# Lecture: Design Patterns

## SIT320 - Advanced Algorithms

DEAKIN UNIVERSITY

Dr. Nayyar Zaidi

# About the Lecture

- Introduction

- Design Patterns
  - Singleton
  - Factory
  - Abstract Factory
  - Facade
  - Observer

- Refactoring

# Introduction

# Design Patterns

- Inspired by the work of Christopher Alexander, who first described patterns in Architecture:
  - “Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
- The idea was embraced by computing analysis and design theorists and practitioners
- Martin Fowler’s definition of a pattern is:
  - “An idea that has been useful in one practical context and will probably be useful in others”

# Design Patterns

- Effective patterns provide solutions that are used again and again
  - …. even over thousands of years

**The Coliseum, Rome 1st century**

**The Melbourne Cricket Ground, 20 Century**

# Design Patterns for Software

- The classic work on the application of patterns in software design is:
  - Design Patterns: Elements of Reusable Object- Oriented Software (1995) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (a.k.a. GoF, The Gang of Four)
  - Used object modeling techniques to represent common solutions to problems in the design of OO software, taken from multiple actual systems
  - Since the GoF book came out, there have been many more
- There are also many online resources, this is a good starting point:
  - http://hillside.net/patterns/patterns-catalog
- There are also books on Analysis Patterns
  - Patterns that help to model situations that often arise during analysis
    - Use concepts rather than actual classes to be implemented in code

# Design Patterns for Software

- Design patterns are general, reusable solution to commonly occurring problem within a given context in software design
  - It is not a finished design that can be transformed directly into source code
  - Rather, it is a description or template for how to solve a problem
  - It can be used in many different situations
  - Design patterns are formalised best practice that the programmer can use to solve common problems when designing an application or system

# A Design Pattern is:

- **Smart**
  - an elegant solution not obvious to a novice
- **Well-Proven**
  - has been identified from real OO systems
- **Reusable**
  - is documented in such a fashion that it is easy to reuse
- **Generic**
  - not dependent upon a system, programming language or application domain
- **Simple**
  - is usually quite small, involving only a handful of classes
- **Object-Oriented**
  - built with OO mechanisms such as classes, objects, generalization and polymorphism

# A Design Pattern has:

- **A Pattern Name:**
  - a handle we can use to describe a design problem, its solutions and consequences
- **The Problem:**
  - describes when to apply the pattern. It explains the problem and its context
- **The Solution:**
  - describes the elements which make up the solution and their relationships
- **The Consequences:**
  - the results and trade-offs of using the design pattern

# Categorizing Design Patterns

- **Purpose**
  - **Creational**: concern the process of object creation,
    - e.g. – Abstract Factory, Singleton
  - **Structural**: deal with the composition of classes and objects,
    - e.g. – Adapter, Facade
  - **Behavioural**: characterize the way in which classes or objects interact and distribute responsibility,
    - e.g. Iterator, Observer

- **Scope**
  - **Class:** the pattern is primarily concerned with classes, they deal with the relationships between classes and their sub-classes
    - These relationships are established through Inheritance and are static
  - **Object:** the pattern is primarily concerned with object relationships, which are more dynamic and can change at run-time

# Factory Patterns

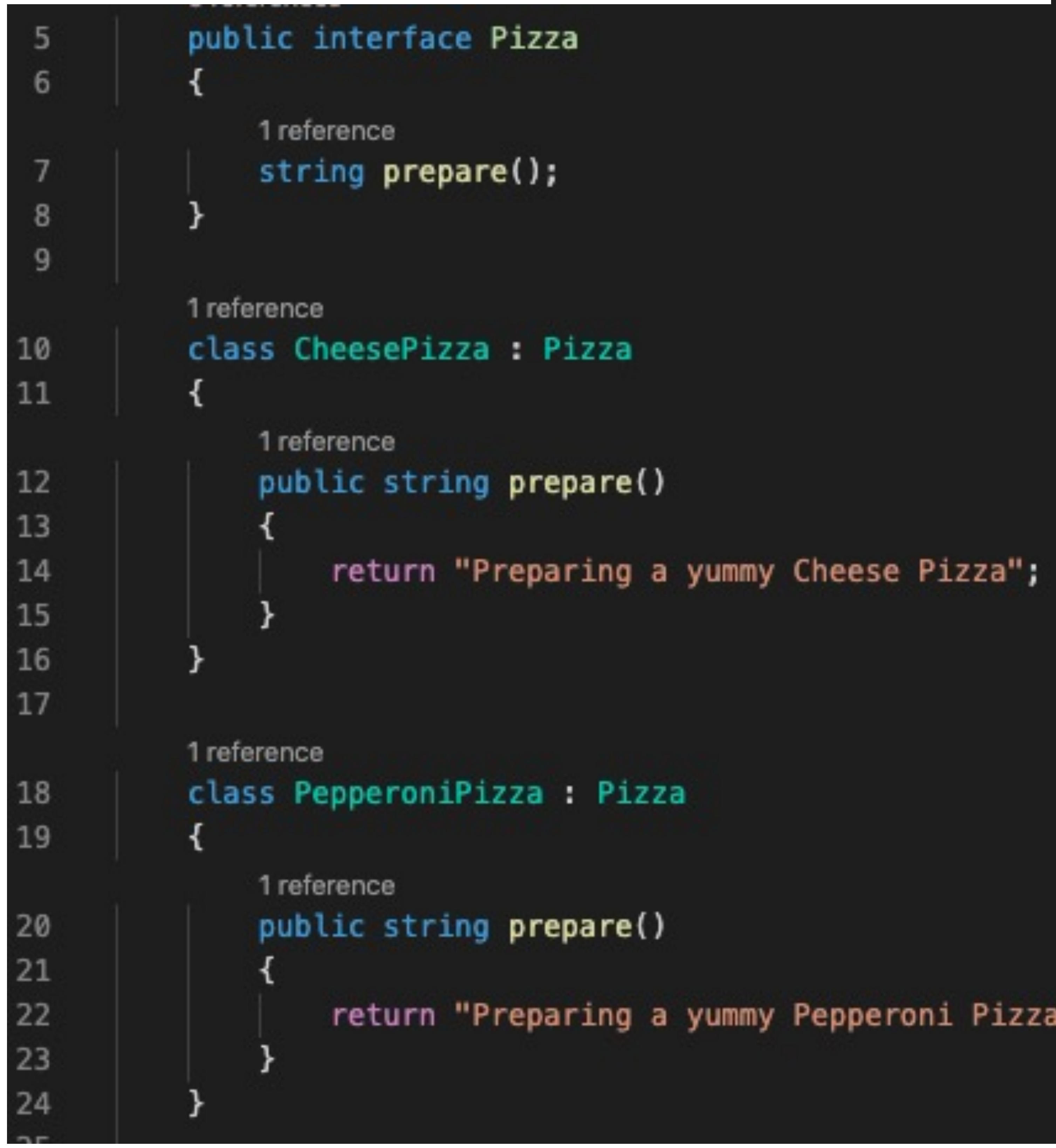
# Factory Patterns

- Three variants
  - Simple Factory
  - Factory Method
  - Abstract Factory

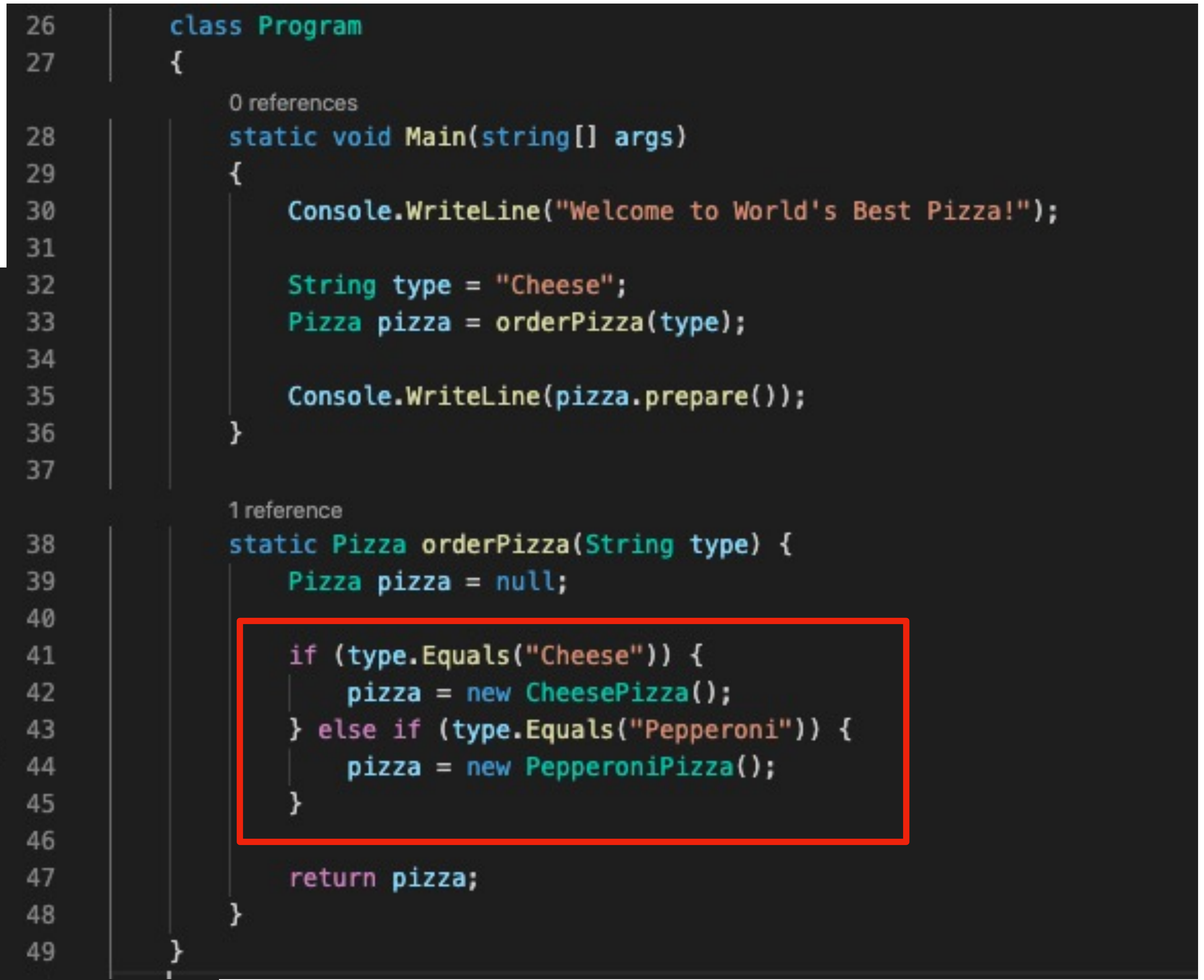
# Simple Factory

- A simple factory is an object for creating other objects
  - We have a factory class which has a method that returns different types of object based on given input
- Motivations:
  - Calling **new** is coding to an implementation (binds your code to a concrete class)
  - We should aim to code (or program) to an abstract class or interface
  - Concrete class are often instantiated at more than on place, therefore, when changes or extensions are made, all the instantiations will have to be changed
    - Error-prone, difficult, messy

# No Factory



```
public interface Pizza
{
    1 reference
    string prepare();
}

1 reference
class CheesePizza : Pizza
{
    1 reference
    public string prepare()
    {
        return "Preparing a yummy Cheese Pizza";
    }
}

1 reference
class PepperoniPizza : Pizza
{
    1 reference
    public string prepare()
    {
        return "Preparing a yummy Pepperoni Pizza";
    }
}
```

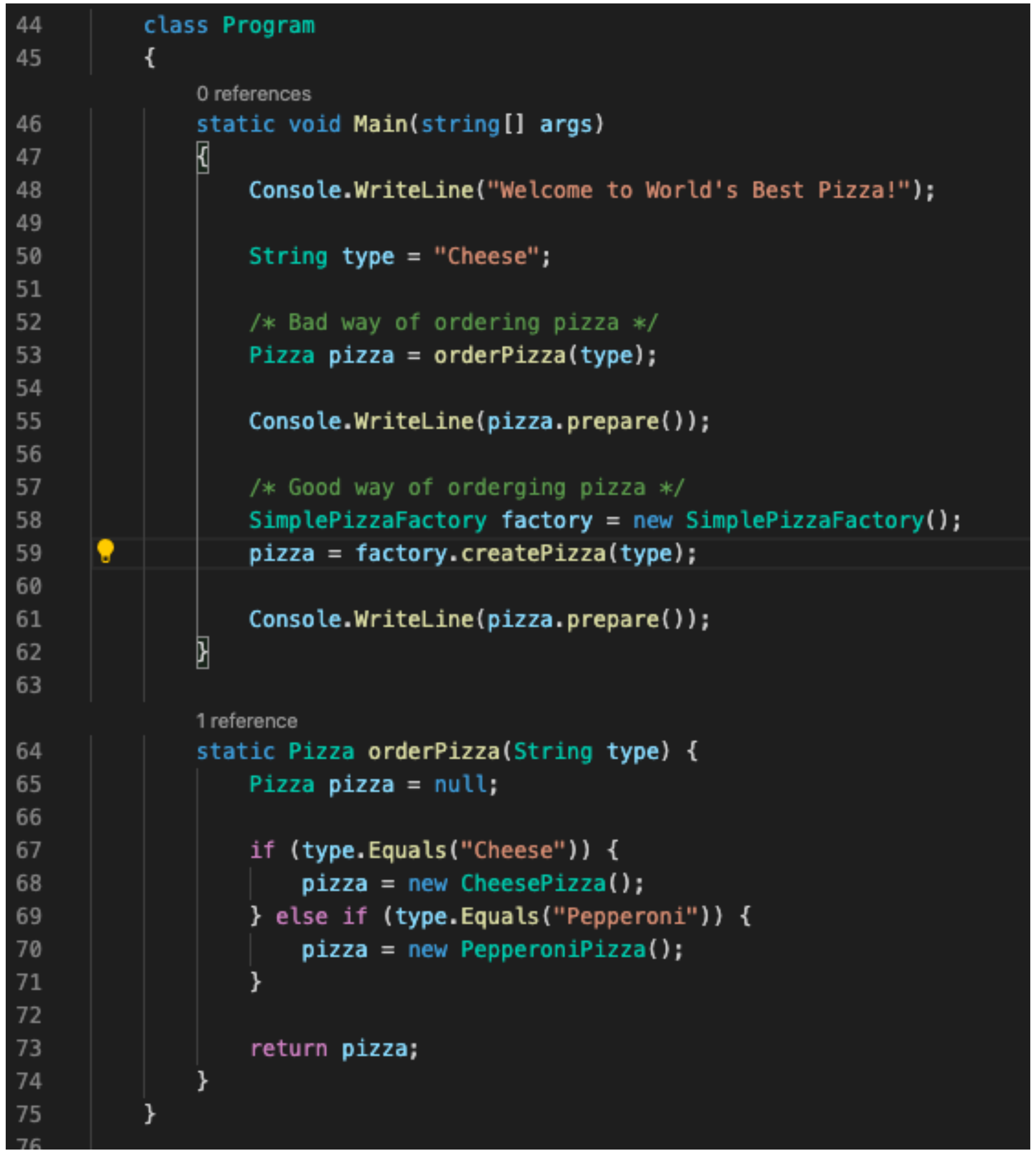


```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Welcome to World's Best Pizza!");

        String type = "Cheese";
        Pizza pizza = orderPizza(type);

        Console.WriteLine(pizza.prepare());
    }

    1 reference
    static Pizza orderPizza(String type) {
        Pizza pizza = null;

        if (type.Equals("Cheese")) {
            pizza = new CheesePizza();
        } else if (type.Equals("Pepperoni")) {
            pizza = new PepperoniPizza();
        }

        return pizza;
    }
}
```

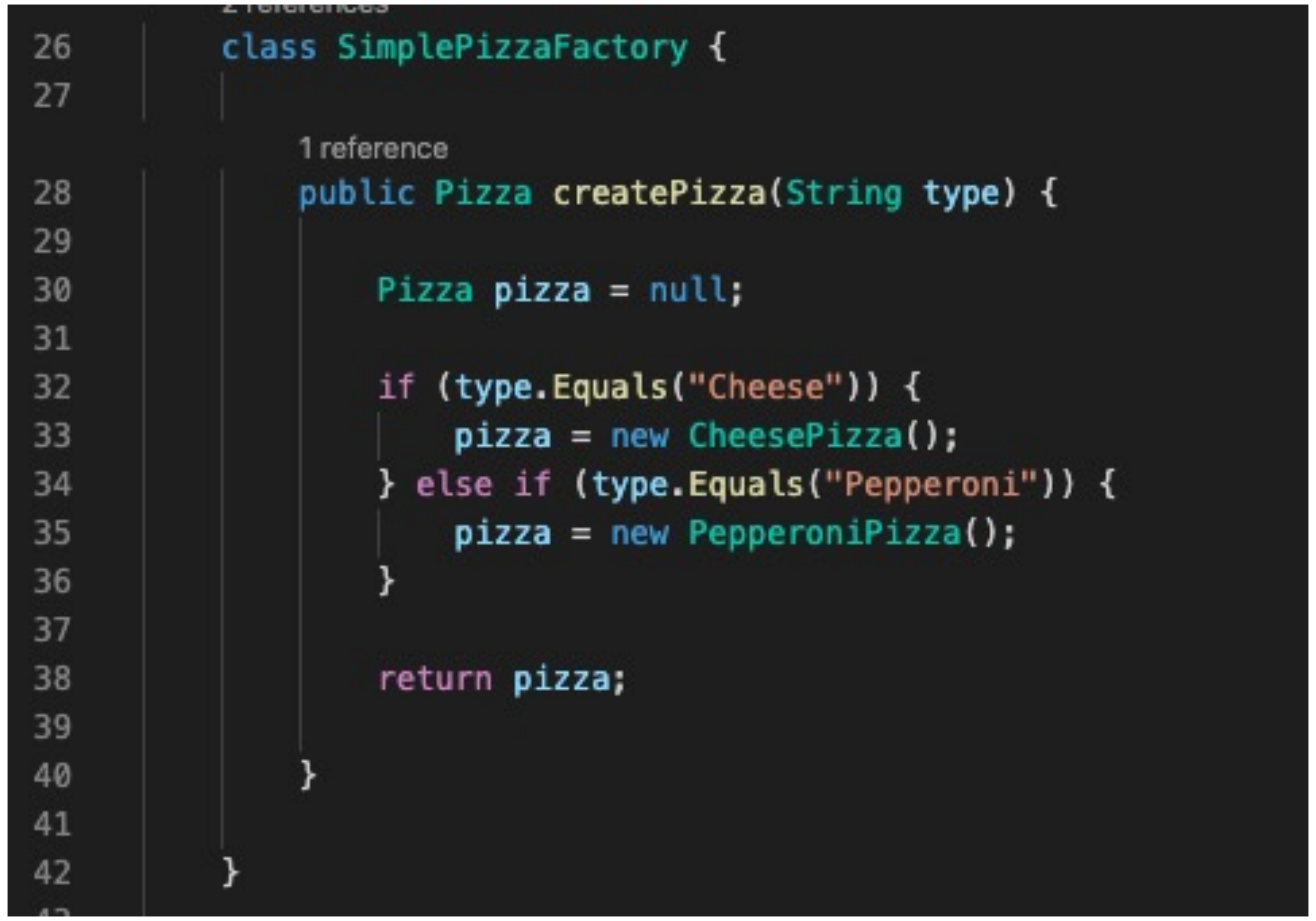
- Problems:
  - What if the name of pizza class changes?
  - What if the constructors of pizza class changes?

