Creational Patterns 1: Factory & Abstract Factory

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 9

Last Class

Structural Patterns

- Patterns that help us define complex structures and data types that reduce coupling and can be dynamic at run-time

Proxy

- Use a wrapper to control access /manage an expensive or limited object/service.

Façade

- Provide a simpler interface to a more complex interface/sub-system/3rd Party API

Bridge

- Split a large set of classes (or one large class) into separate hierarchies and introduce composition between the base classes of these hierarchies.

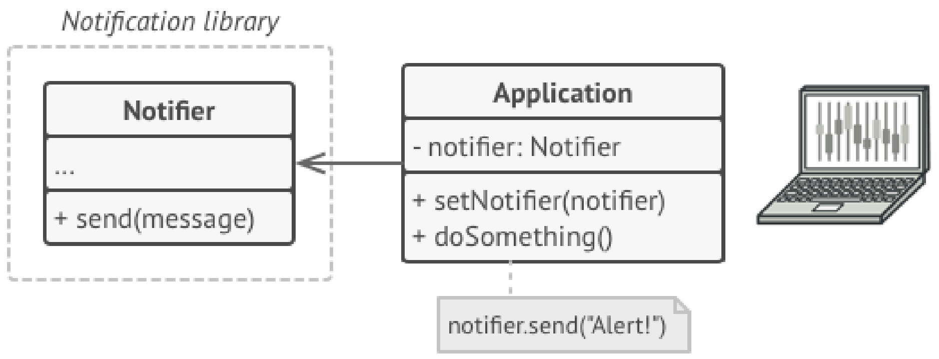
Decorator

- Add additional behavior to an object dynamically without affecting behavior of other objects of the same class

Review: Let’s take a look at a real world example..

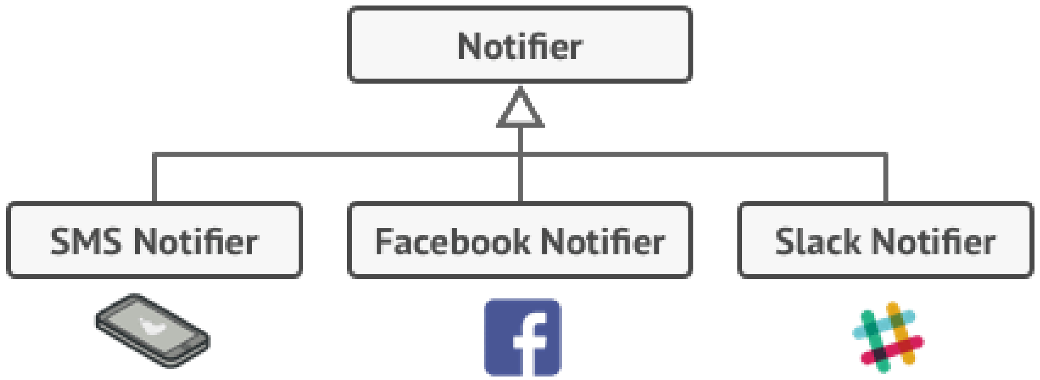
Say we have a notification system.

When something happens we want to send a notification, perhaps via email.



But then over time, we realize that we may want to send notifications via other channels as well. So we do what any sensible programmer would do.

We inherit! Notifier is now a base class and we can have different notifications. Easy right?



Review: Let’s take a look at a real world example..

Wait a second, what if we want to send a notification alongside multiple channels at once?

