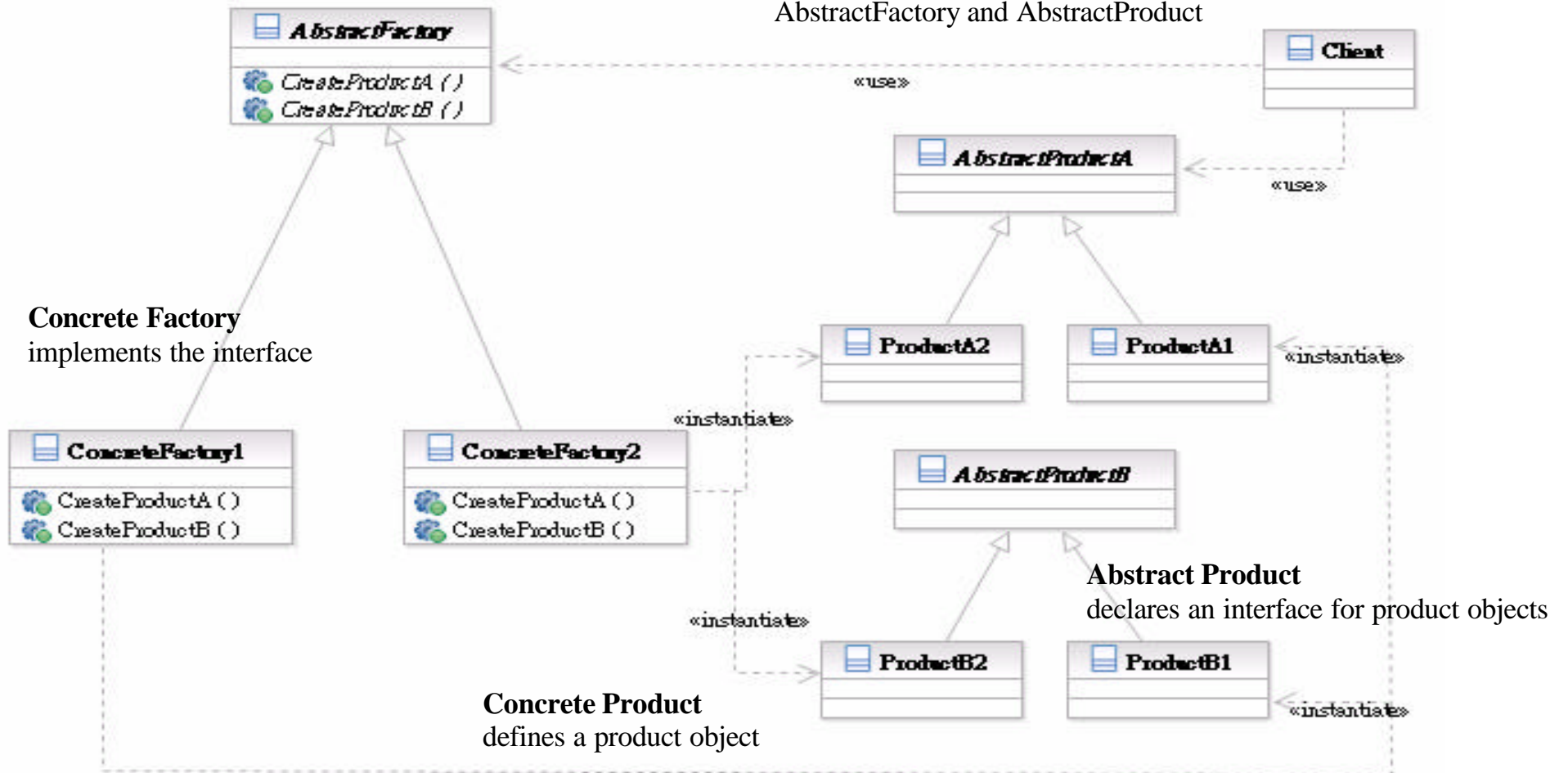
# Structure

## Abstract Factory

declares an interface for creating product objects

## Client

uses only the interface defined by AbstractFactory and AbstractProduct



# Participants

- Class **AbstractFactory** declares an interface for creating product objects;
- Class **ConcreteFactory** implements the interface;
- Class **AbstractProduct** declares an interface for product objects;
- Class **ConcreteProduct** defines a product object;
- Class **Client** uses only the interface defined by **AbstractFactory** and **AbstractProduct**