**Solution — Better to encapsulate this**

# Simple Factory



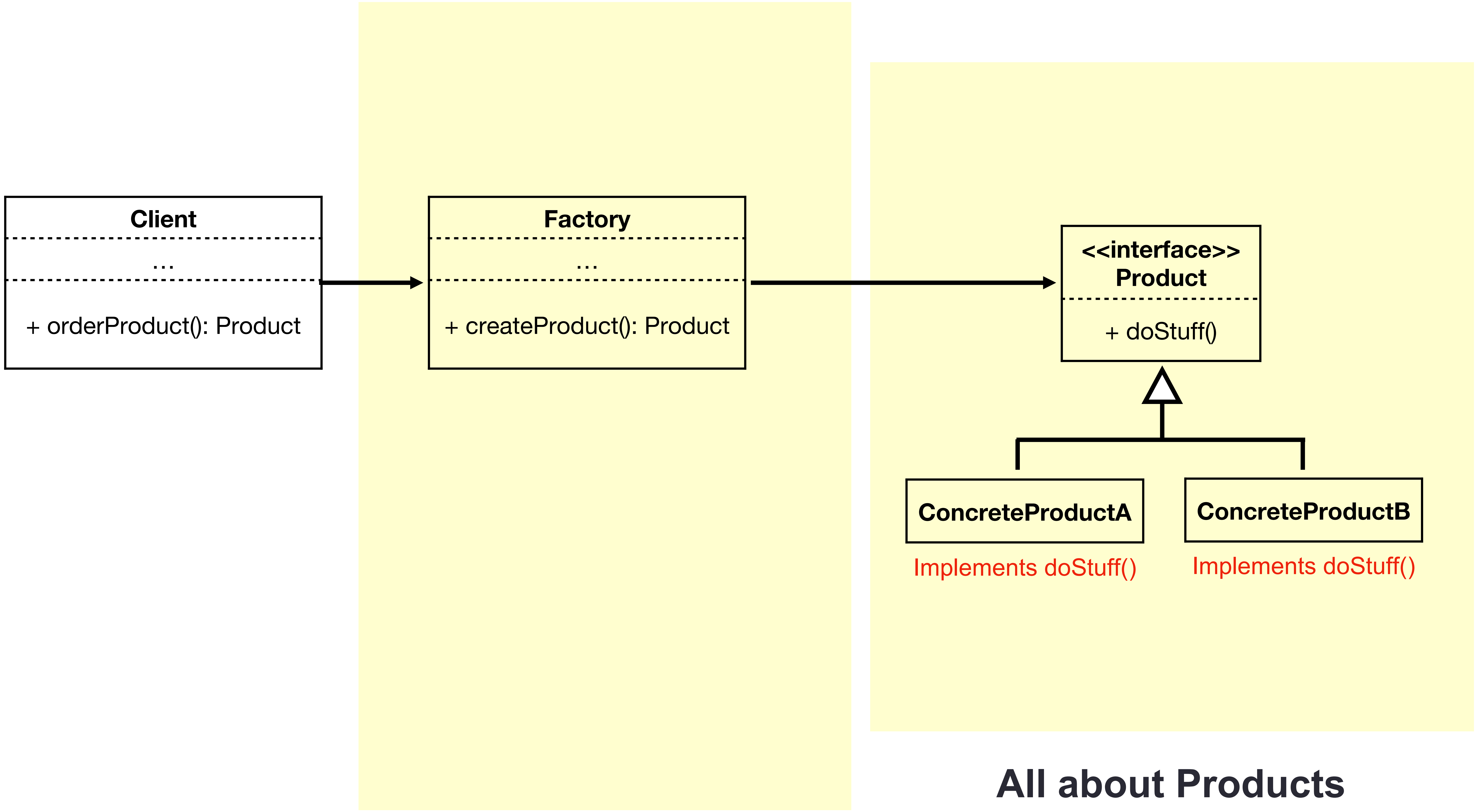
```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Welcome to World's Best Pizza!");

        String type = "Cheese";

        /* Bad way of ordering pizza */
        Pizza pizza = orderPizza(type);

        Console.WriteLine(pizza.prepare());

        /* Good way of orderging pizza */
        SimplePizzaFactory factory = new SimplePizzaFactory();
        pizza = factory.createPizza(type);

        Console.WriteLine(pizza.prepare());
    }

    1 reference
    static Pizza orderPizza(String type) {
        Pizza pizza = null;

        if (type.Equals("Cheese")) {
            pizza = new CheesePizza();
        } else if (type.Equals("Pepperoni")) {
            pizza = new PepperoniPizza();
        }

        return pizza;
    }
}
```



```
2 references
class SimplePizzaFactory {

    1 reference
    public Pizza createPizza(String type) {

        Pizza pizza = null;

        if (type.Equals("Cheese")) {
            pizza = new CheesePizza();
        } else if (type.Equals("Pepperoni")) {
            pizza = new PepperoniPizza();
        }

        return pizza;

    }

}
```

- **Summary of Simple Factory:**
  - Pull the code that builds the instances out and put it into a separate class
  - Identify the aspects of your application that vary and separate from what stays the same

# Simple Factory (Structure)



Implements doStuff()

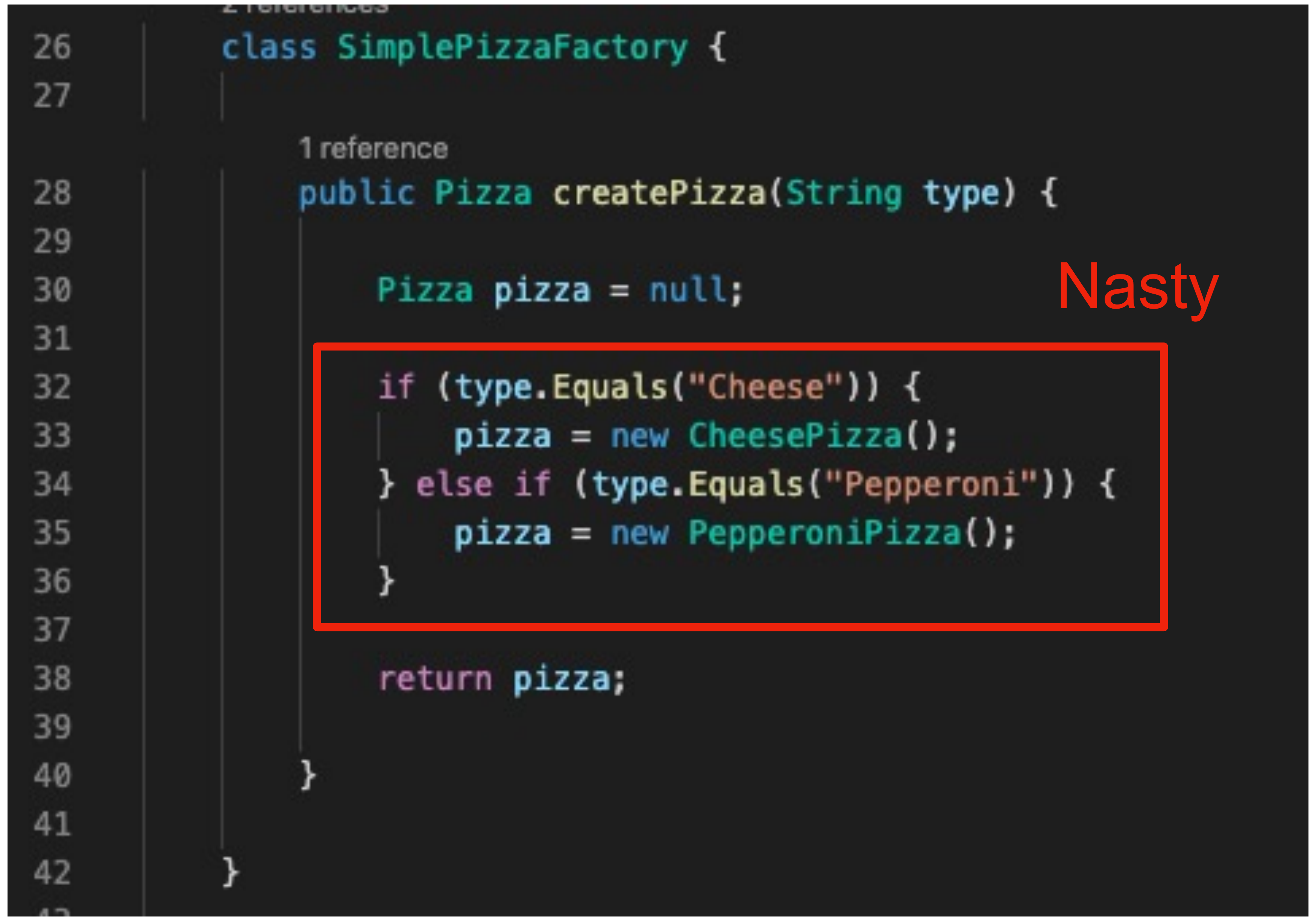
Implements doStuff()

All about Products

All about Creations

# Factory Method Pattern

- Factory method provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created
    - Lets class defer instantiation to sub-classes

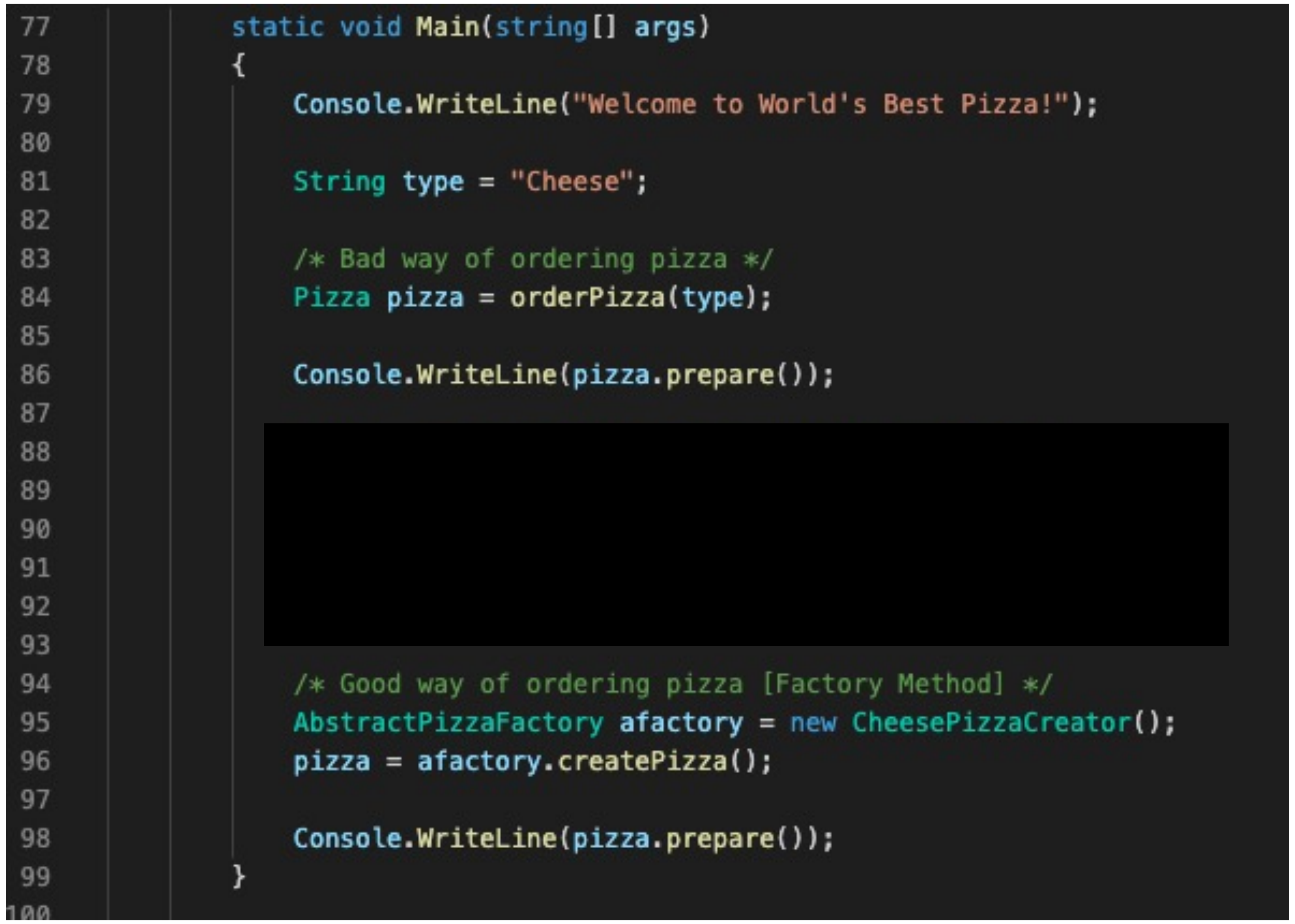


```
class SimplePizzaFactory {

    1 reference
    public Pizza createPizza(String type) {

        Pizza pizza = null;

        if (type.Equals("Cheese")) {
            pizza = new CheesePizza();
        } else if (type.Equals("Pepperoni")) {
            pizza = new PepperoniPizza();
        }

        return pizza;

    }

}
```

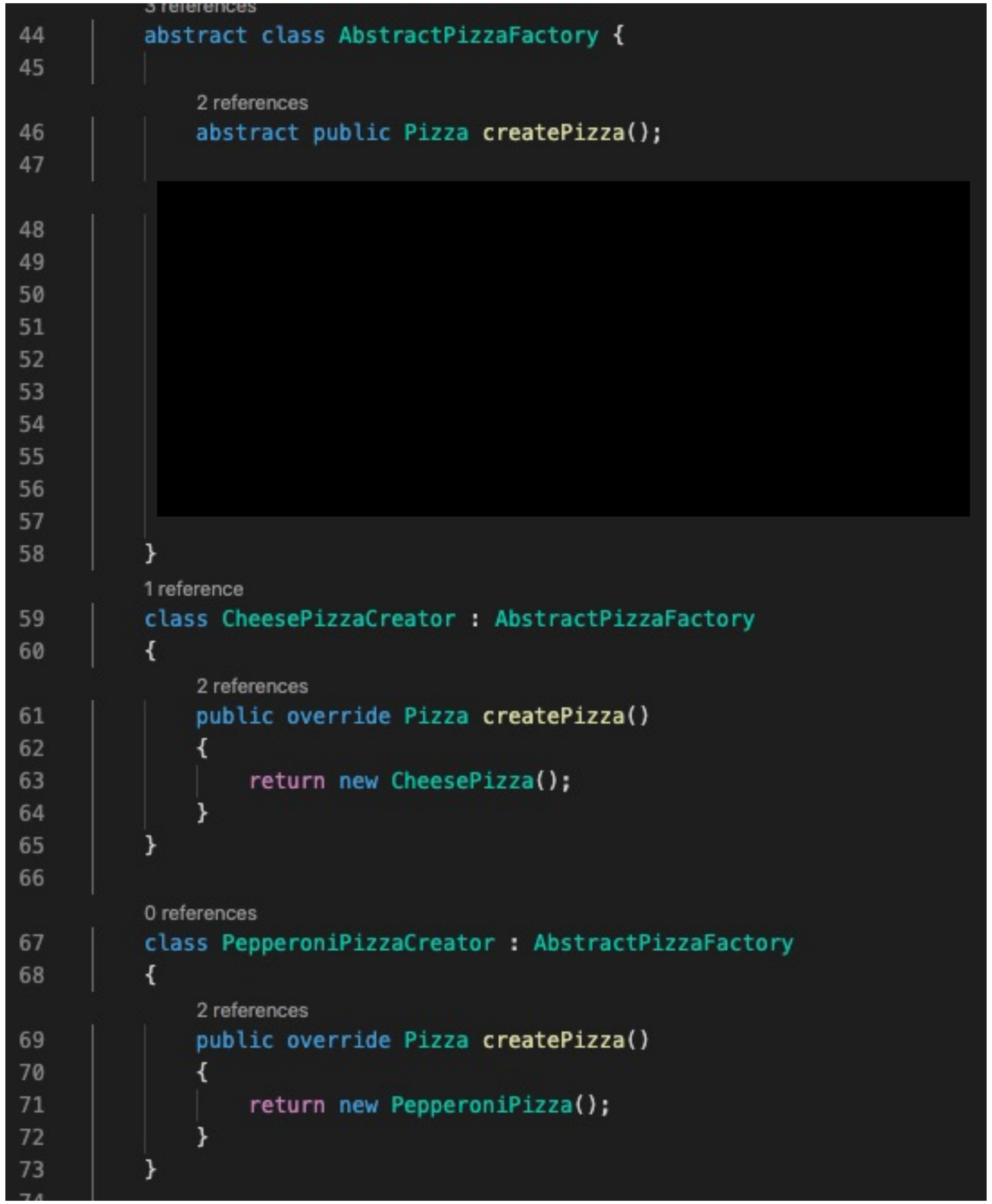
Nasty

- **Problems:**
    - Every time you add a new product, you will have to add an if statement
    - Your dependence is on Factory class, which is still concrete
        - Isn’t that frown-upon?

# Factory Method Pattern

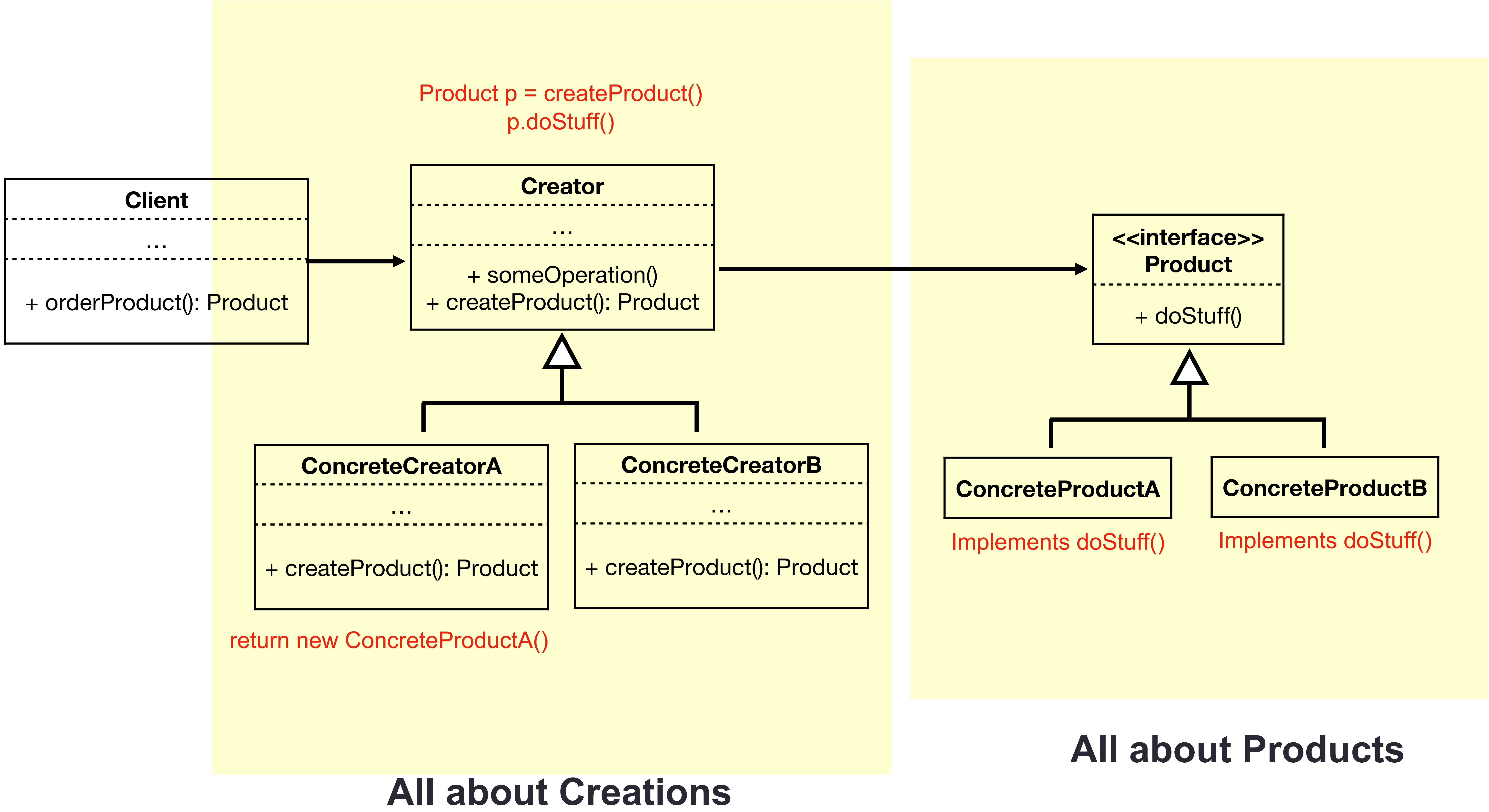


```
    static void Main(string[] args)
    {
        Console.WriteLine("Welcome to World's Best Pizza!");

        String type = "Cheese";

        /* Bad way of ordering pizza */
        Pizza pizza = orderPizza(type);

        Console.WriteLine(pizza.prepare());

[redacted]

        /* Good way of ordering pizza [Factory Method] */
        AbstractPizzaFactory afactory = new CheesePizzaCreator();
        pizza = afactory.createPizza();

        Console.WriteLine(pizza.prepare());
    }
```

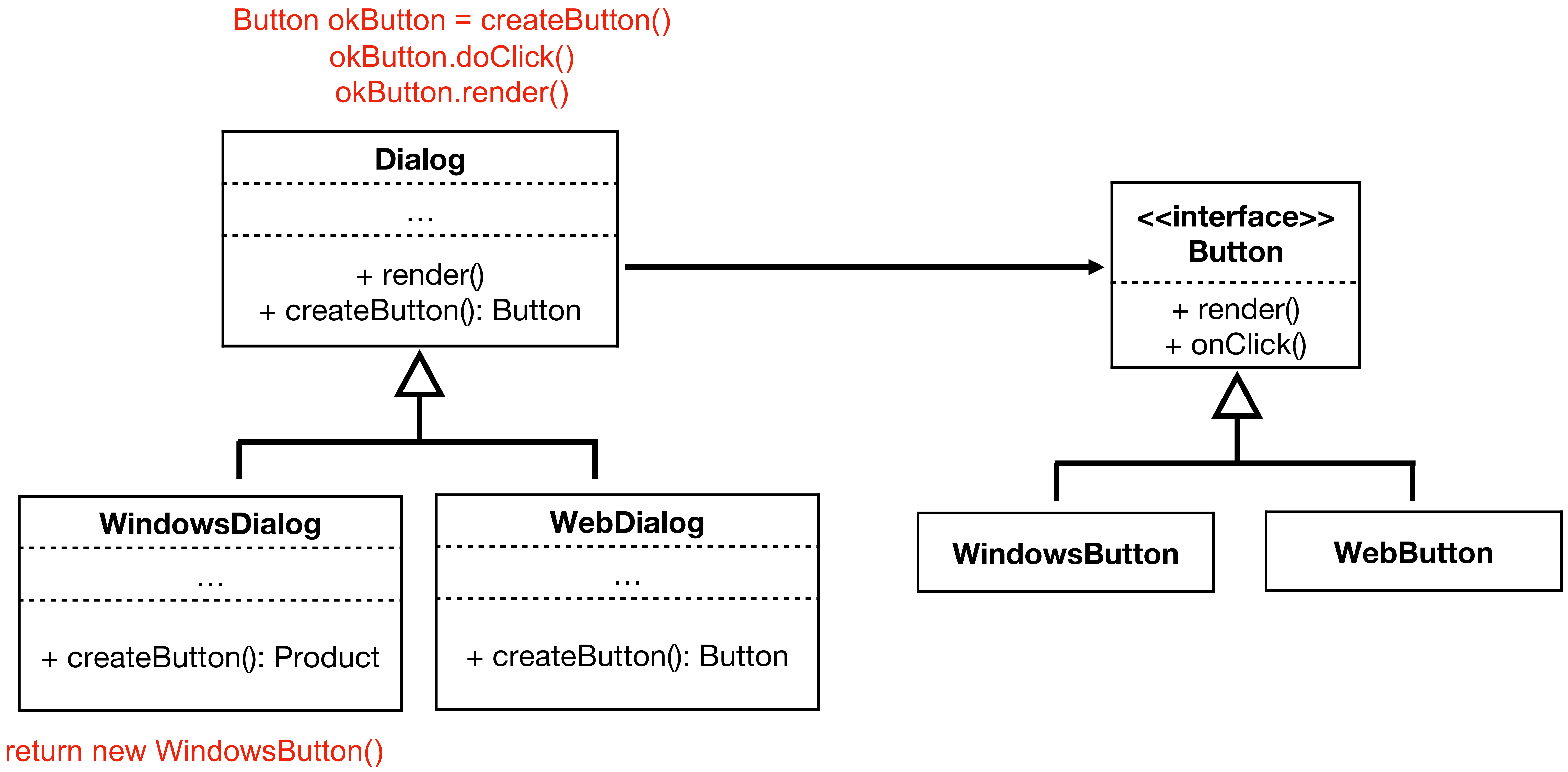


```
3 references
abstract class AbstractPizzaFactory {

    2 references
    abstract public Pizza createPizza();

[redacted]

}
1 reference
class CheesePizzaCreator : AbstractPizzaFactory
{
    2 references
    public override Pizza createPizza()
    {
        return new CheesePizza();
    }
}

0 references
class PepperoniPizzaCreator : AbstractPizzaFactory
{
    2 references
    public override Pizza createPizza()
    {
        return new PepperoniPizza();
    }
}
```