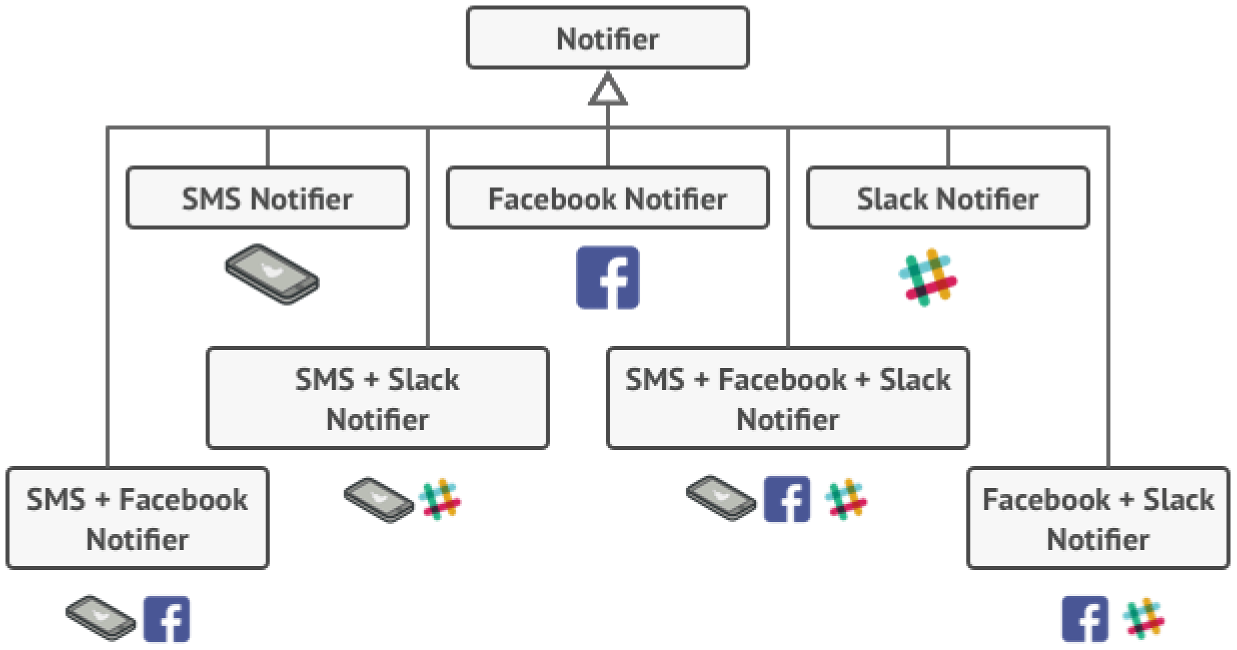
Different users have different devices or apps.

Person A might receive facebook and sms notifications, while **Person B** might receive email and slack notifications.

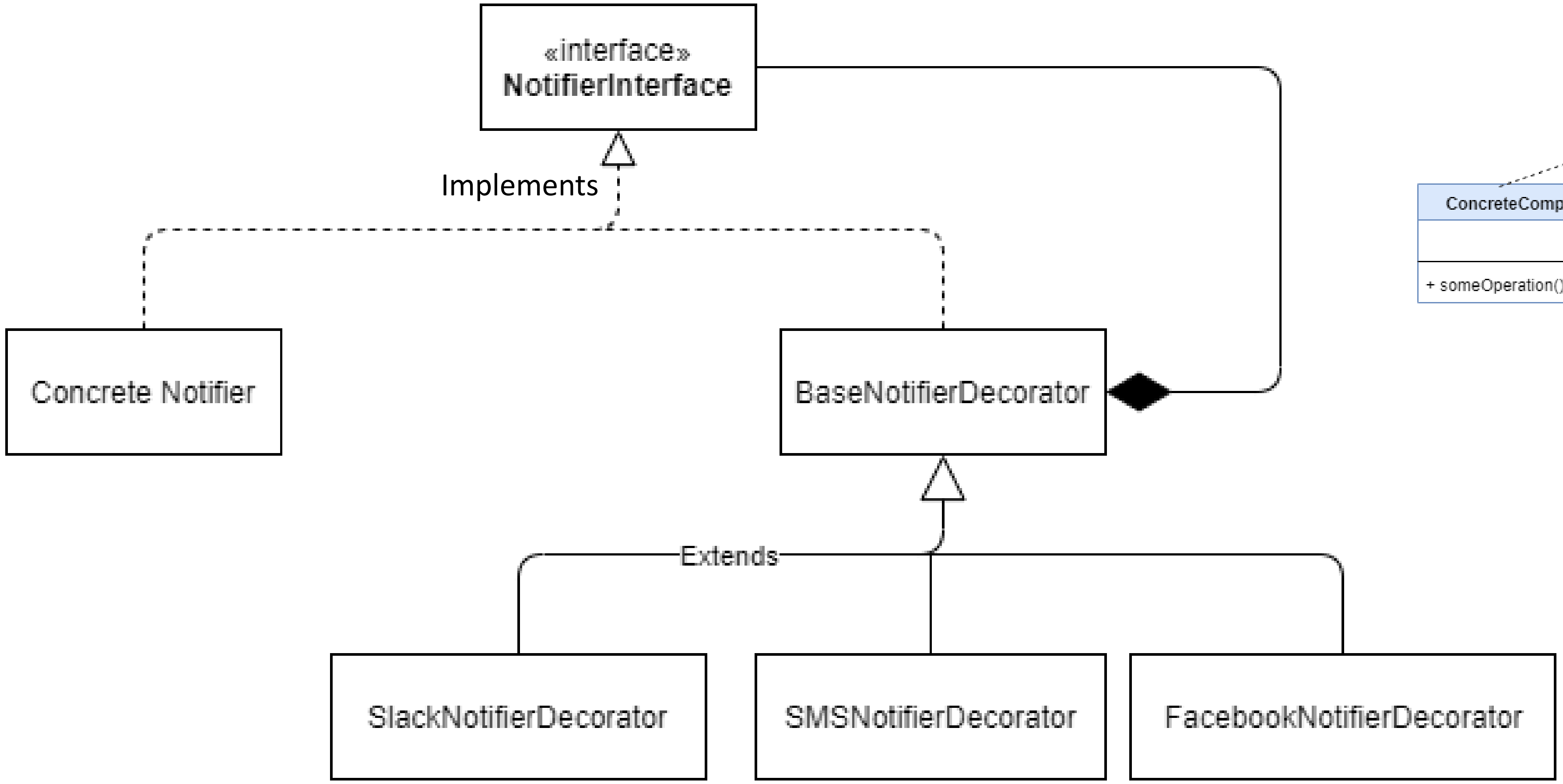
Person C may receive all!