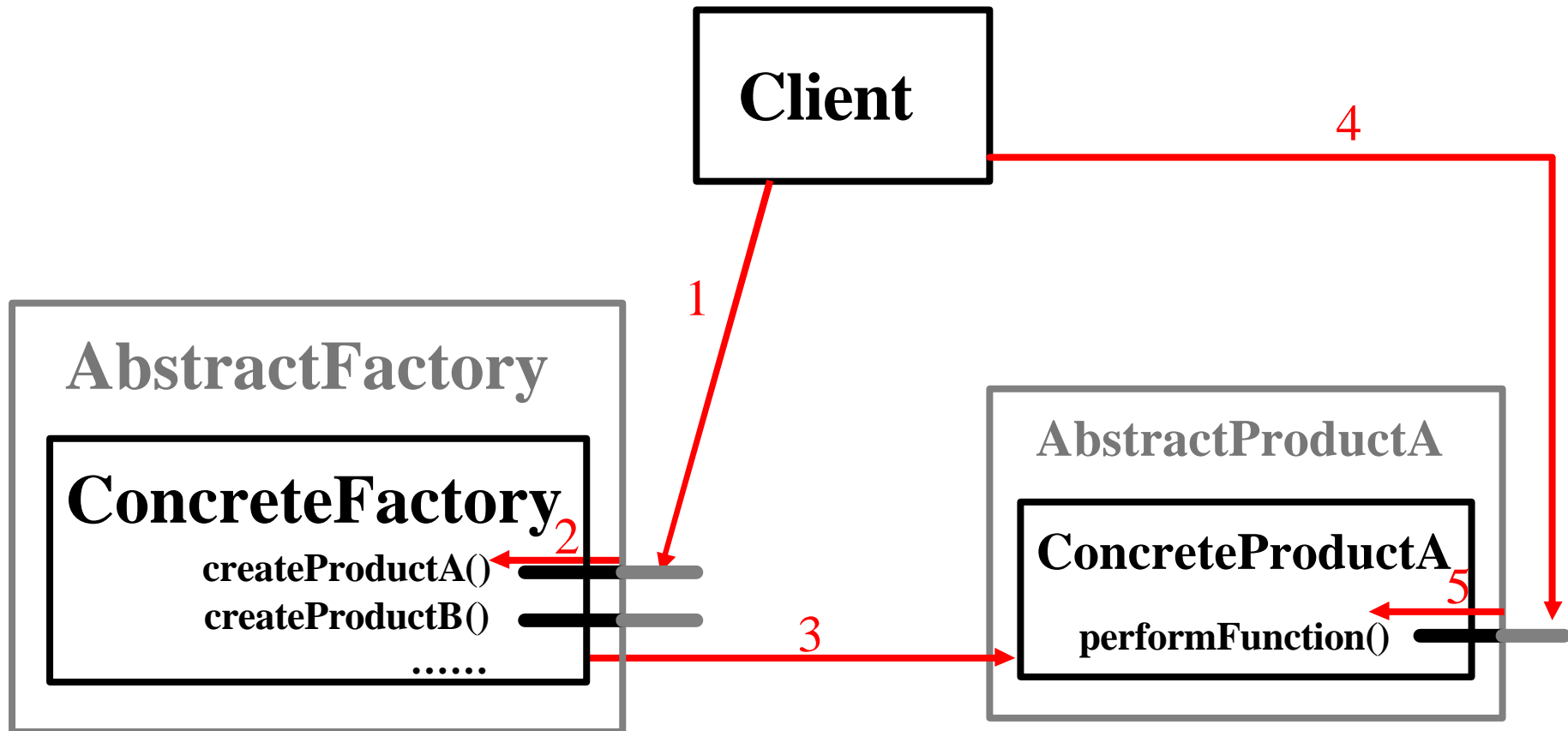
# Interactions between Participants



1. **Client** calls interfaces of **AbstractFactory**
2. **ConcreteFactory** implements **AbstractFactory** interfaces, so **Client**'s call will actually be performed by **ConcreteFactory**
3. **ConcreteFactory** generates **ConcreteProducts** of this family as a result of **Client**'s call
4. **Client** calls interfaces of **AbstractProduct**
5. **ConcreteProduct** implements **AbstractProduct** interfaces, so **Client**'s call will actually be performed by **ConcreteProduct**



# Interaction Flow



# Applicability



- Use the Abstract Factory pattern when
  - ▣ a system should be **independent** of how its products are created, composed, and represented.
  - ▣ a system should be configured with one of **multiple families** of products.
  - ▣ a family of related product objects is designed to be used together, and you need to enforce this **constraint**.
  - ▣ you want to provide a class library of products, and you want to reveal just their **interfaces, not implementations**.