# Factory Method (Structure)



# Factory Method (Example)



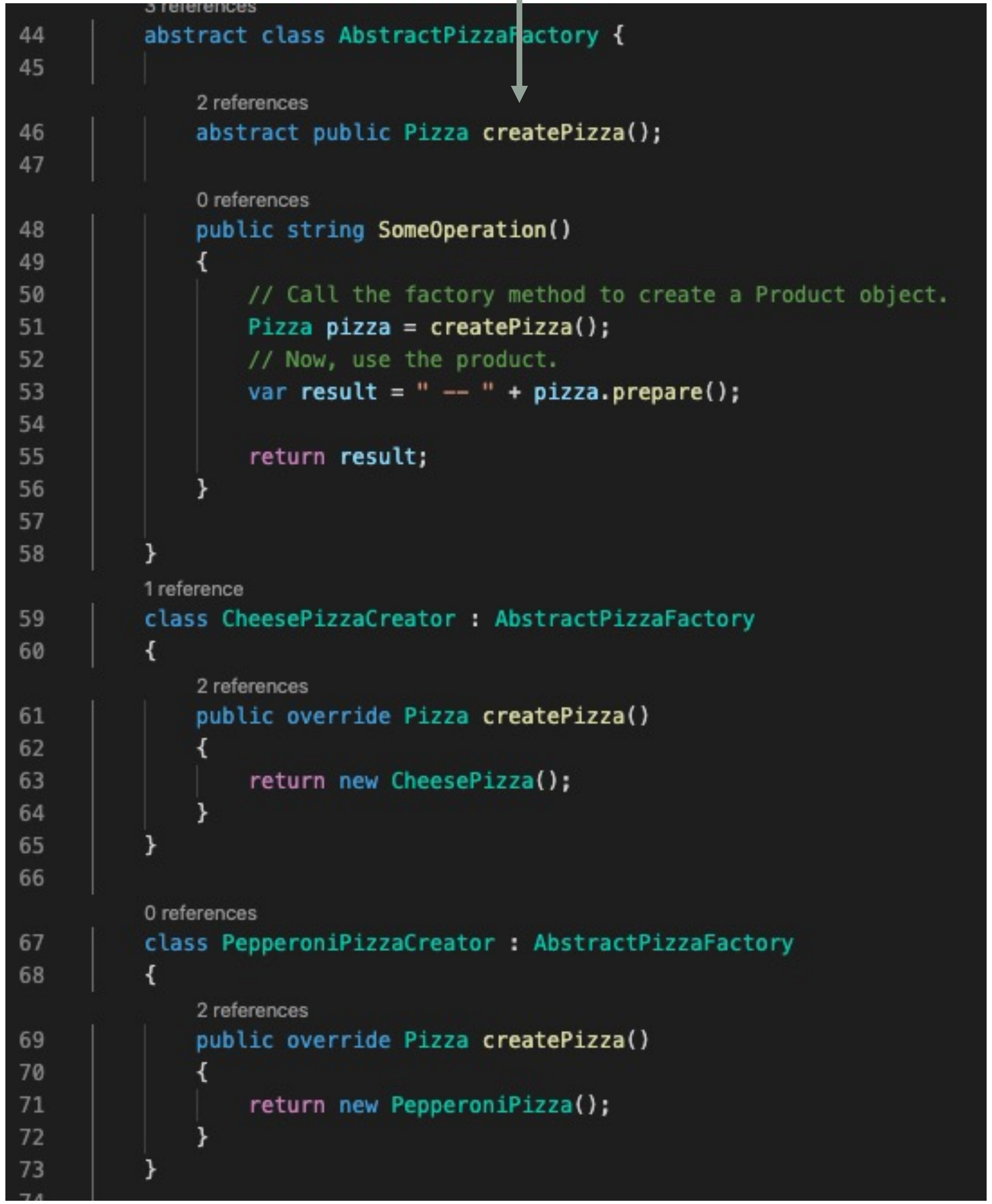
# Factory Method (Summary)

- Eliminates the need to bind creation code to specific subclasses
- Example:
  - Framework knows when to create a document, but does not know what type of document to create
- **Guidelines:**
  - No variable should hold a reference to a concrete class,
  - No class should derive from a concrete class,
  - No method should override and implemented method of its base classes
- Pattern follows the Open-Close design principle

# Factory Method (Final Comment)

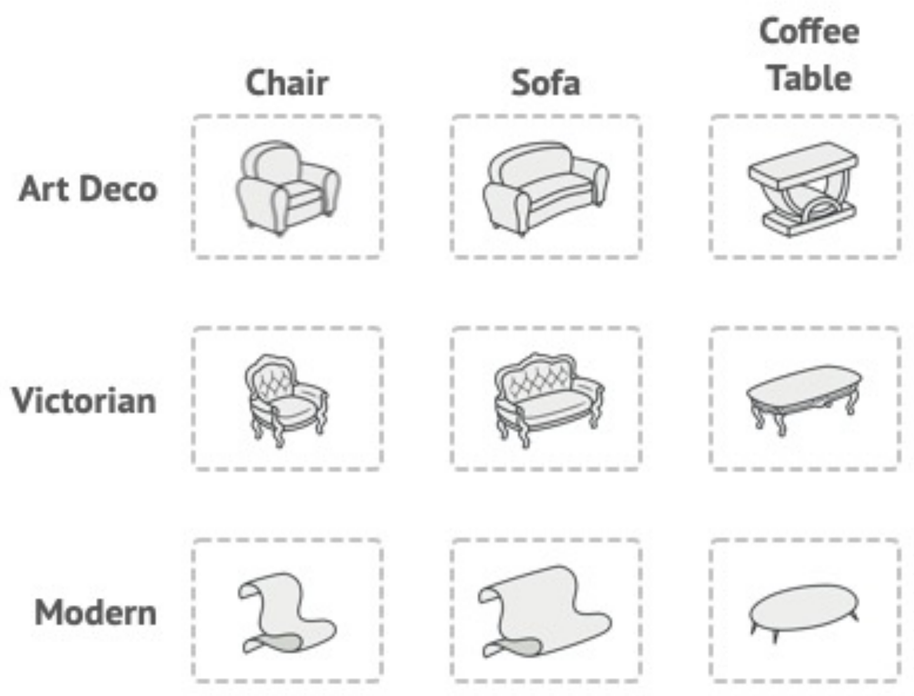
- Factory method is more than just creation of specific objects
    - Creation method is generally called: ‘**factoryMethod()**’
    - This is the only method that should be overridden by the subclasses
- Other method such as SomeOperation(), etc. are methods that operate on the product produced by the factory

factoryMethod()

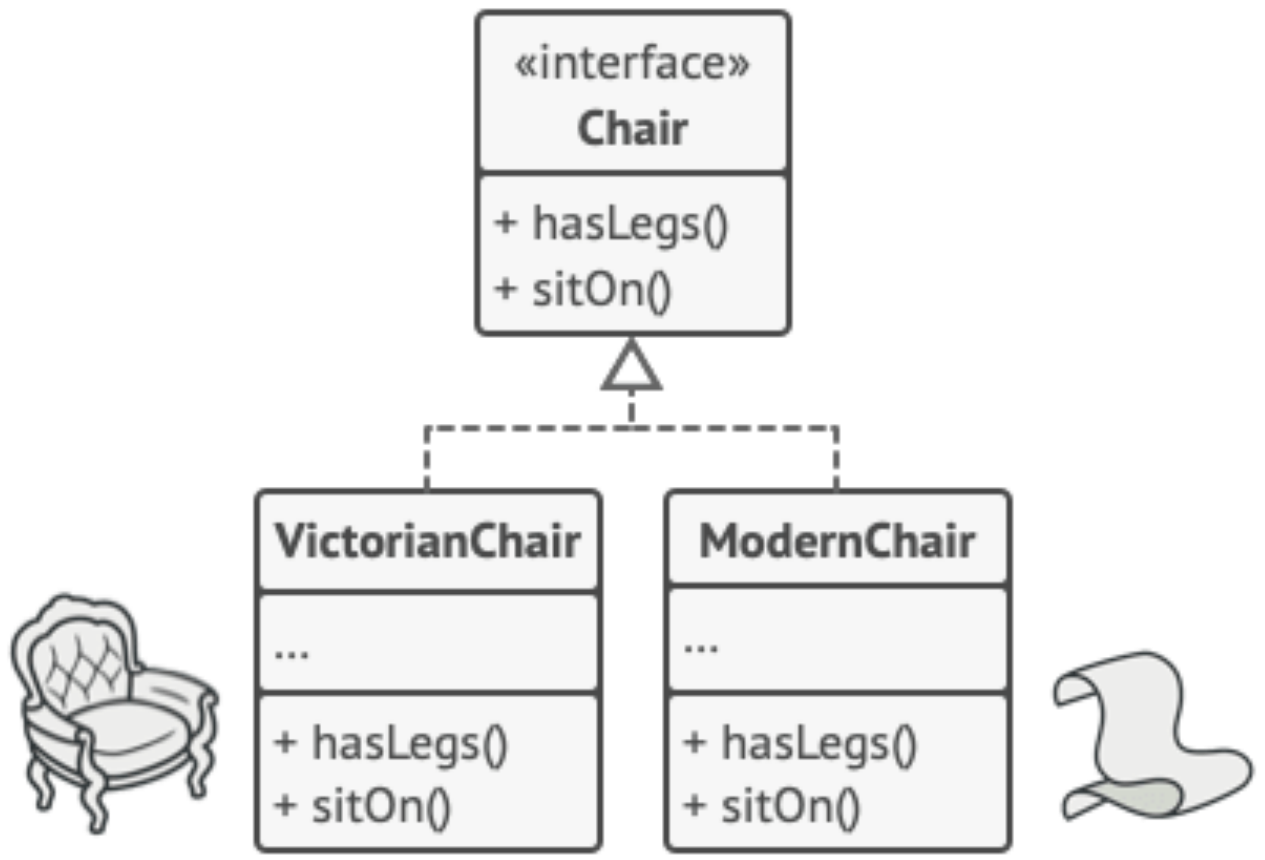


```
3 references
abstract class AbstractPizzaFactory {

    2 references
    abstract public Pizza createPizza();

    0 references
    public string SomeOperation()
    {
        // Call the factory method to create a Product object.
        Pizza pizza = createPizza();
        // Now, use the product.
        var result = " -- " + pizza.prepare();

        return result;
    }

}
1 reference
class CheesePizzaCreator : AbstractPizzaFactory
{
    2 references
    public override Pizza createPizza()
    {
        return new CheesePizza();
    }
}

0 references
class PepperoniPizzaCreator : AbstractPizzaFactory
{
    2 references
    public override Pizza createPizza()
    {
        return new PepperoniPizza();
    }
}
```

# Abstract Factory Pattern



- **Abstract Factory** lets you produce families of related or dependent objects without specifying their concrete classes
- Whenever we need to create different kind of related objects, ABF pattern should be our choice
  - Each factory will create a particular kind of related objects
  - ABF is factory of factories
- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table)
- Then you can make all variants of products follow those interfaces
- For example, all chair variants can implement the chair interface; all coffee table variants can implement the CoffeeTable interface, and so on

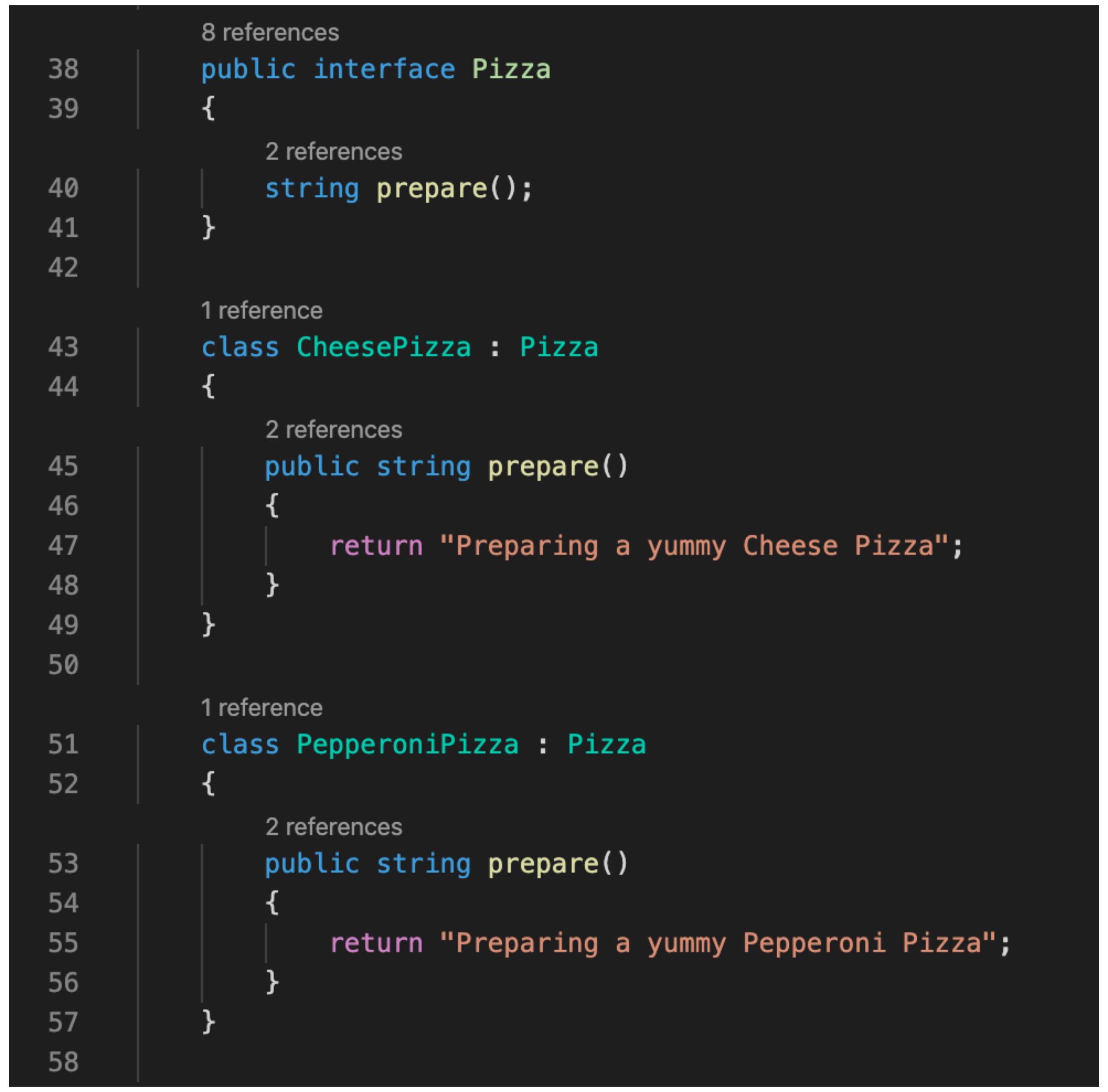


«interface»
Chair
+ hasLegs()
+ sitOn()

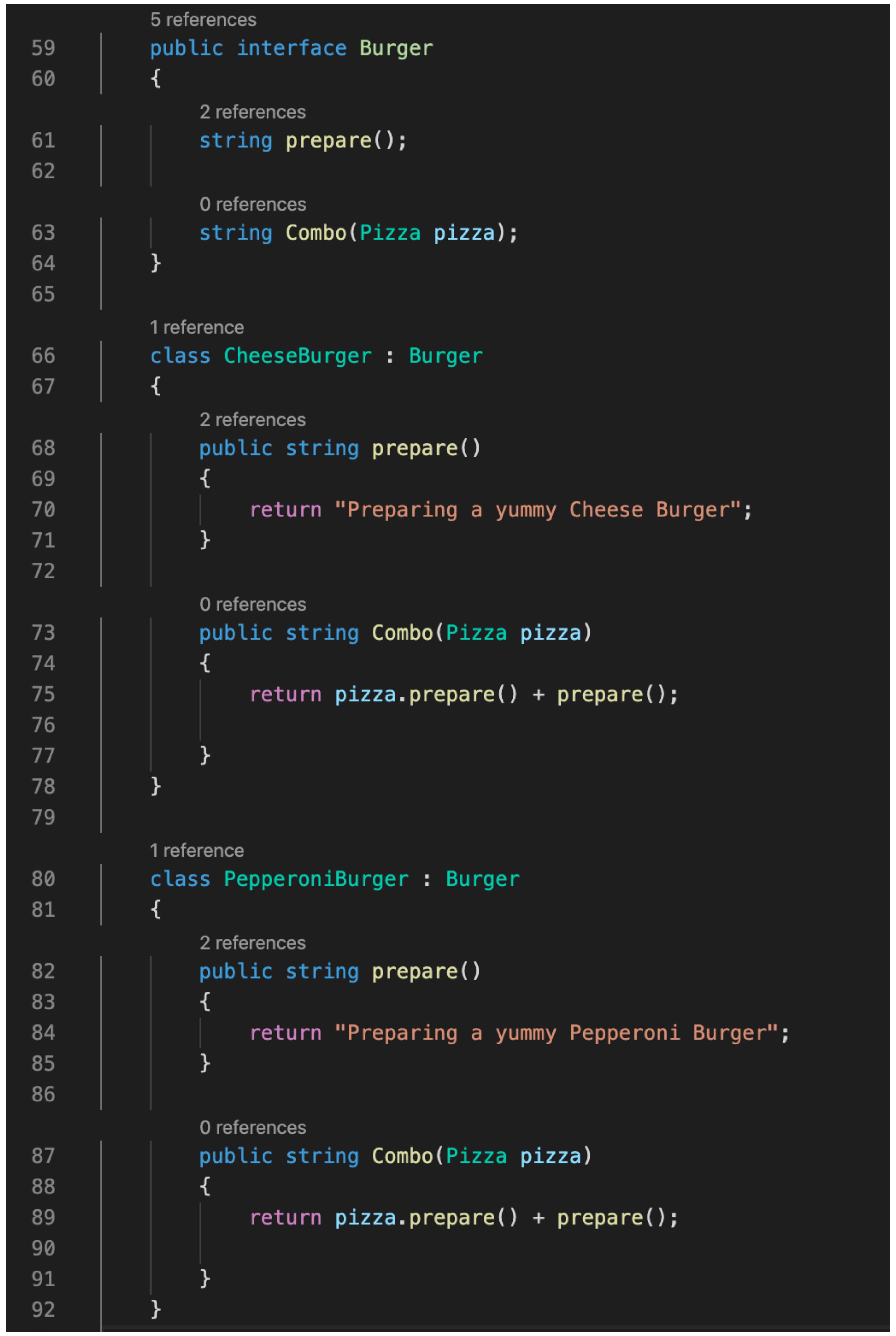
VictorianChair
...
+ hasLegs()
+ sitOn()

ModernChair
...
+ hasLegs()
+ sitOn()