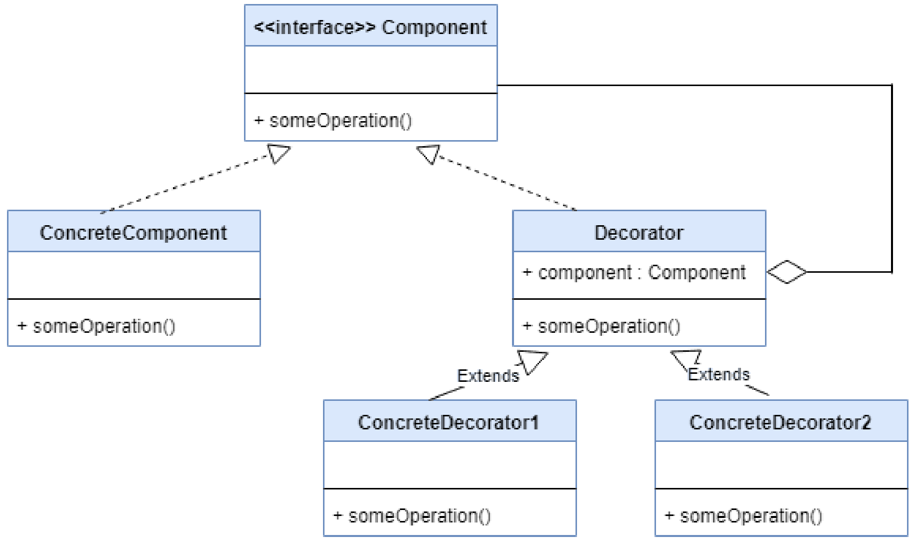
This inheritance hierarchy is becoming convoluted and difficult to maintain. There must be a simpler solution!

What if we defined each channel as a decorator that could wrap around a Notifier?



Review: Notifier Decorator

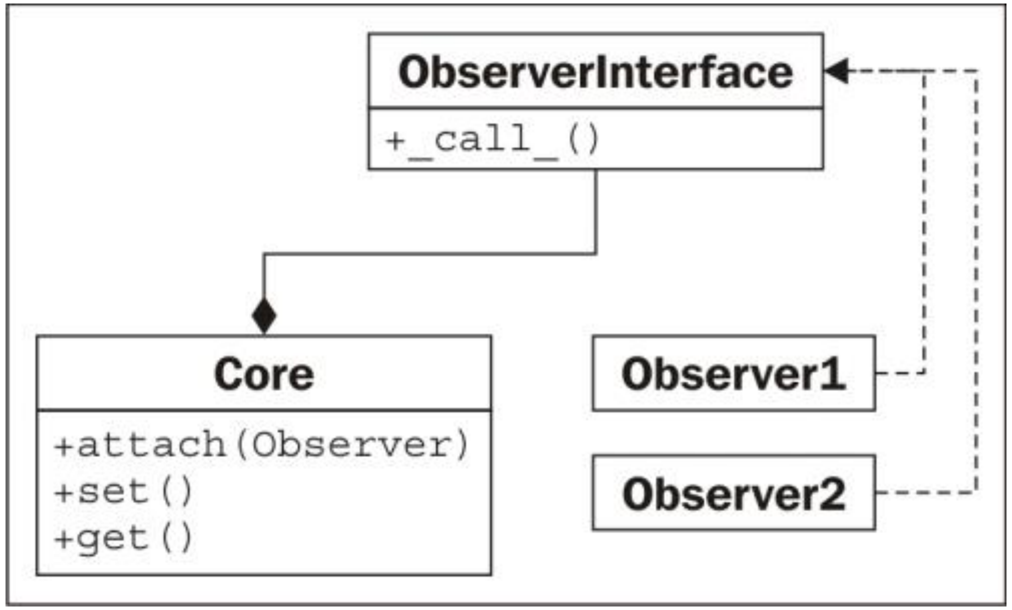




Review: The Observer Pattern

These are the requirements:

1. The core doesn’t care who’s observing it or what they do when a change occurs.
2. Observers are notified when something changes in the subject
3. All the core cares about is the fact that **it has to run a set number of callbacks** that have the same interface to notify <u>*some*</u> observers that do <u>*something*</u>.