# Consequences

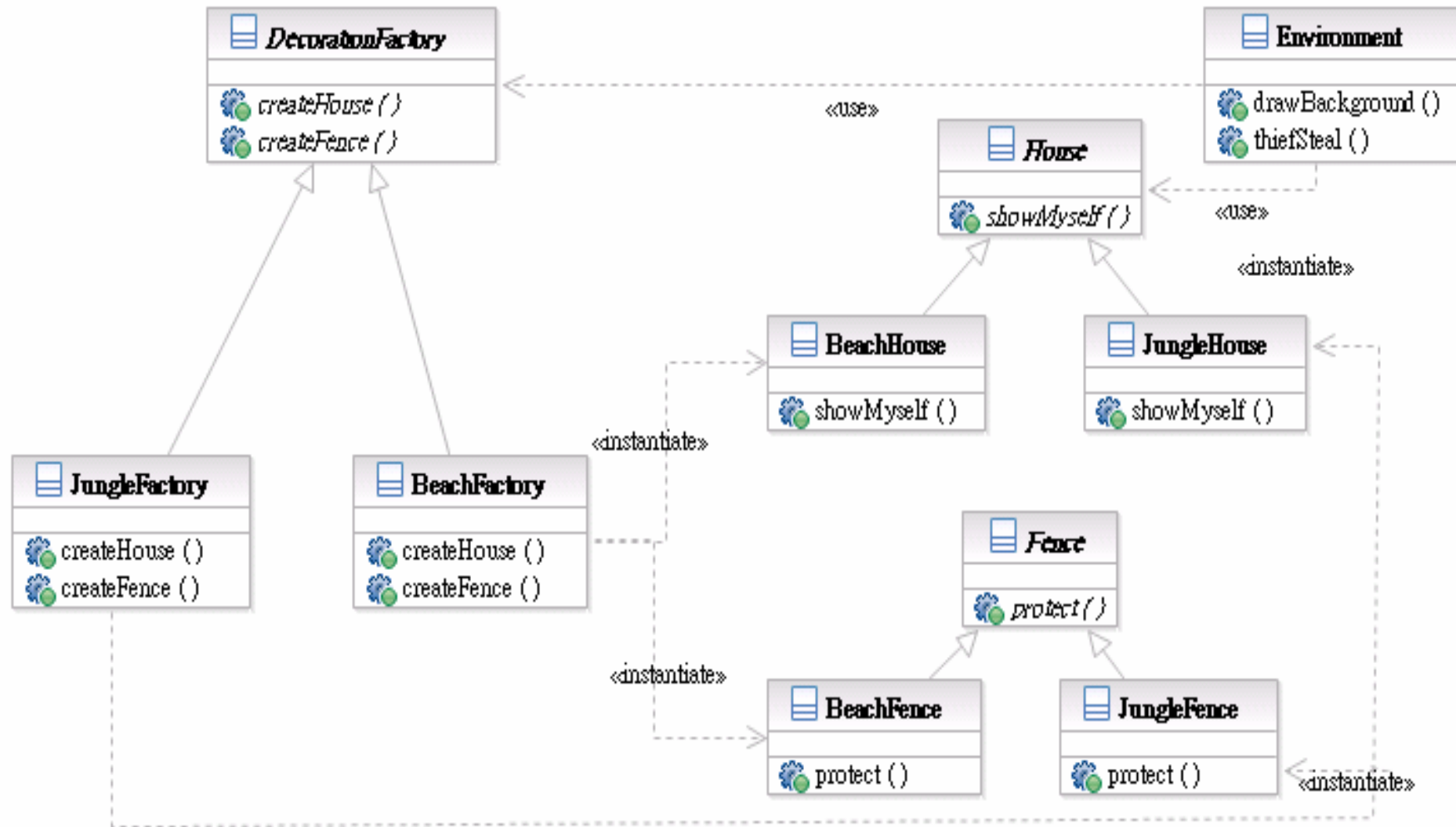


- ❑ **It isolates concrete classes.** Clients manipulate instances via their abstract interfaces.
- ❑ **It makes exchanging product families easy.** The class of a concrete factory appears only when it is instantiated.
- ❑ **It promotes consistency among products.** It is enforced in concrete factories.
- ❑ **Supporting new kinds of products is difficult.** AbstractFactory interface fixes the set of products that can be created.