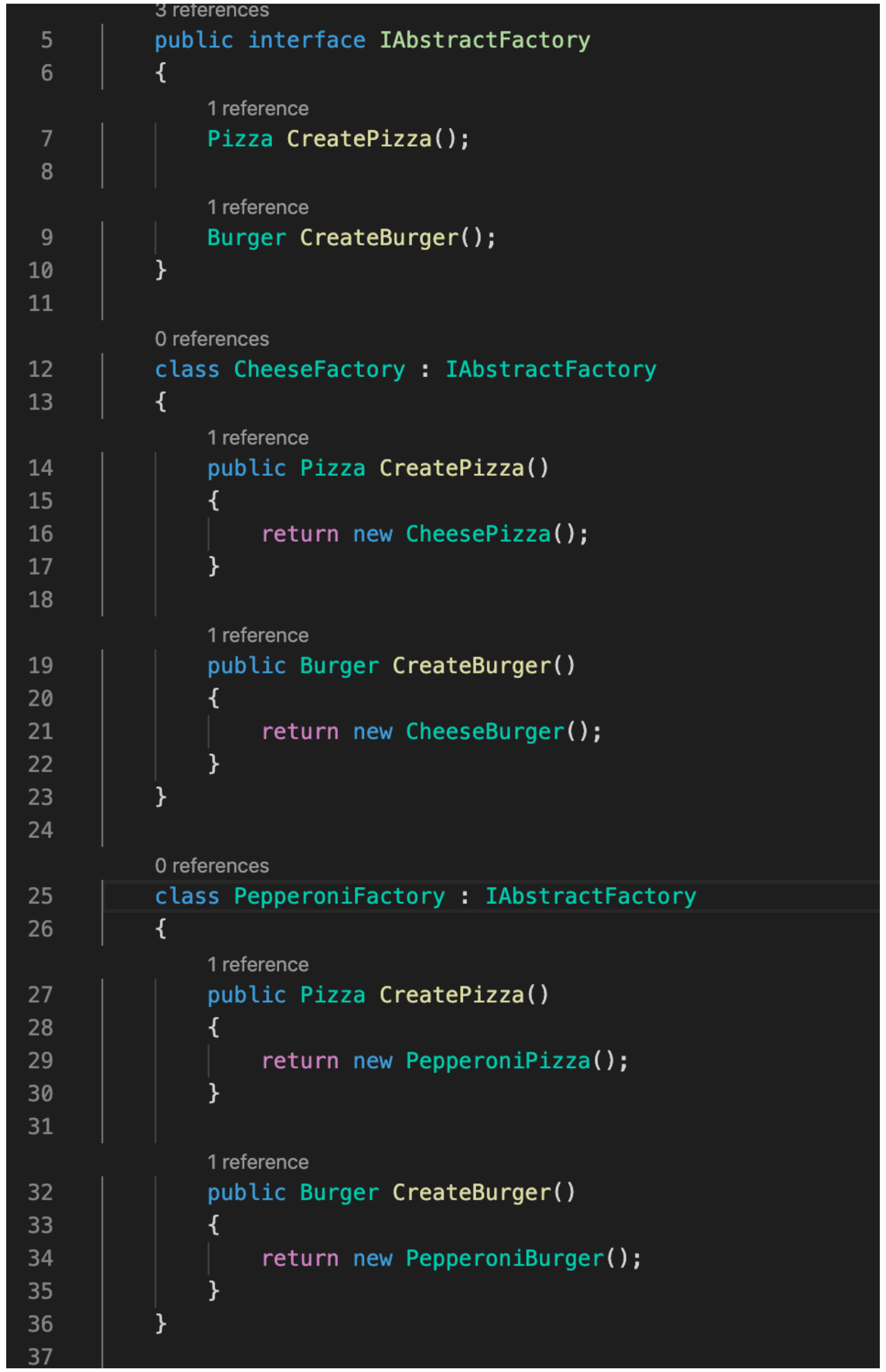
# Abstract Factory (Solution)



```
8 references
public interface Pizza
{
    2 references
    string prepare();
}

1 reference
class CheesePizza : Pizza
{
    2 references
    public string prepare()
    {
        return "Preparing a yummy Cheese Pizza";
    }
}

1 reference
class PepperoniPizza : Pizza
{
    2 references
    public string prepare()
    {
        return "Preparing a yummy Pepperoni Pizza";
    }
}

```

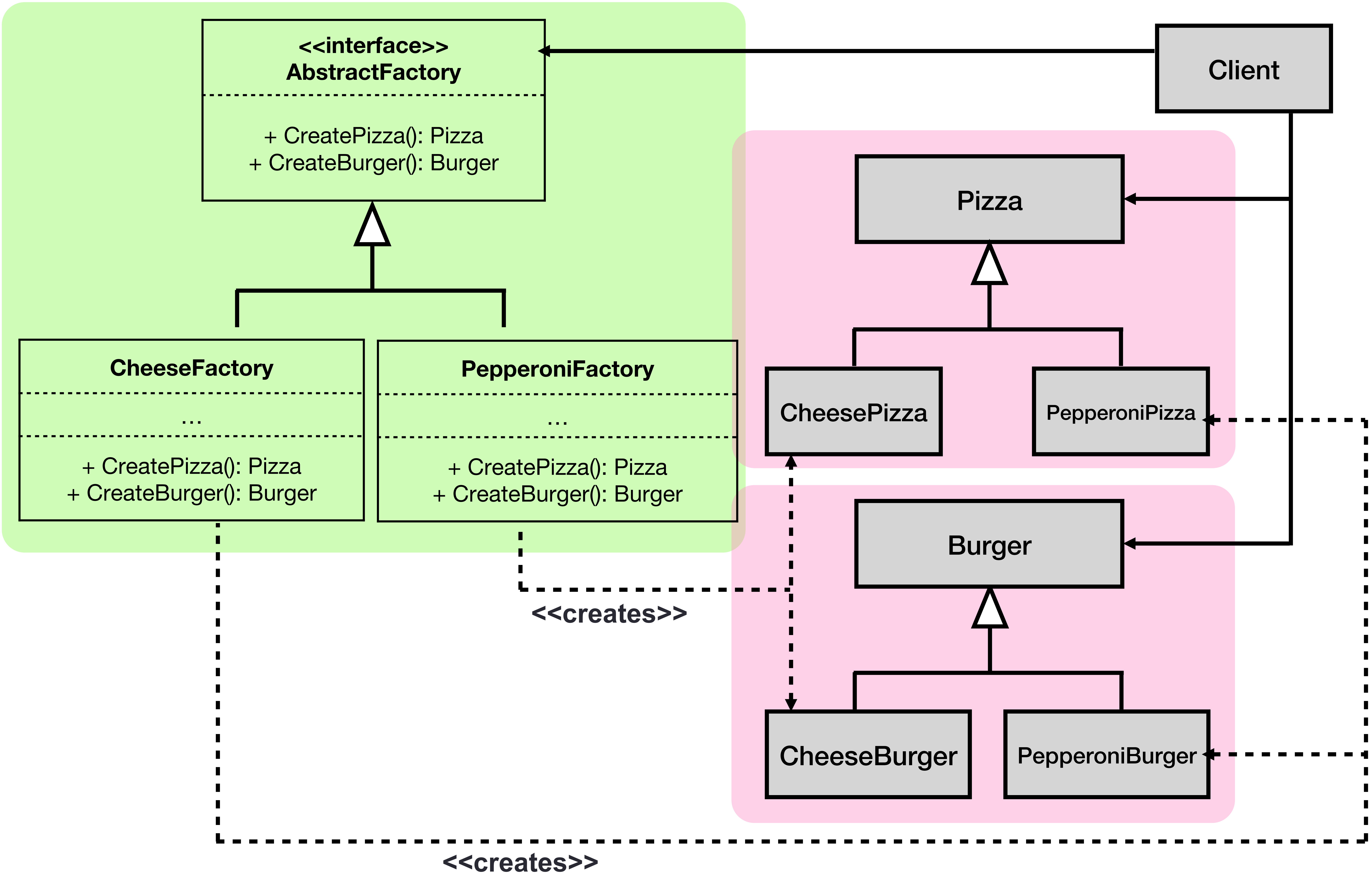


```
5 references
public interface Burger
{
    2 references
    string prepare();

    0 references
    string Combo(Pizza pizza);
}

1 reference
class CheeseBurger : Burger
{
    2 references
    public string prepare()
    {
        return "Preparing a yummy Cheese Burger";
    }

    0 references
    public string Combo(Pizza pizza)
    {
        return pizza.prepare() + prepare();

    }
}

1 reference
class PepperoniBurger : Burger
{
    2 references
    public string prepare()
    {
        return "Preparing a yummy Pepperoni Burger";
    }

    0 references
    public string Combo(Pizza pizza)
    {
        return pizza.prepare() + prepare();

    }
}
```

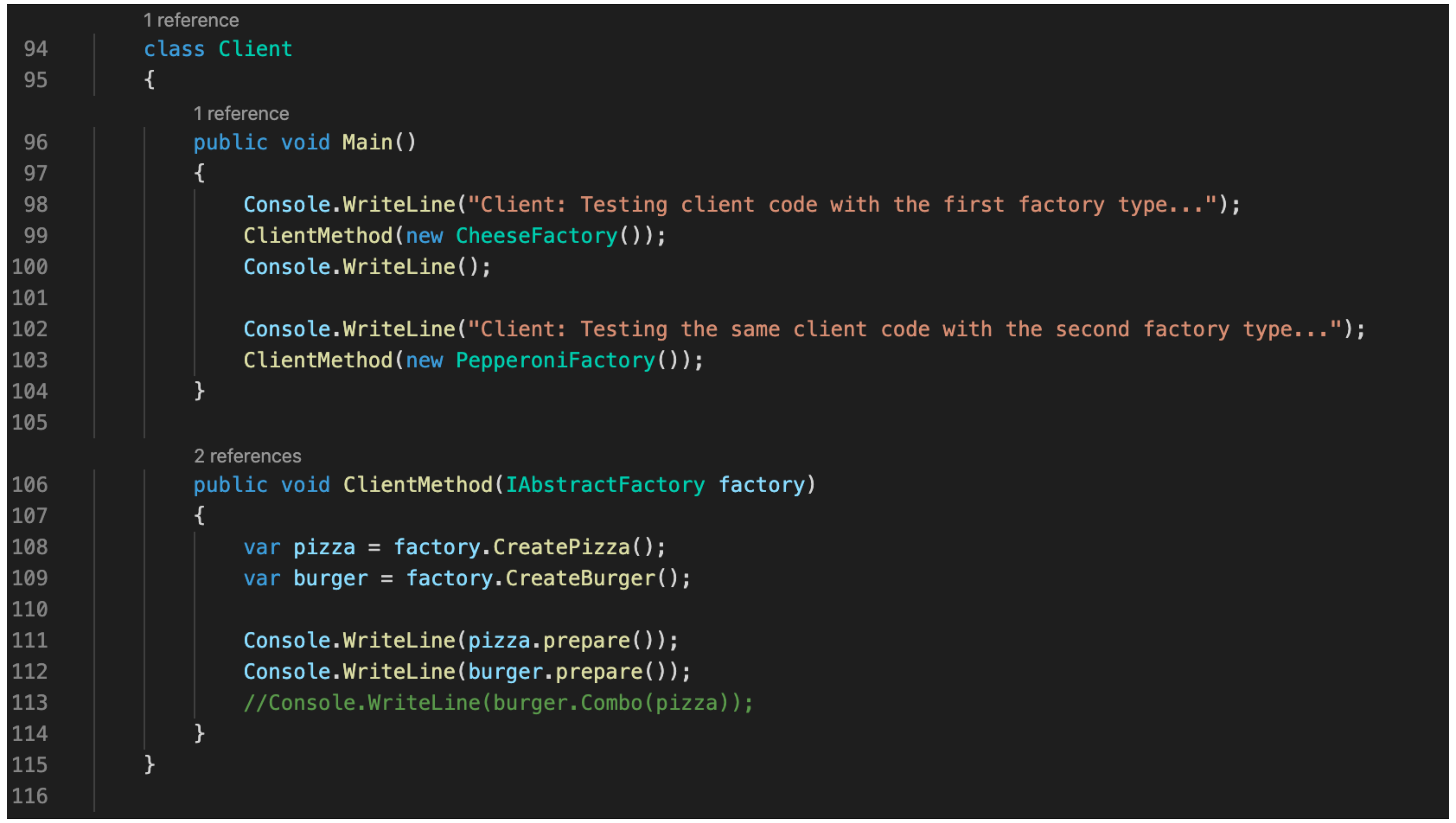


```
3 references
public interface IAbstractFactory
{
    1 reference
    Pizza CreatePizza();

    1 reference
    Burger CreateBurger();
}

0 references
class CheeseFactory : IAbstractFactory
{
    1 reference
    public Pizza CreatePizza()
    {
        return new CheesePizza();
    }

    1 reference
    public Burger CreateBurger()
    {
        return new CheeseBurger();
    }
}

0 references
class PepperoniFactory : IAbstractFactory
{
    1 reference
    public Pizza CreatePizza()
    {
        return new PepperoniPizza();
    }

    1 reference
    public Burger CreateBurger()
    {
        return new PepperoniBurger();
    }
}

```

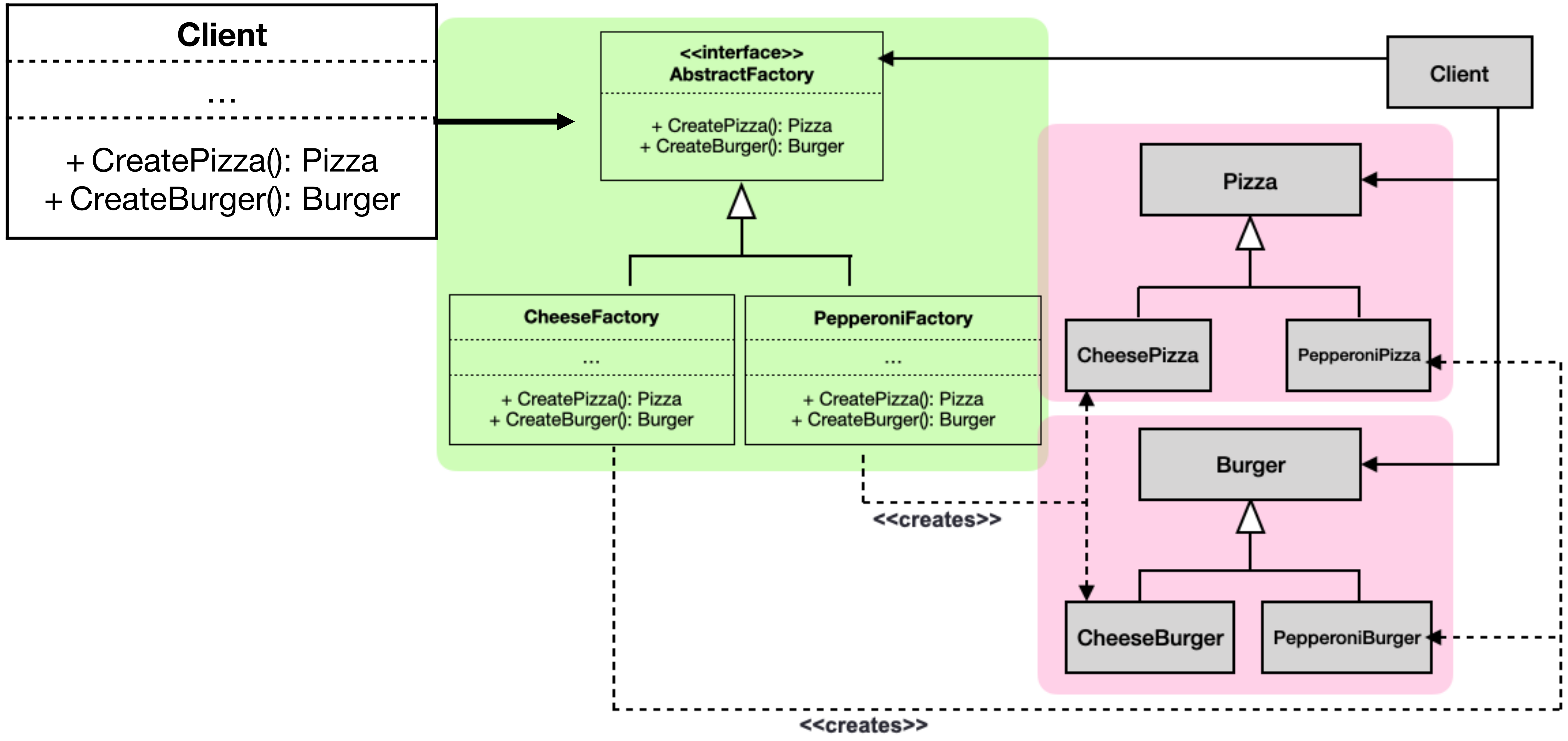
# Abstract Factory (Example)



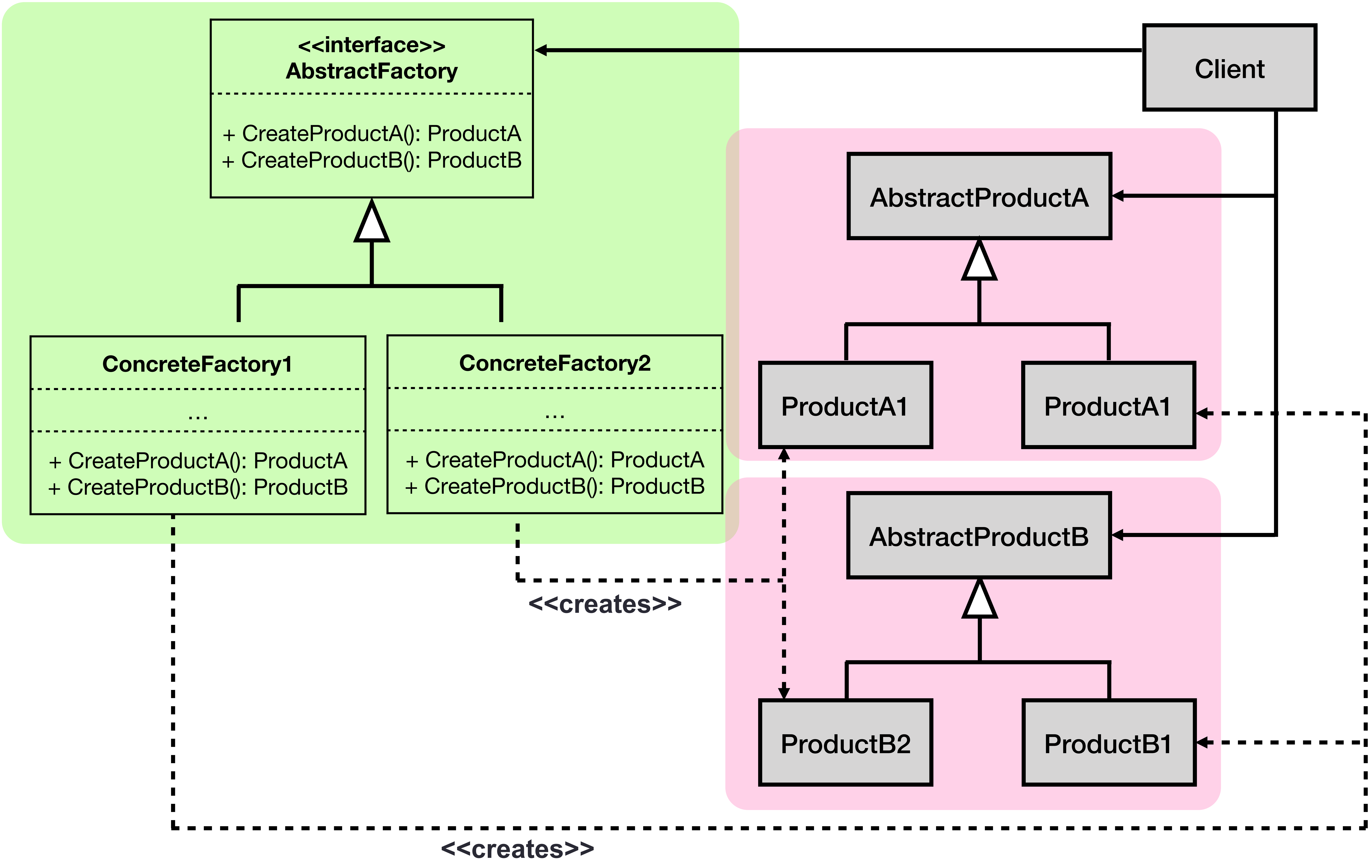
# Abstract Factory (Client)



```
1 reference
class Client
{
    1 reference
    public void Main()
    {
        Console.WriteLine("Client: Testing client code with the first factory type...");
        ClientMethod(new CheeseFactory());
        Console.WriteLine();

        Console.WriteLine("Client: Testing the same client code with the second factory type...");
        ClientMethod(new PepperoniFactory());
    }

    2 references
    public void ClientMethod(IAbstractFactory factory)
    {
        var pizza = factory.CreatePizza();
        var burger = factory.CreateBurger();

        Console.WriteLine(pizza.prepare());
        Console.WriteLine(burger.prepare());
        //Console.WriteLine(burger.Combo(pizza));
    }
}

```

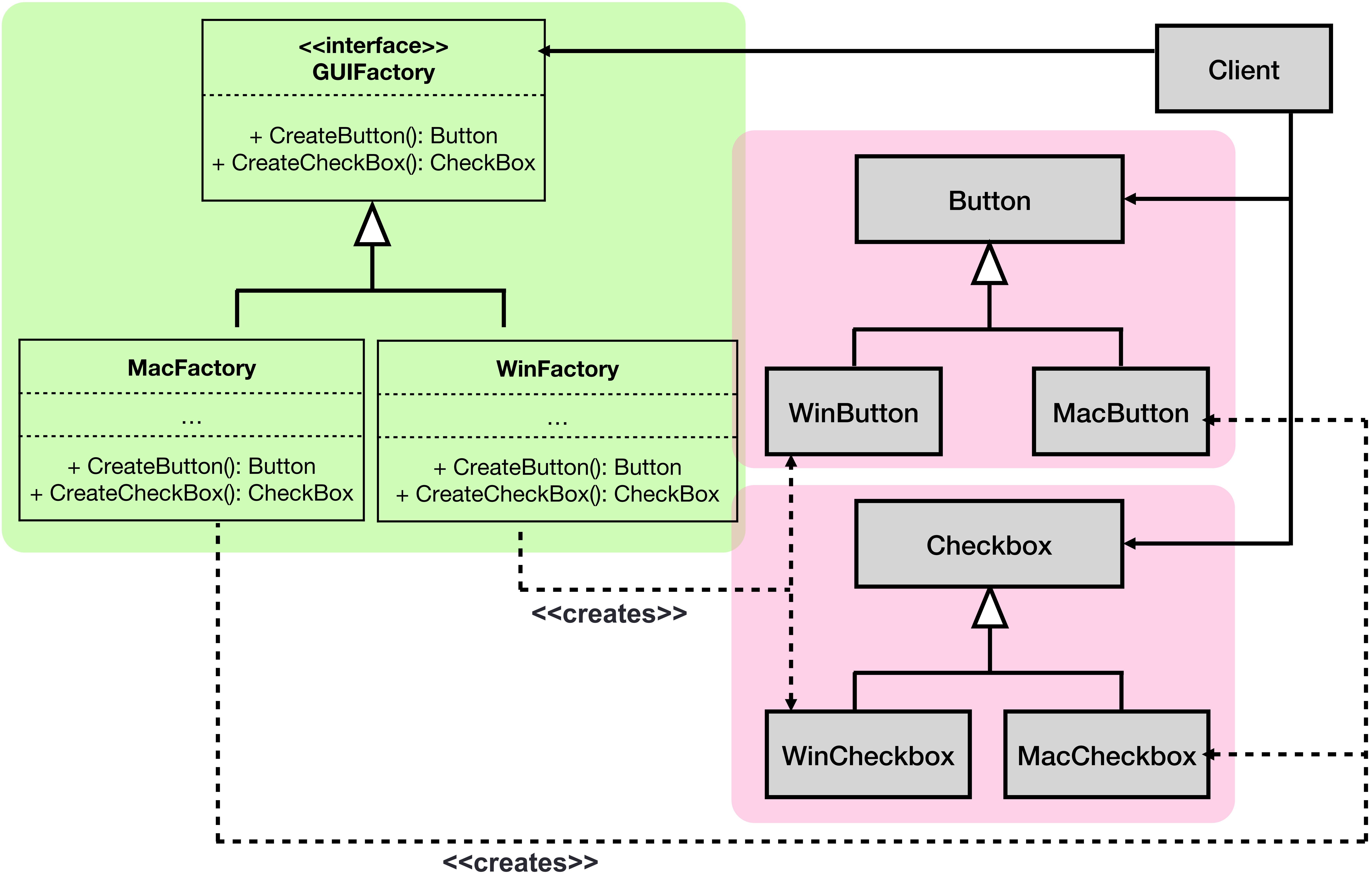


# Abstract Factory (Structure)

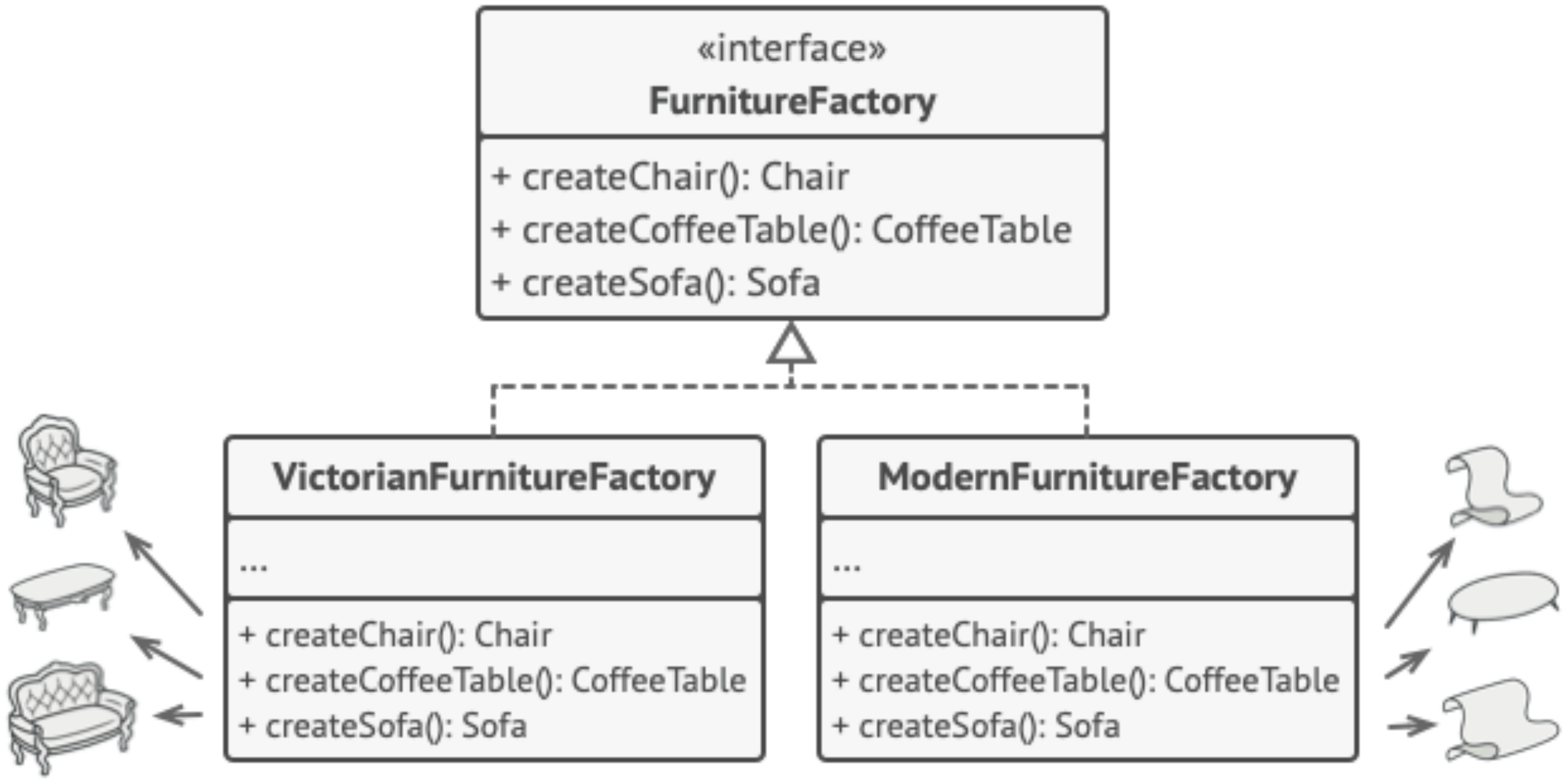


<<interface>>
AbstractFactory
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
ConcreteFactory1
...
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
ConcreteFactory2
...
+ CreateProductA(): ProductA
+ CreateProductB(): ProductB
Client
AbstractProductA
ProductA1
ProductA1
AbstractProductB
ProductB2
ProductB1
<<creates>>
<<creates>>

# Abstract Factory (Example)



# Abstract Factory (Solution)



«interface»
**FurnitureFactory**

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa

**VictorianFurnitureFactory**

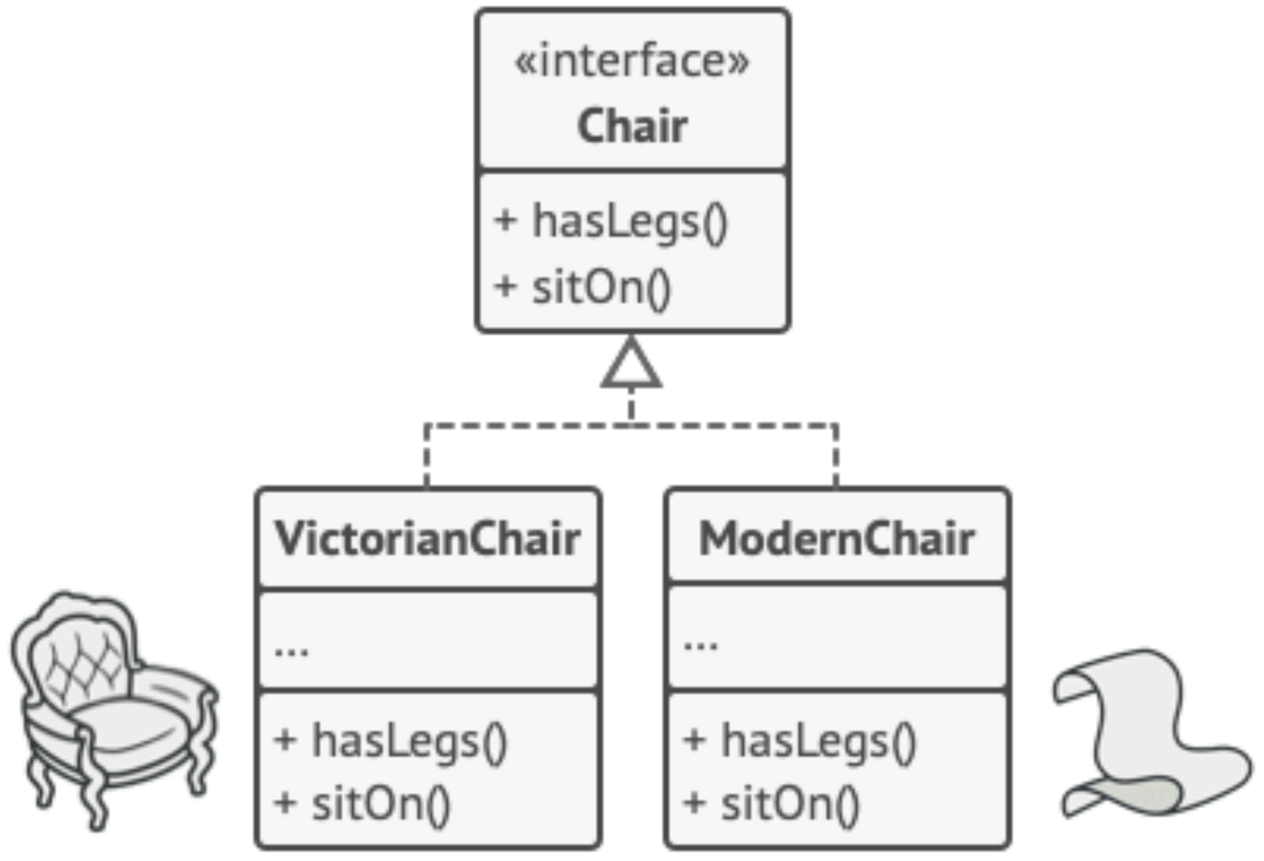
...

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa

**ModernFurnitureFactory**

...

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa



«interface»
**Chair**

+ hasLegs()
+ sitOn()

**VictorianChair**

...

+ hasLegs()
+ sitOn()

**ModernChair**

...

+ hasLegs()
+ sitOn()

# Consequences

- **It isolates concrete classes**
  - isolates clients from implementation classes
  - clients manipulate instances through their abstract interface
  - product class names are isolated in the implementation of the concrete factory; they do not appear in client code
- **It makes exchanging product families easy**
  - The class of a concrete factory appears only once in the application, i.e. where it's instantiated
  - Use different product configurations simply by changing the concrete factory

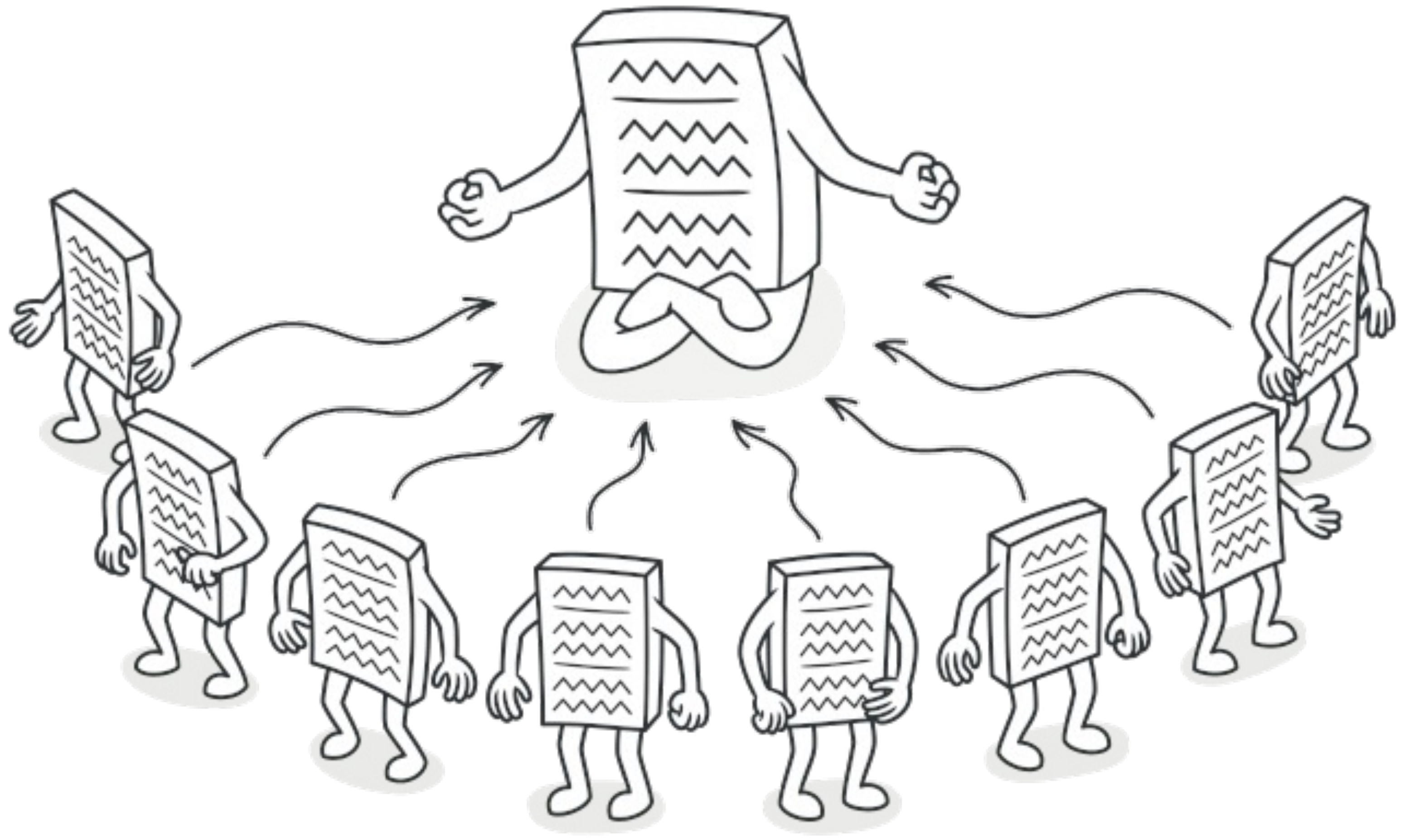
# Consequences

- **It promotes consistency among products**
  - When product objects in a family are designed to work together, it’s important to use only one family at a time
  - Abstract factory makes this constraint easy to implement
- **Supporting new kinds of products is difficult**
  - Abstract Factory fixes the set of products which can be created
  - To extend the products, means that the Abstract Factory interface must be changed and all the Concrete Factory subclasses must be changed as well

# Singleton

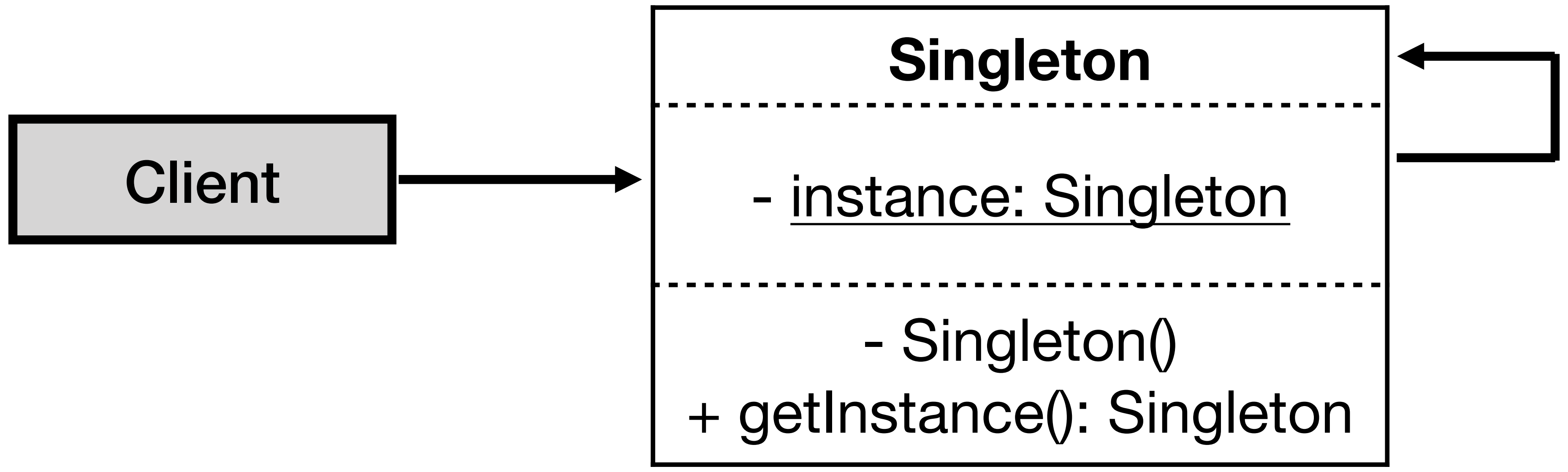
# Singleton

- Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance
  - It’s trivial to initiate an object of a class — but how do we ensure that only one object ever gets created?
  - Why Singleton?
    - Plays the role of global variables (issue was that any body can overwrite them)
    - Singleton lets you access some object from any where in the program like any global variable, also it protects the object from being overwritten by other code



# Singleton (Solution and Structure)

- (1) Make the default constructor private, to prevent other objects from using the new operator with the Singleton class
- (2) Create a **static** creation method that acts as a constructor
  - This method calls the private constructor to create an object and saves it in a **static** field.
  - All following calls to this method return the cached object.



```
if (instance == null) {
  Instance = new Singleton()
}
return instance
```

# Singleton (Use Cases)

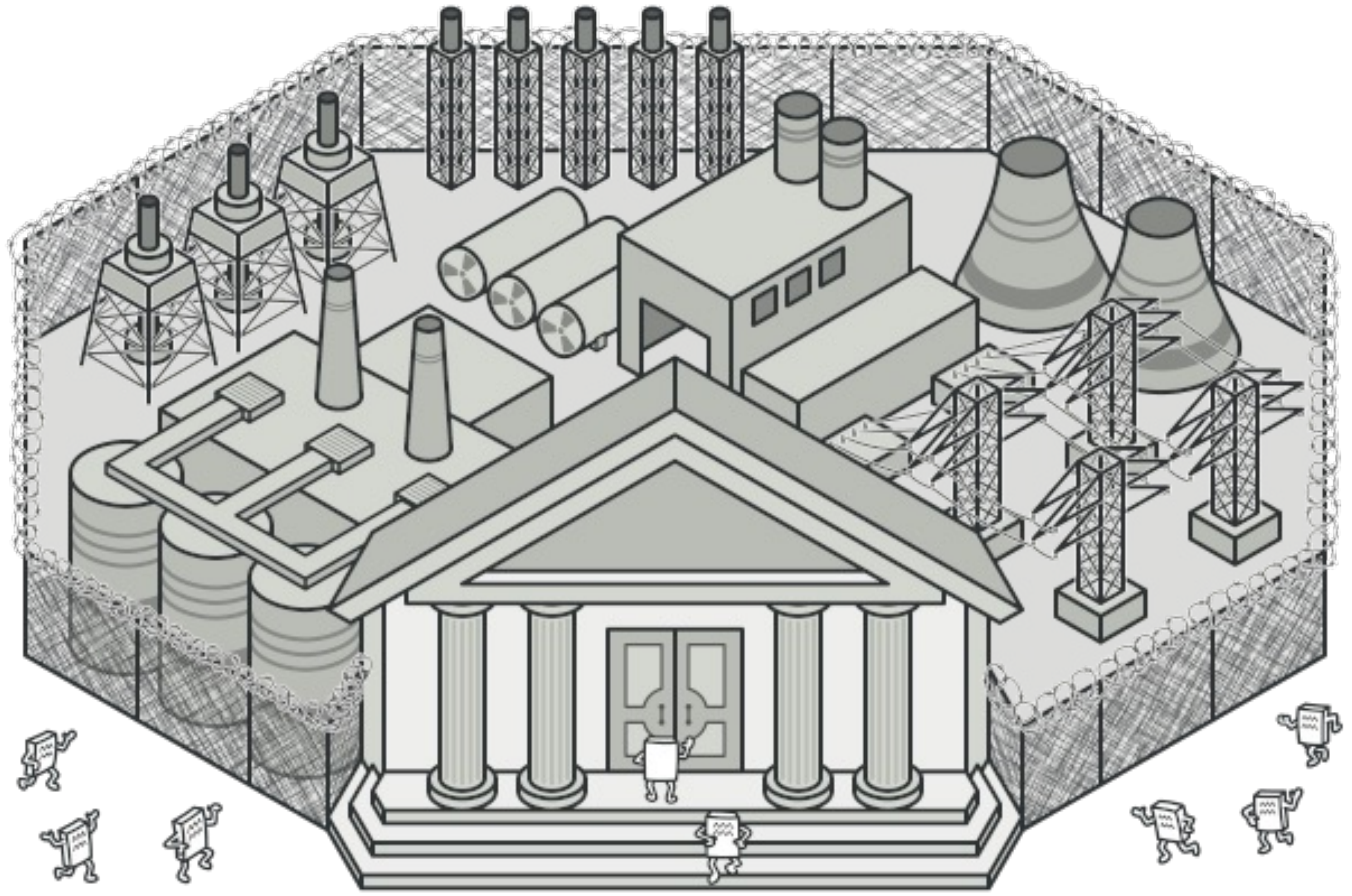
- Use Singleton pattern, when a class in your program should have just a single instance available to all clients
  - There are several examples of where only a single instance of a class should exist, including caches, thread pools, and registries
  - A single database object shared by different parts of the program
- Use Singleton pattern when you need stricter control over global variables
  - Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code

- Singleton pattern has several advantages over just use of static classes, e.g., singleton can be passed as a reference in methods, it can implement certain interfaces, it can inherit from classes etc.