# Apply Abstract Factory Pattern

<b>Decoration Style</b>	<b>Decoration-Factory</b> <b>(AbstractFactory)</b>	<b>House</b> <b>(AbstractProductA)</b>	<b>Fence</b> <b>(AbstractProductB)</b>
Jungle	Jungle-Factory <b>(ConcreteFactory1)</b>	JungleHouse <b>(ConcreteProductA1)</b>	JungleFence <b>(ConcreteProductB1)</b>
Beach	Beach-Factory <b>(ConcreteFactory2)</b>	BeachHouse <b>(ConcreteProductA2)</b>	BeachFence <b>(ConcreteProductB2)</b>

# Structure of Sample



# Sample Code Flow

- Target: Generate Jungle Style Environment
  1. Assign **JungleFactory** to **Environment**
  2. **Environment** calls interfaces of **DecorationFactory** to generate **House** and **Fence**
  3. **JungleFactory** is the one really being called and generate **JungleHouse** and **JungleFence**
  4. **Environment** calls interfaces of **House** or **Fence**
  5. **JungleHouse** or **JungleFence** is really called and performs functions

# Related Patterns



- AbstractFactory classes are often implemented with factory methods **Factory Method**, but they can also be implemented using **Prototype**
- A concrete factory is often a **Singleton**

# Singleton





# Challenge



- We need a clock for each player for them to calculate their time staying in our game.

# First Attempt



- ▣ generate a clock object for each player

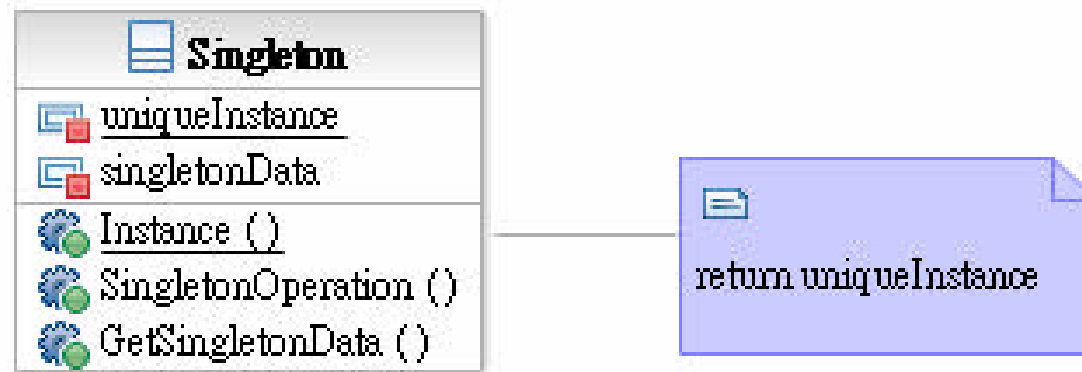
```
getClock() {  
    return new Clock();  
}
```

# Singleton



- Problem: The class only need one instance in our system.
- Think: Is the resource wasted on no necessary duplication object?
- Target: Ensure a class only has one instance, and provide a global point of access to it.

# Structure



## **Singleton**

defines a static member function that lets clients access its unique instance

# Participants



- Class **Singleton** defines a static member function that lets clients access its unique instance.

# Applicability







- Use the Singleton pattern when
  - ▣ there must be **exactly one instance** of a class, and it must be accessible to clients from a well-known access point.

# Consequences



- ❑ **Controlled access to sole instance.** Singleton has strict control over how and when clients access it.
- ❑ **Reduced name space.** It is an improvement over global variable.
- ❑ **Permits a variable number of instances.** It is easy to modify the pattern in case you need more than one instances.
- ❑ **More flexible than class operations.** It is also possible to use static member function to keep the instance. But it would be hard to change the design.

# Structure of Sample

 <b>Clock</b>	
 <u>instance</u>	
 <u>getInstance ()</u>	
 <u>getCurrentTime ()</u>	



# Sample Code Flow



- Target: Get Clock instance
  1. main() calls **Clock.getInstance()** to get the unique instance of **Clock**
  2. main() calls other method of Clock

# Related Patterns



- Many patterns can be implemented using the Singleton pattern. See **Abstract Factory**, **Builder**, and **Prototype**