# Facade

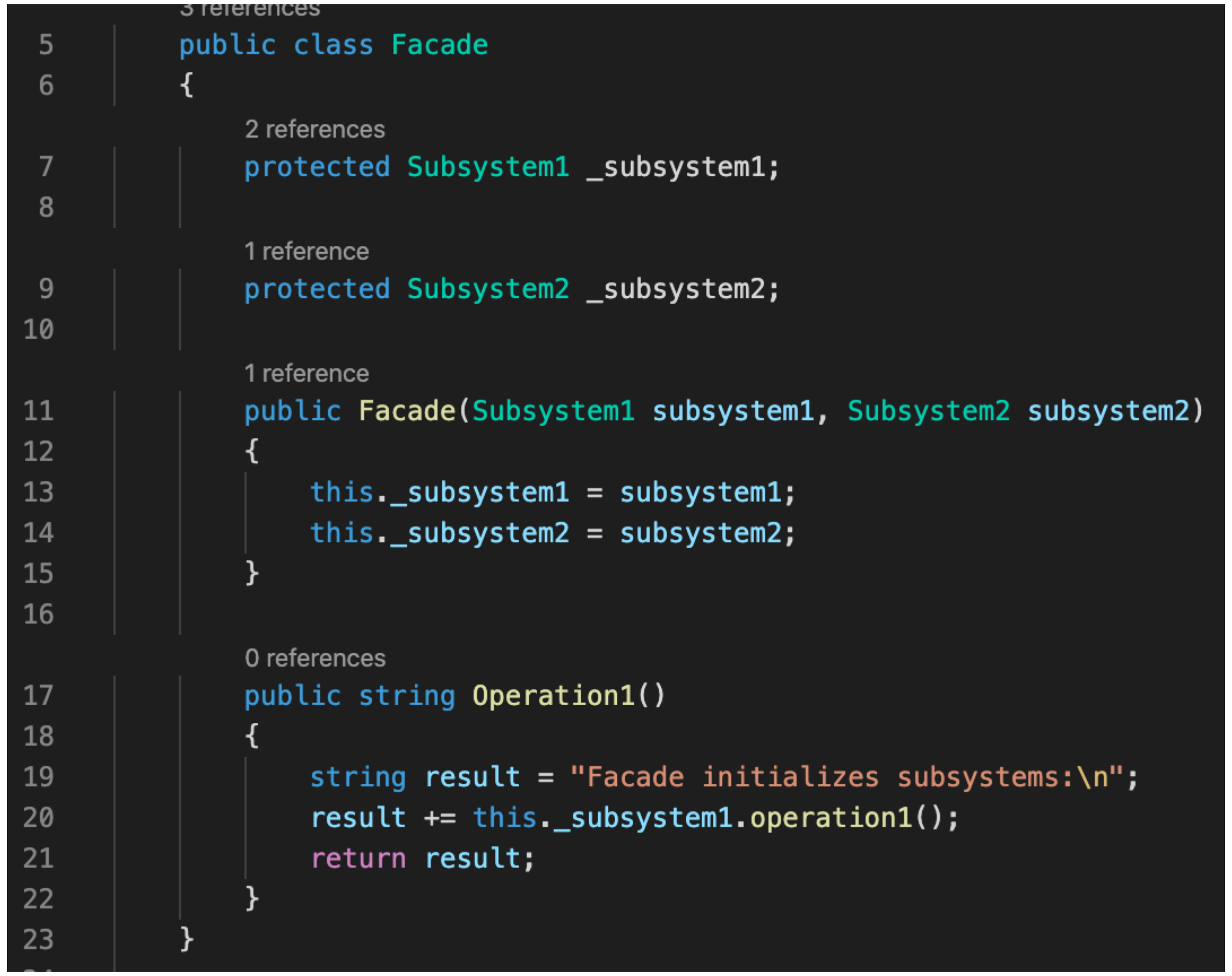
# Facade

- **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes

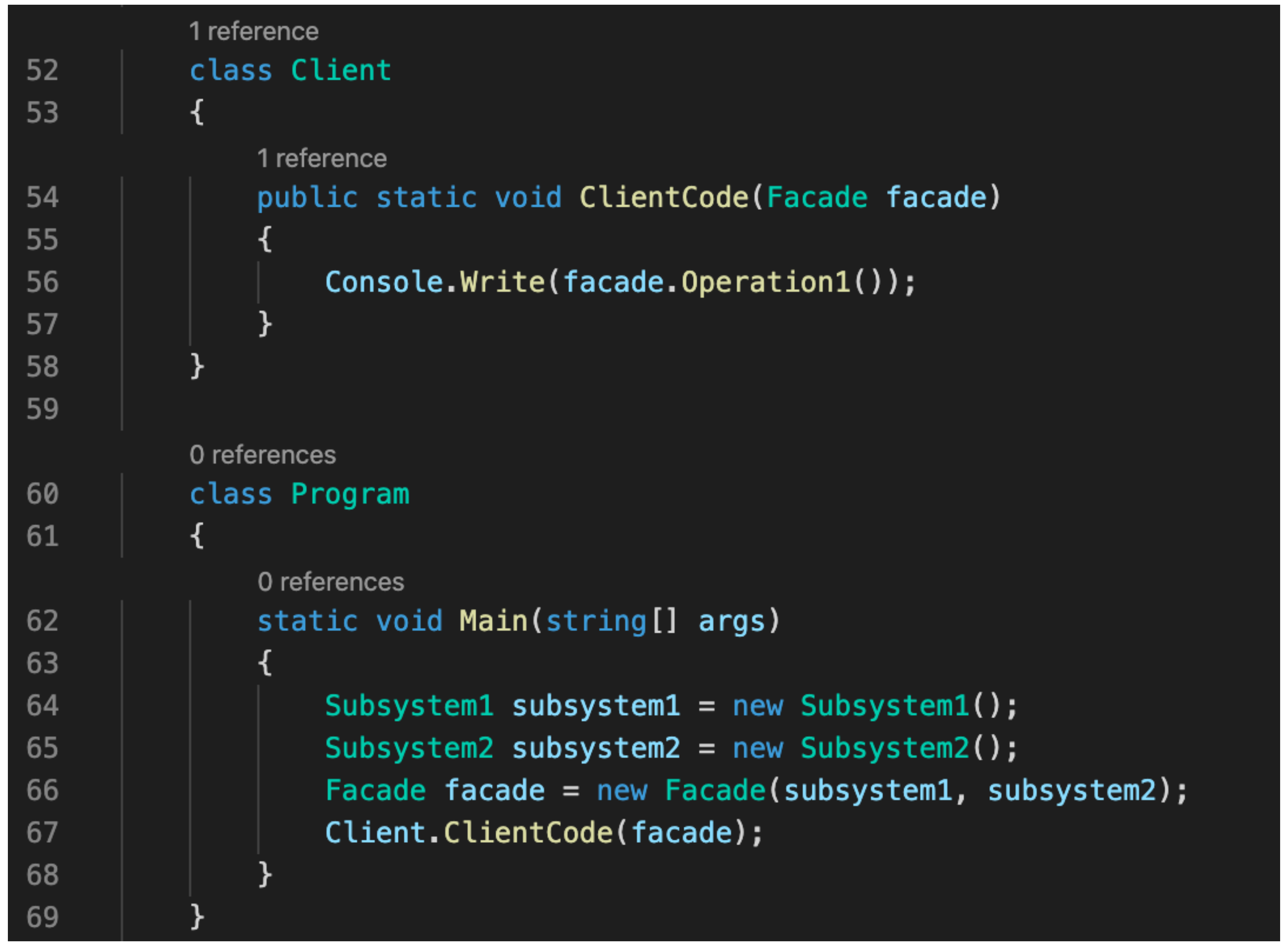


- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework
  - Ordinarily, you’d need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on
  - As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain

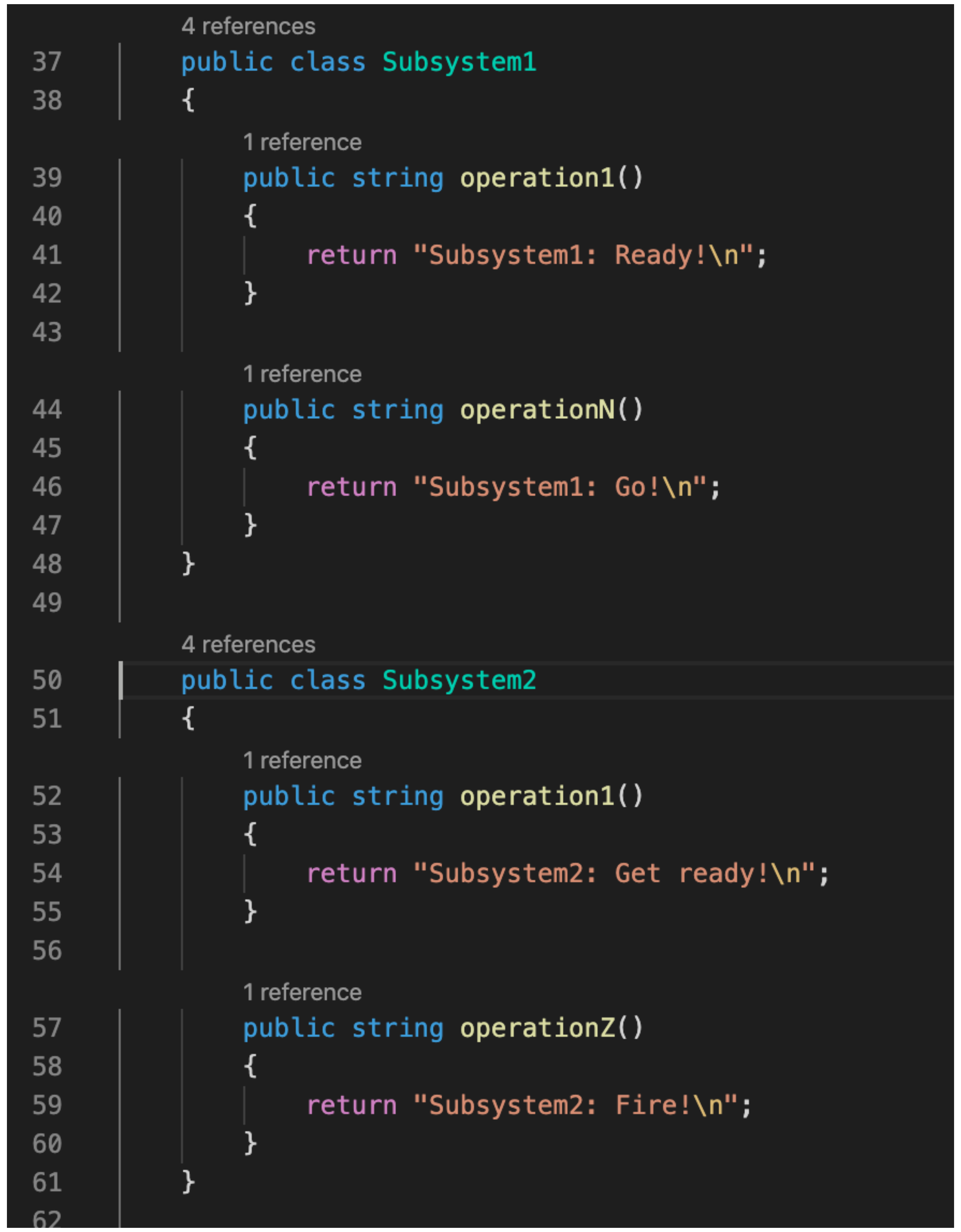
# Facade



```
3 references
public class Facade
{
    2 references
    protected Subsystem1 _subsystem1;

    1 reference
    protected Subsystem2 _subsystem2;

    1 reference
    public Facade(Subsystem1 subsystem1, Subsystem2 subsystem2)
    {
        this._subsystem1 = subsystem1;
        this._subsystem2 = subsystem2;
    }

    0 references
    public string Operation1()
    {
        string result = "Facade initializes subsystems:\n";
        result += this._subsystem1.operation1();
        return result;
    }
}
```



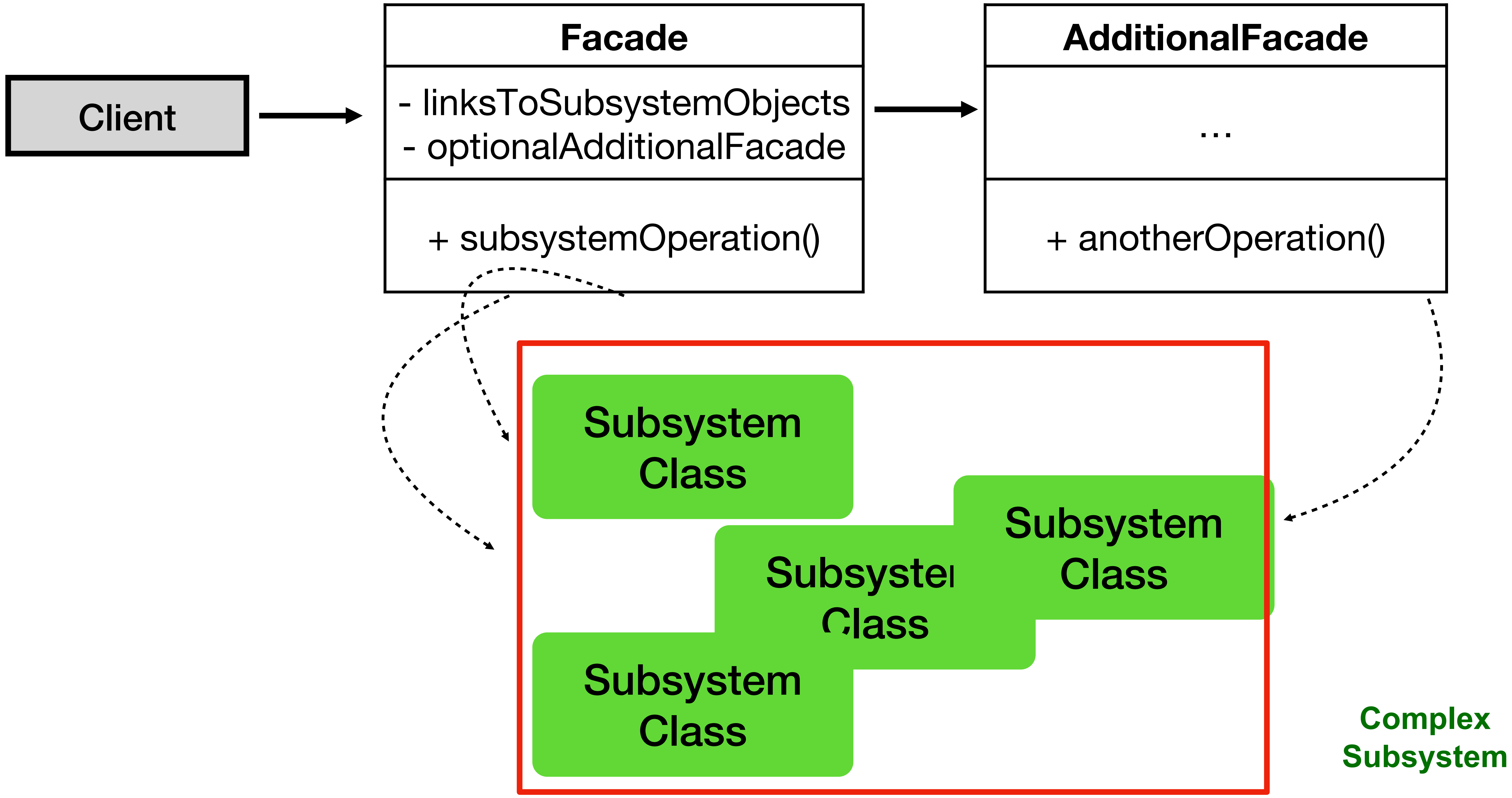
```
1 reference
class Client
{
    1 reference
    public static void ClientCode(Facade facade)
    {
        Console.Write(facade.Operation1());
    }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Subsystem1 subsystem1 = new Subsystem1();
        Subsystem2 subsystem2 = new Subsystem2();
        Facade facade = new Facade(subsystem1, subsystem2);
        Client.ClientCode(facade);
    }
}
```



```
4 references
public class Subsystem1
{
    1 reference
    public string operation1()
    {
        return "Subsystem1: Ready!\n";
    }

    1 reference
    public string operationN()
    {
        return "Subsystem1: Go!\n";
    }
}

4 references
public class Subsystem2
{
    1 reference
    public string operation1()
    {
        return "Subsystem2: Get ready!\n";
    }

    1 reference
    public string operationZ()
    {
        return "Subsystem2: Fire!\n";
    }
}
```

# Facade (Solution & Structure)

- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts
- A facade might provide limited functionality in comparison to working with the subsystem directly
- However, it includes only those features that clients really care about



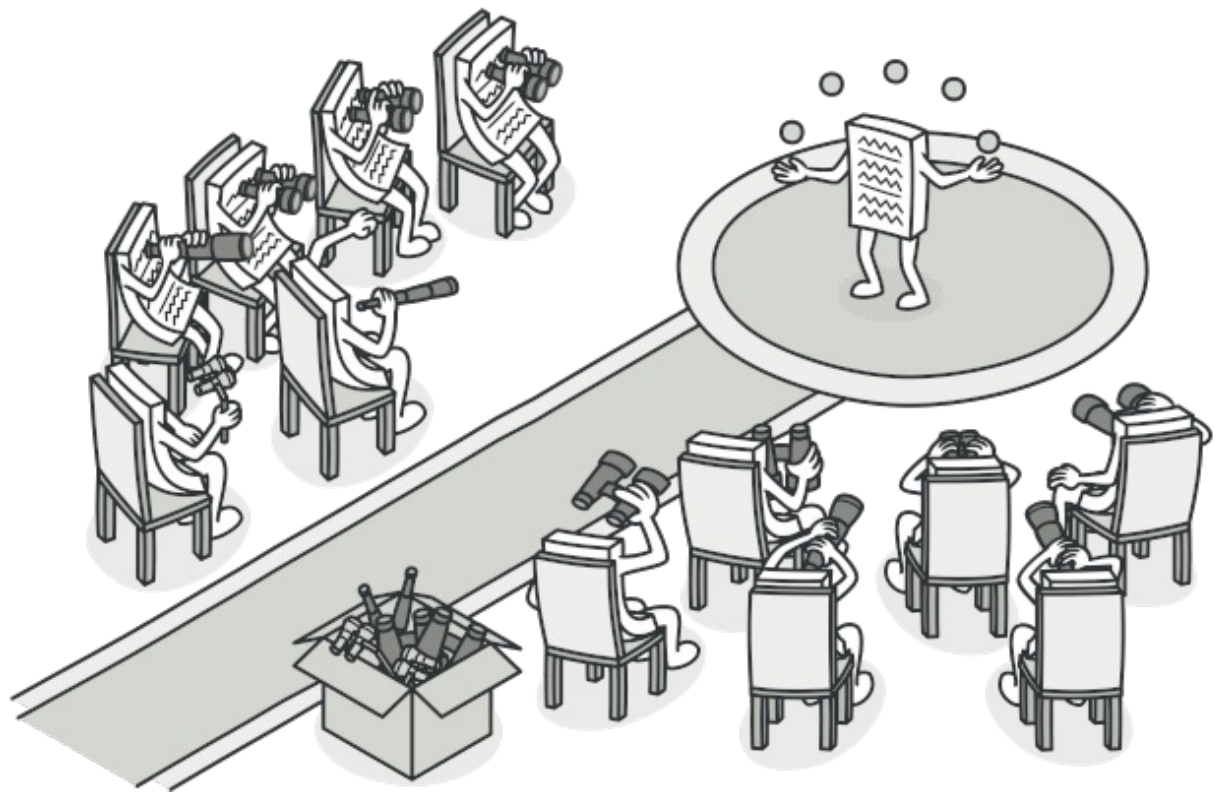
# Facade (Use Cases)

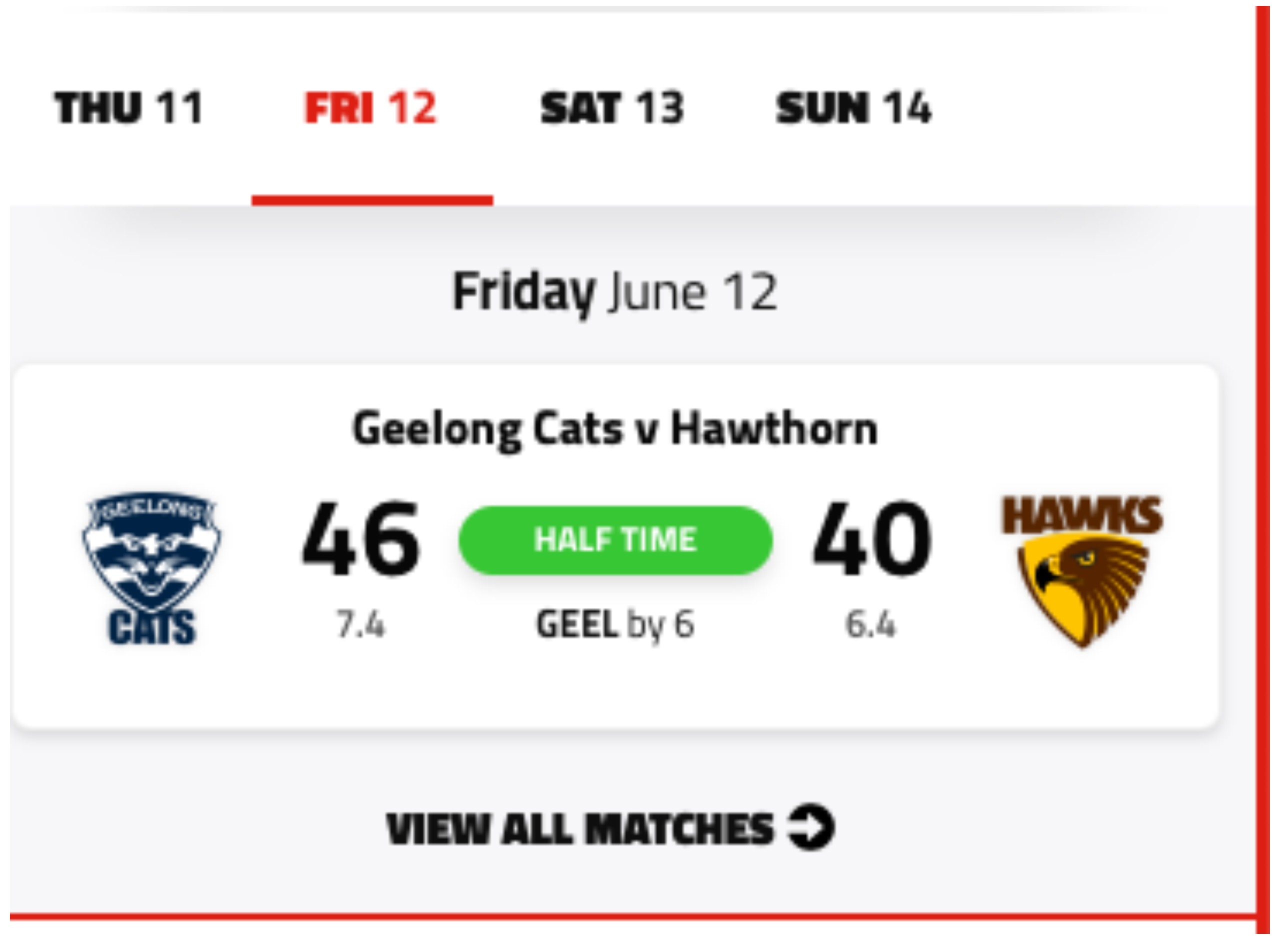
- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem
- Use the Facade when you want to structure a subsystem into layers

# Observer

# Observer

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





THU 11 FRI 12 SAT 13 SUN 14

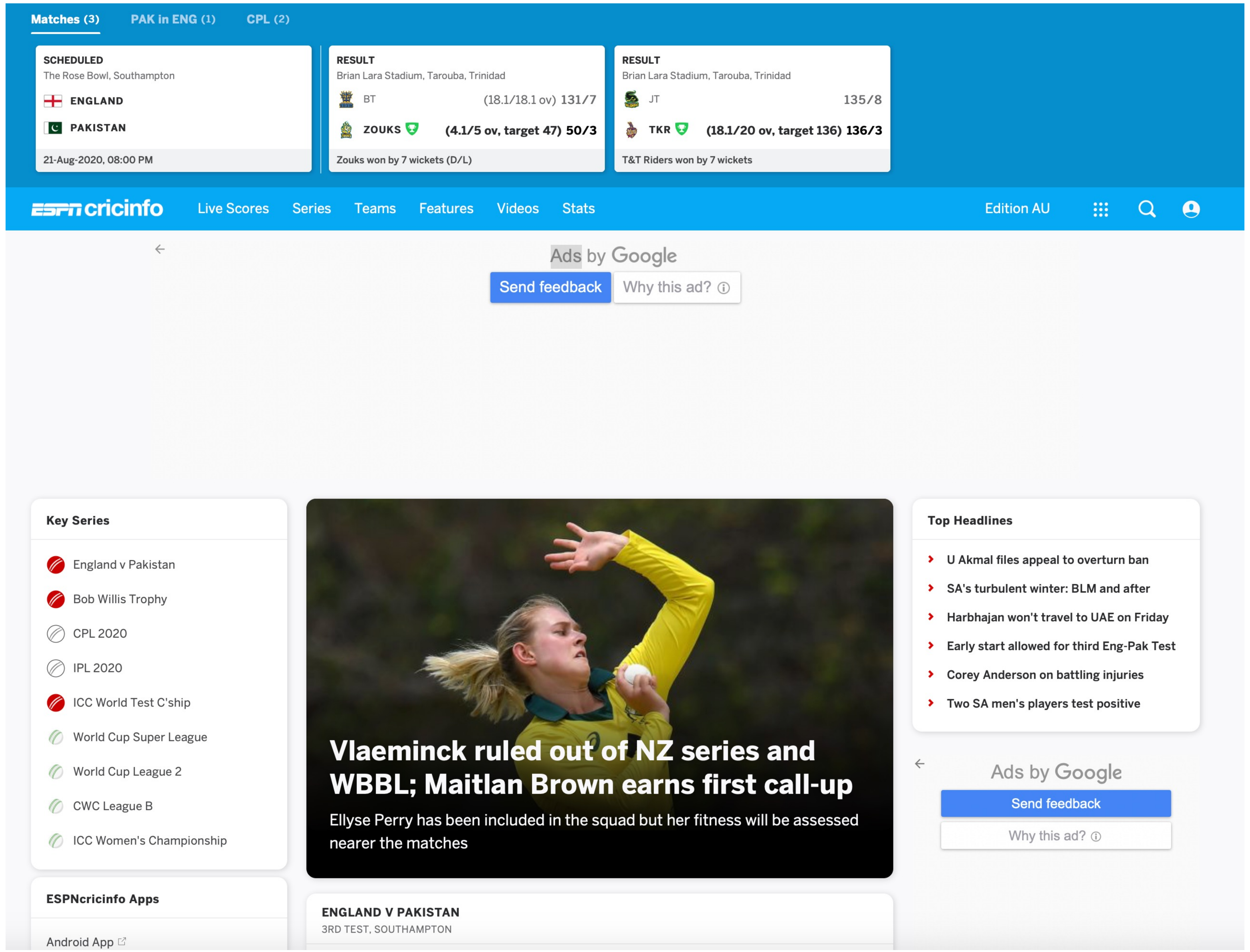
Friday June 12

Geelong Cats v Hawthorn

46 HALF TIME 40

7.4 GEEL by 6 6.4

VIEW ALL MATCHES



Matches (3) PAK in ENG (1) CPL (2)

SCHEDULED
The Rose Bowl, Southampton
ENGLAND
PAKISTAN
21-Aug-2020, 08:00 PM

RESULT
Brian Lara Stadium, Tarouba, Trinidad
BT (18.1/18.1 ov) 131/7
ZOUKS (4.1/5 ov, target 47) 50/3
Zouks won by 7 wickets (D/L)

RESULT
Brian Lara Stadium, Tarouba, Trinidad
JT 135/8
TKR (18.1/20 ov, target 136) 136/3
T&T Riders won by 7 wickets

espn cricinfo Live Scores Series Teams Features Videos Stats Edition AU

Ads by Google
Send feedback Why this ad?

Key Series
- England v Pakistan
- Bob Willis Trophy
- CPL 2020
- IPL 2020
- ICC World Test C'ship
- World Cup Super League
- World Cup League 2
- CWC League B
- ICC Women's Championship

Vlaeminck ruled out of NZ series and WBBL; Maitlan Brown earns first call-up

Ellyse Perry has been included in the squad but her fitness will be assessed nearer the matches

Top Headlines
- U Akmal files appeal to overturn ban
- SA's turbulent winter: BLM and after
- Harbhajan won't travel to UAE on Friday
- Early start allowed for third Eng-Pak Test
- Corey Anderson on battling injuries
- Two SA men's players test positive

Ads by Google
Send feedback
Why this ad?

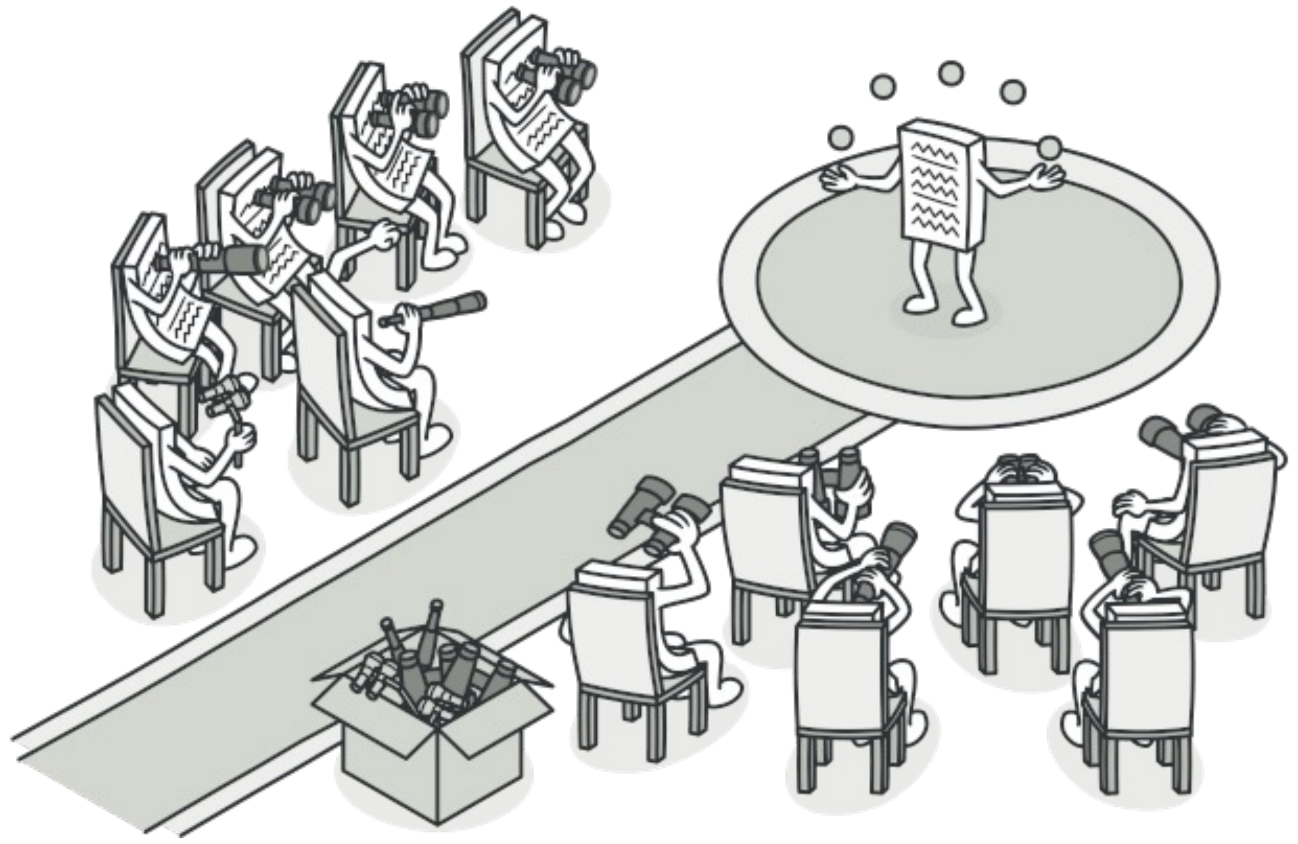
ESPNcricinfo Apps
Android App

ENGLAND V PAKISTAN
3RD TEST, SOUTHAMPTON

# Use-cases

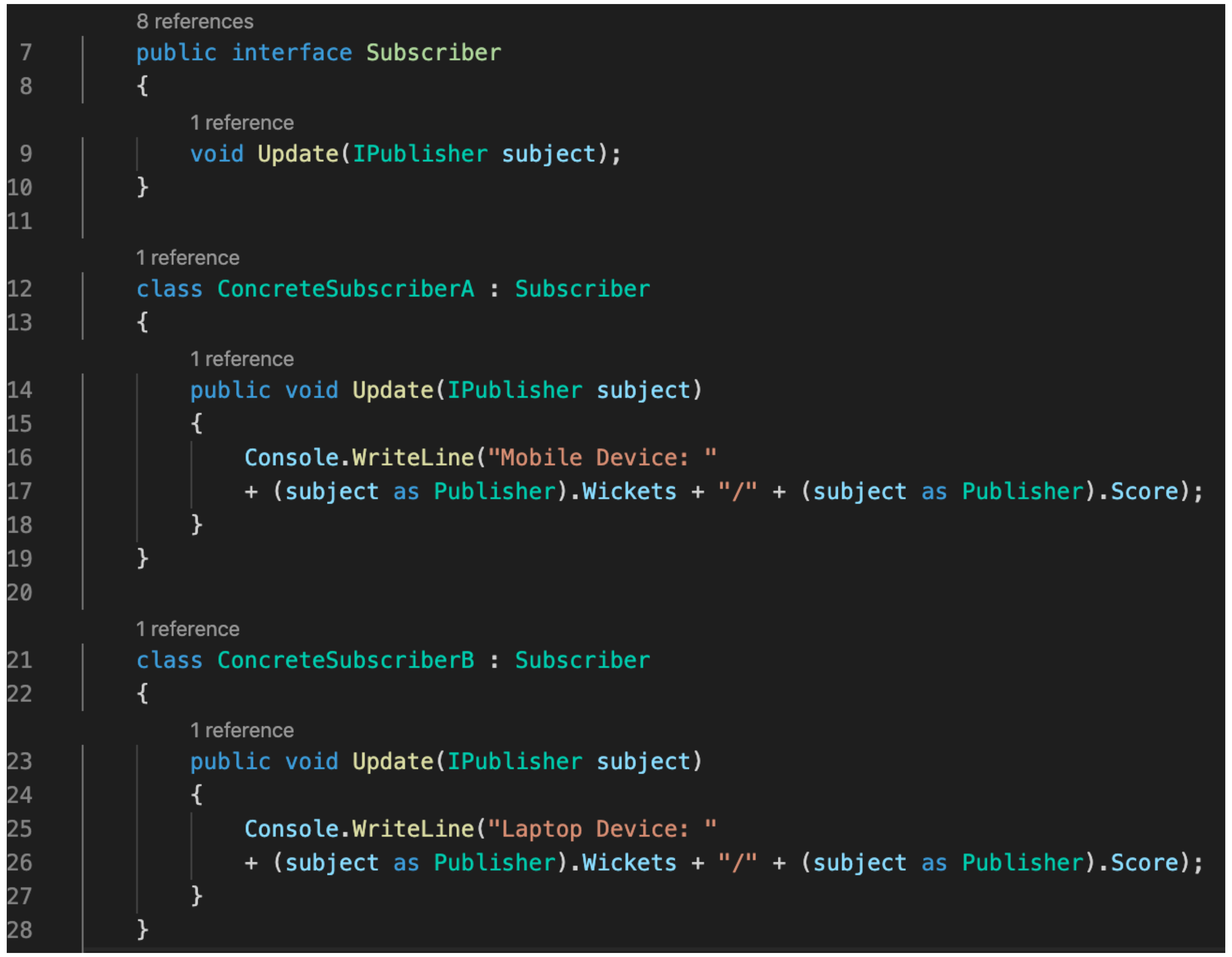
- Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon

- 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

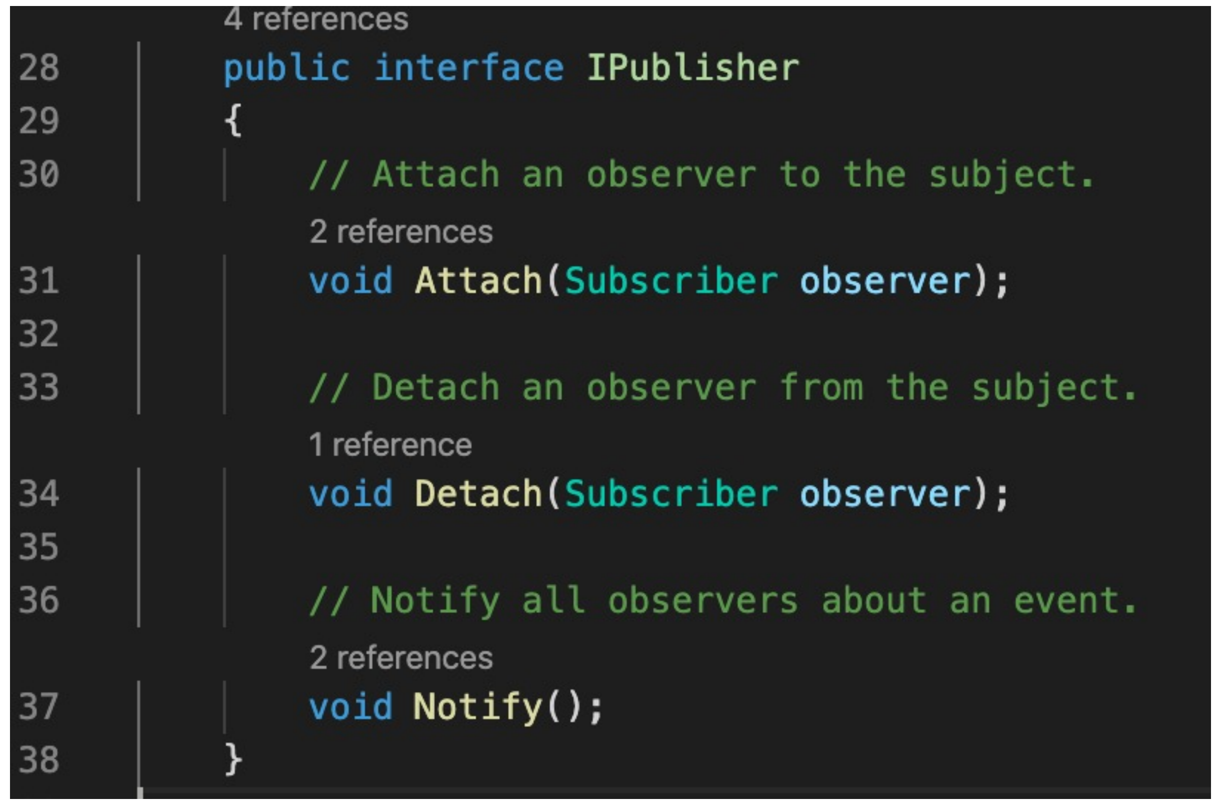


- On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products

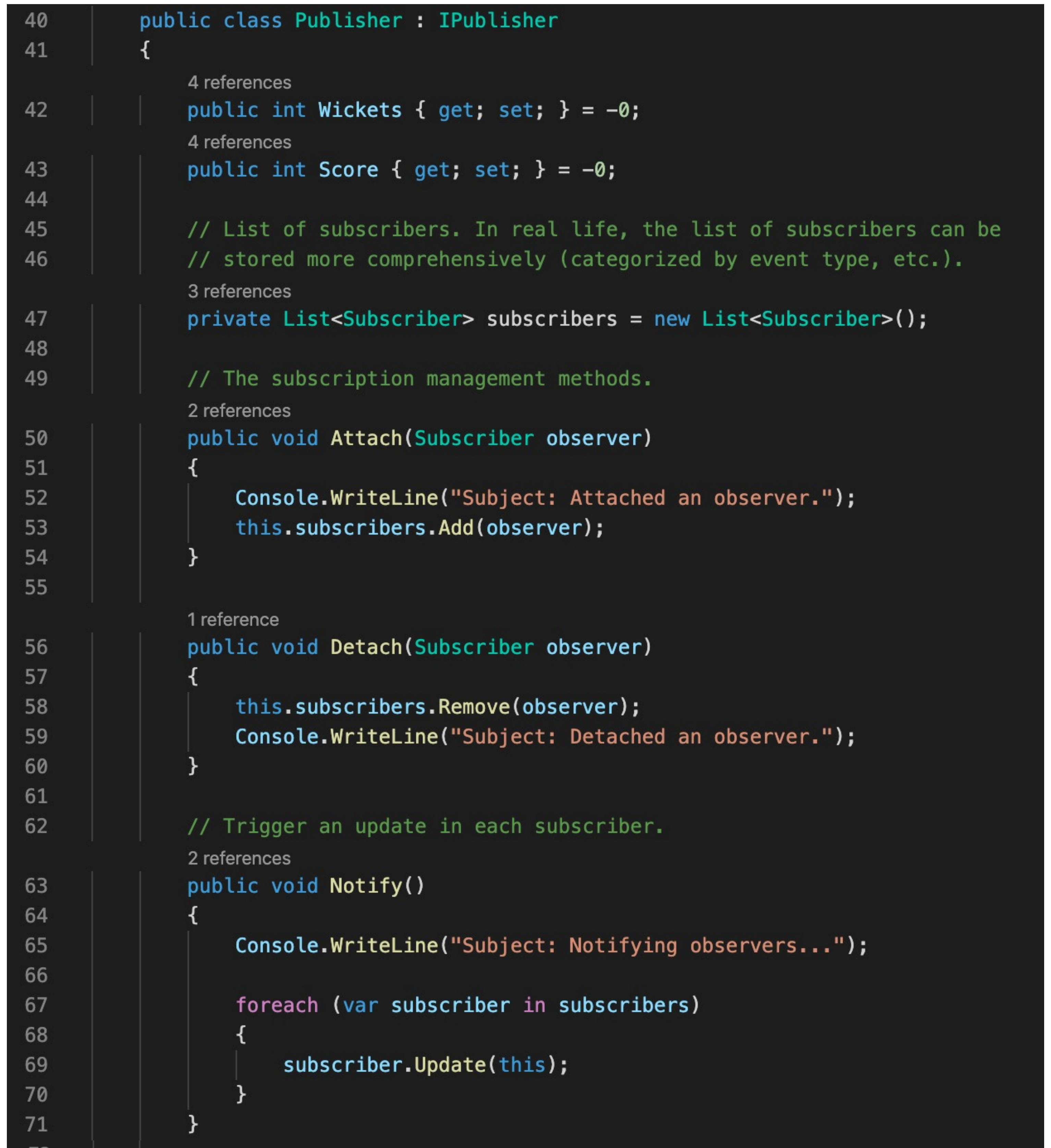
# Observer Pattern



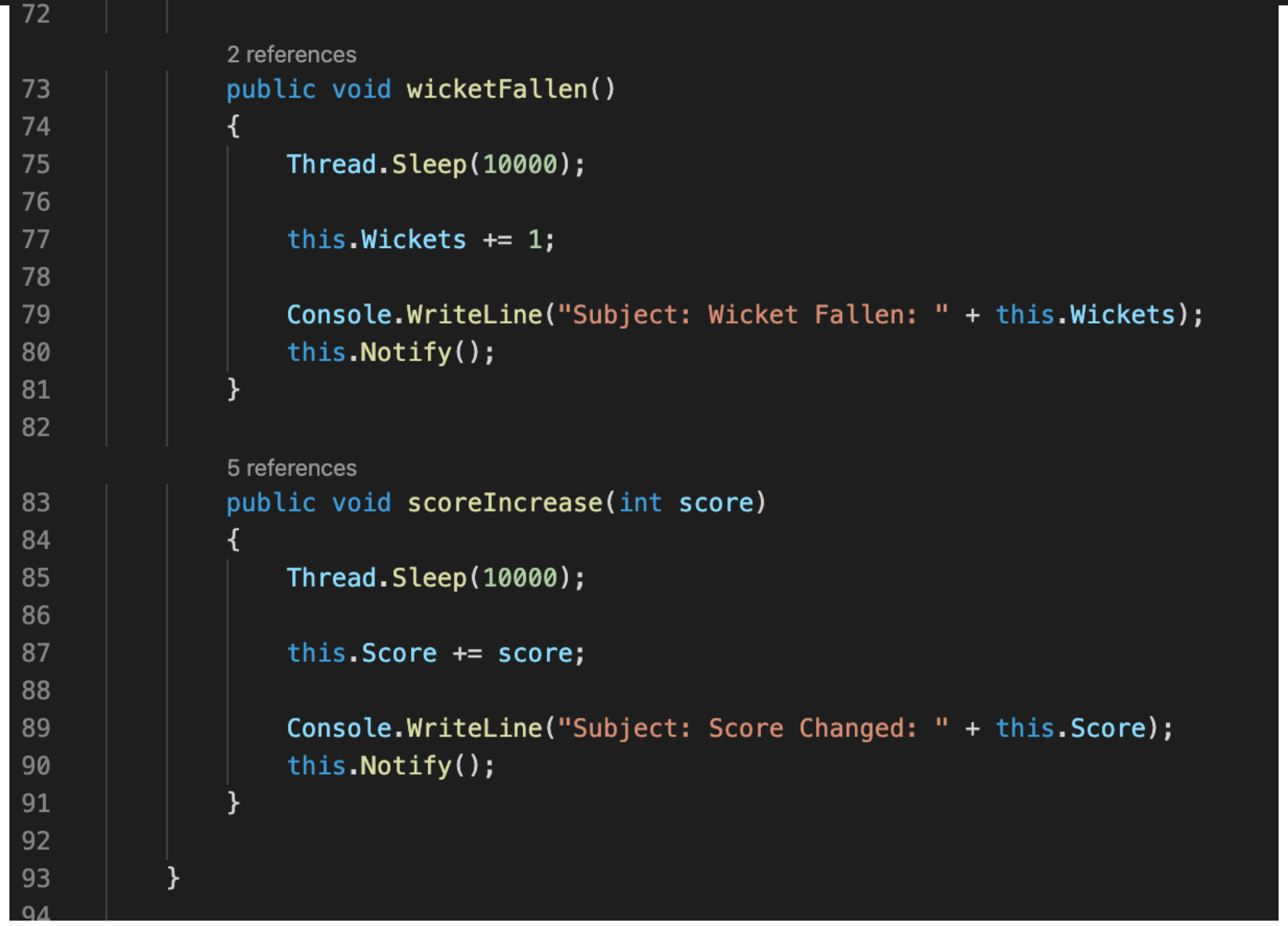
```
    8 references
    public interface Subscriber
    {
        1 reference
        void Update(IPublisher subject);
    }

    1 reference
    class ConcreteSubscriberA : Subscriber
    {
        1 reference
        public void Update(IPublisher subject)
        {
            Console.WriteLine("Mobile Device: "
            + (subject as Publisher).Wickets + "/" + (subject as Publisher).Score);
        }
    }

    1 reference
    class ConcreteSubscriberB : Subscriber
    {
        1 reference
        public void Update(IPublisher subject)
        {
            Console.WriteLine("Laptop Device: "
            + (subject as Publisher).Wickets + "/" + (subject as Publisher).Score);
        }
    }
```



```
    4 references
    public interface IPublisher
    {
        // Attach an observer to the subject.
        2 references
        void Attach(Subscriber observer);

        // Detach an observer from the subject.
        1 reference
        void Detach(Subscriber observer);

        // Notify all observers about an event.
        2 references
        void Notify();
    }
```

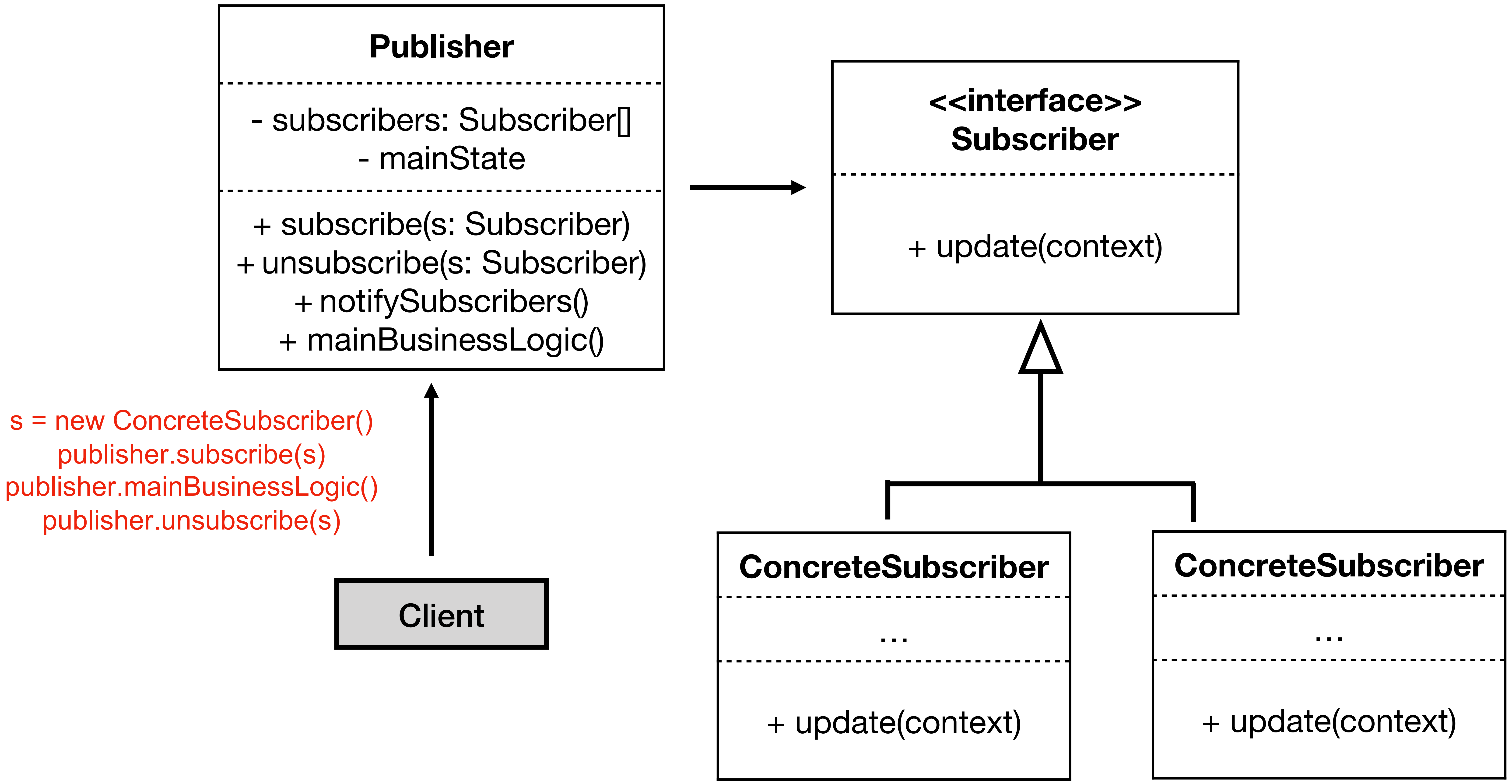


```
    public class Publisher : IPublisher
    {
        4 references
        public int Wickets { get; set; } = -0;
        4 references
        public int Score { get; set; } = -0;

        // List of subscribers. In real life, the list of subscribers can be
        // stored more comprehensively (categorized by event type, etc.).
        3 references
        private List<Subscriber> subscribers = new List<Subscriber>();

        // The subscription management methods.
        2 references
        public void Attach(Subscriber observer)
        {
            Console.WriteLine("Subject: Attached an observer.");
            this.subscribers.Add(observer);
        }

        1 reference
        public void Detach(Subscriber observer)
        {
            this.subscribers.Remove(observer);
            Console.WriteLine("Subject: Detached an observer.");
        }

        // Trigger an update in each subscriber.
        2 references
        public void Notify()
        {
            Console.WriteLine("Subject: Notifying observers...");

            foreach (var subscriber in subscribers)
            {
                subscriber.Update(this);
            }
        }
```

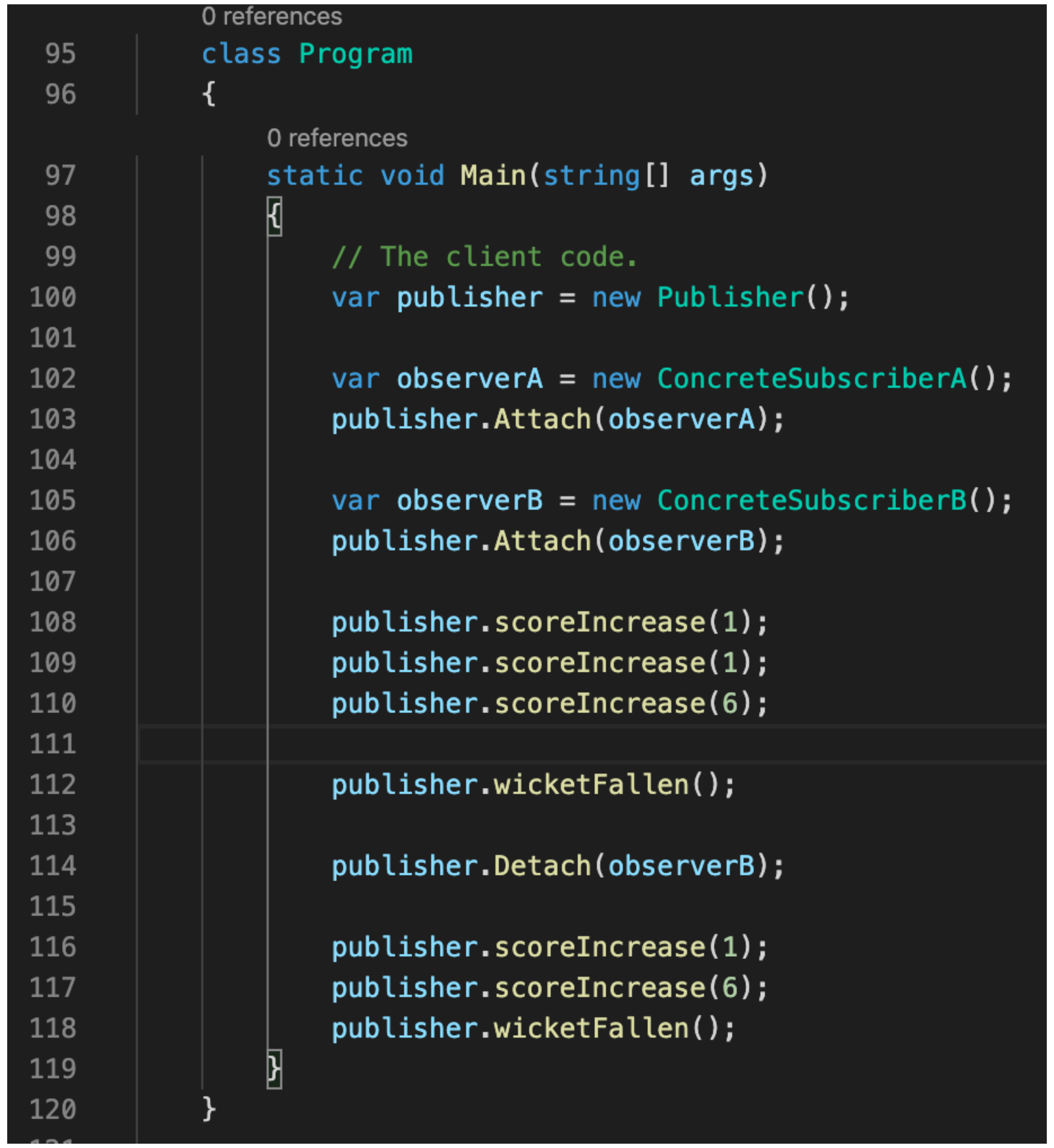


```

        2 references
        public void wicketFallen()
        {
            Thread.Sleep(10000);

            this.Wickets += 1;

            Console.WriteLine("Subject: Wicket Fallen: " + this.Wickets);
            this.Notify();
        }

        5 references
        public void scoreIncrease(int score)
        {
            Thread.Sleep(10000);

            this.Score += score;

            Console.WriteLine("Subject: Score Changed: " + this.Score);
            this.Notify();
        }

    }
```

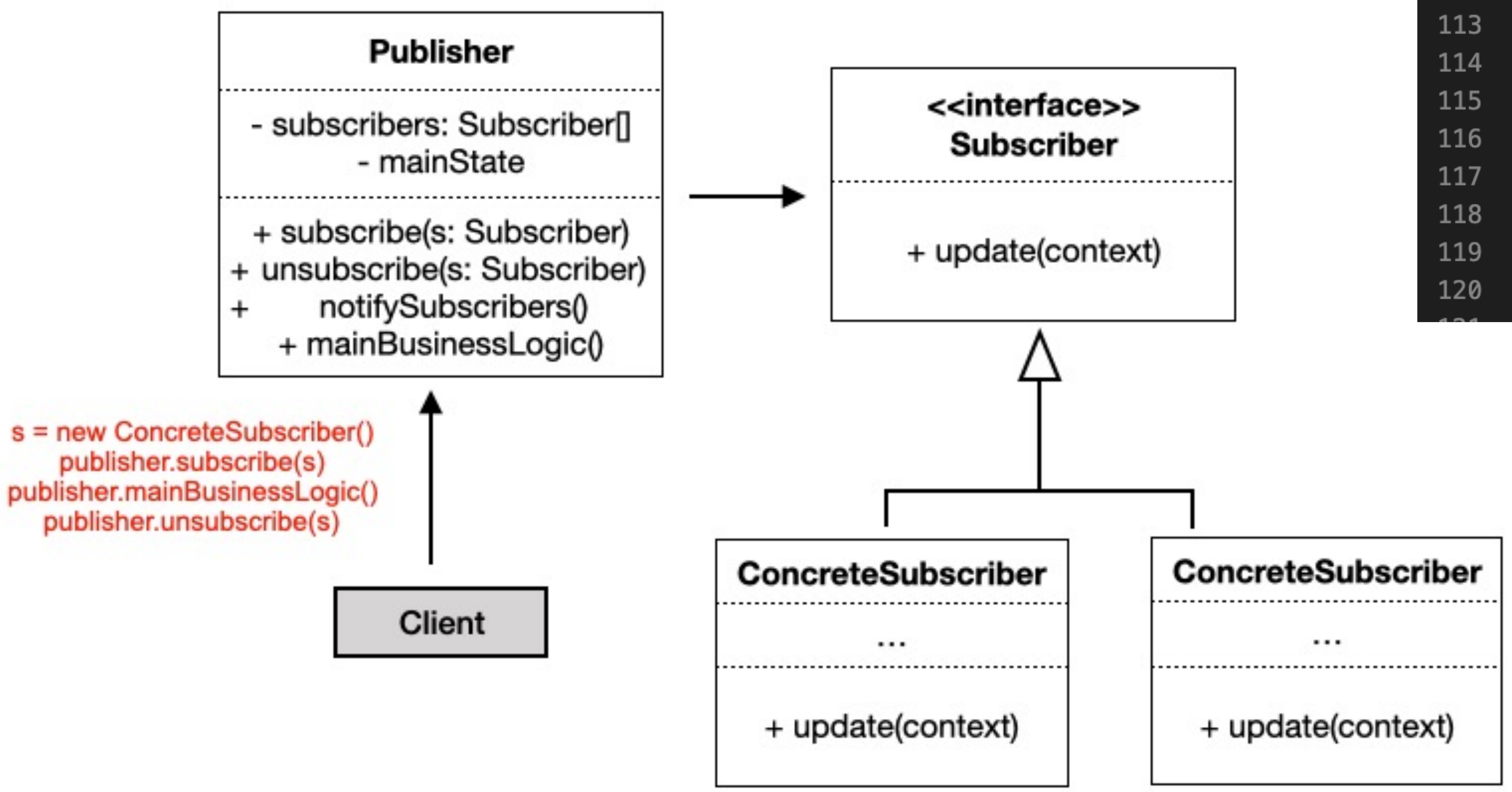
# Observer (Solution)



# Observer (Solution)



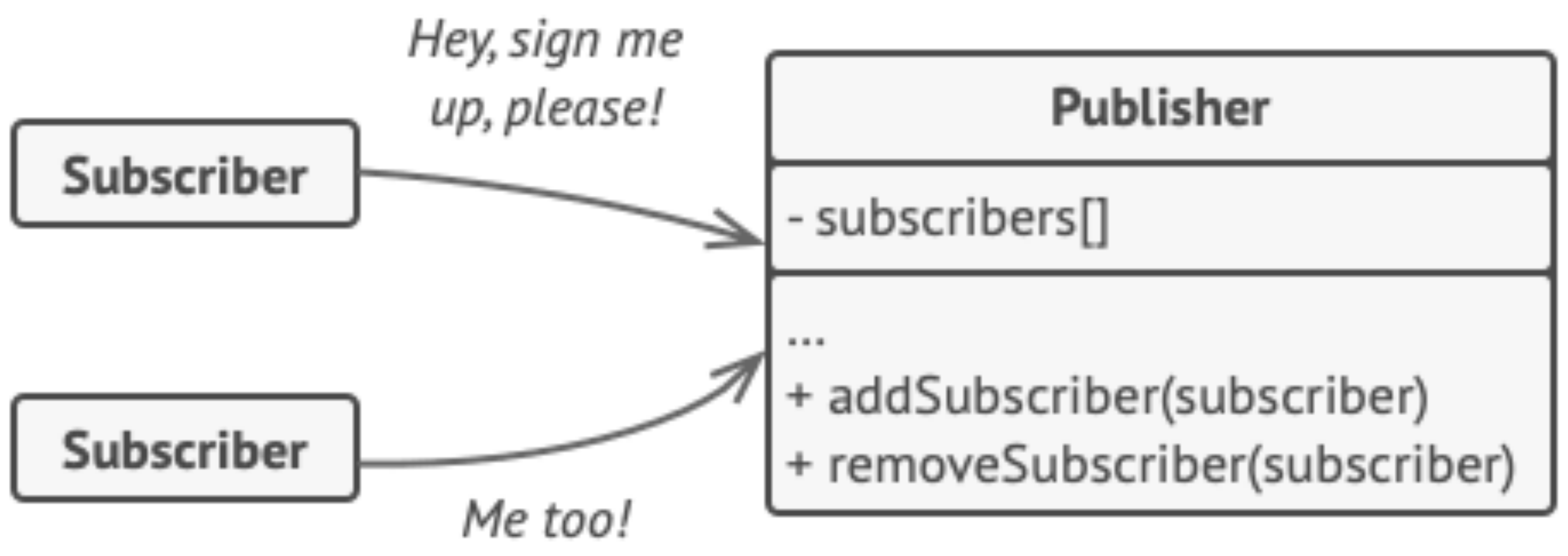
```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // The client code.
        var publisher = new Publisher();

        var observerA = new ConcreteSubscriberA();
        publisher.Attach(observerA);

        var observerB = new ConcreteSubscriberB();
        publisher.Attach(observerB);

        publisher.scoreIncrease(1);
        publisher.scoreIncrease(1);
        publisher.scoreIncrease(6);

        publisher.wicketFallen();

        publisher.Detach(observerB);

        publisher.scoreIncrease(1);
        publisher.scoreIncrease(6);
        publisher.wicketFallen();
    }
}
```



s = new ConcreteSubscriber()
publisher.subscribe(s)
publisher.mainBusinessLogic()
publisher.unsubscribe(s)

# Observer (Solution)

- The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it *publisher*
- All other objects that want to track changes to the publisher's state are called *subscribers* (observer)
- The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.
- Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects



# Observer (Use Cases)

- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically
- Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases

# Consequences

- **Abstract Coupling between Subject (publisher) and subscriber (Observer)**
  - All a subject knows is it has a list of observers, each conforming to an abstract and simple interface
  - Subject doesn't know the concrete class of any observer
  - Coupling is abstract and minimal
  - Subject and Observer can belong to different layers of the system as they are not tightly coupled
- **Support for Broadcast Communication**
  - Subject need not specify the subscribers for its message
  - Message is sent to all interested parties who are subscribed
  - Only responsibility of subject is to notify observers
  - Can add or remove observers at will

# Consequences

- **Unexpected Updates**
  - As observers are unaware of each other, they cannot know the cost of changing the state of the subject
  - A seemingly innocuous operation on the subject (publisher) could cause a cascade of updates to observers and their dependent objects
  - Dependency criteria which are not well-defined or maintained often lead to spurious updates
  - These can be hard to track down, especially with a simple Update protocol, which doesn’t provide details on what changed in the subject

# Word of Caution

# Word of Caution

- Design Patterns have become an object of some controversy in the programming world in recent times, largely due to their perceived ‘over-use’ leading to code that can be harder to understand and manage.
    - It’s important to understand that Design Patterns were never meant to be hacked together shortcuts to be applied in a haphazard, ‘one-size-fits-all’ manner to your code. There is ultimately no substitute for genuine problem solving ability in software engineering.
    - The fact remains, however, that Design Patterns can be incredibly useful if used in the right situations and for the right reasons. When used strategically, they can make a programmer significantly more efficient by allowing them to avoid reinventing the proverbial wheel, instead using methods refined by others already. They also provide a useful common language to conceptualize repeated problems and solutions when discussing with others or managing code in larger teams.
    - That being said, an important caveat is to ensure that the *how* and the *why* behind each pattern is also understood by the developer.

# Summary