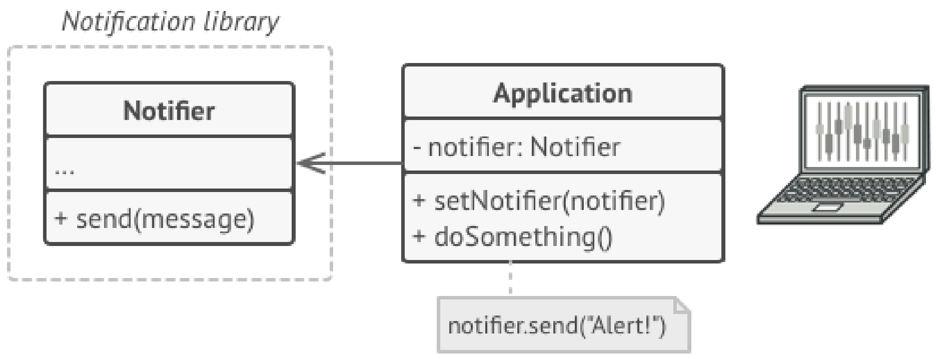
Decorator vs Observer implementation

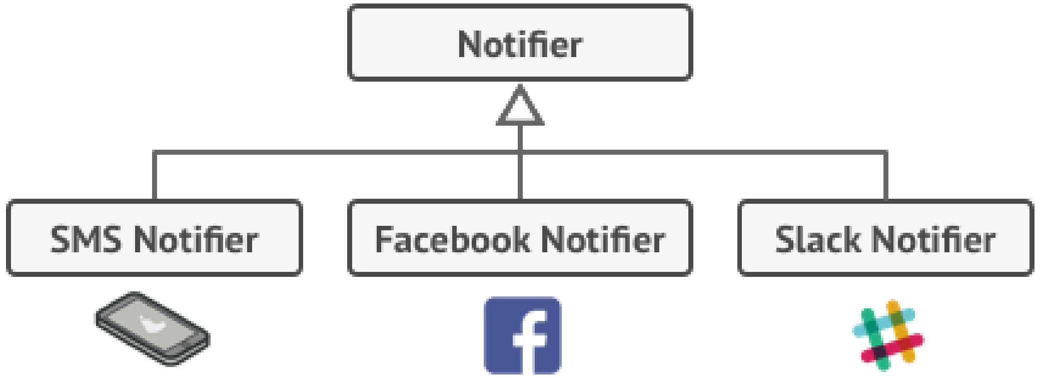
Had a great question at the end of last class:

Can we use the observer design pattern instead of the decorator pattern to implement the notification system?

Definitely!

The results may not be exactly the same but depending on the requirements it's possible





Decorator vs Observer implementation

Decorator implementation

Have a base notification component

Each type of notifier (SMS, Facebook, Slack), is a decorator

Notifiers are added dynamically/at run time to the base notification component depending on the type of notifiers the Person wants

Person can indicate what types of notifiers they want

Observer implementation

Each person is an observer

Each type of notifier (SMS, Facebook, Slack), is a core.

People observers can be added to a core dynamically/at run time

Notifications sent to all observers who are subscribed to that notification (maybe this isn't desired)

Possible to work around this. ie: Core sends notification with a person id

Design patterns are flexible

Can use different design patterns to solve the same problem

Can merge design patterns together (Singleton with Façade is common)

It’s all up to the design and what we’re attempting to model

Design patterns are flexible

Donut shop example:

- We used the decorator pattern to add toppings to donuts
  - This seems to closely model the idea of an object that has things added to it
- Theoretically modelling donut creation would work with the observer design pattern as well
  - Each topping is a core, base donut are observers
  - Donuts that want toppings attach themselves to the topping
  - Toppings notify donuts when to add the topping to itself
  - This seems kind of a round about way to create donuts…

Generally, we attempt to write code that models/simulates the problem as closely as possible

Bottom line: <u>keep it simple and use the design pattern that most fits the problem</u>

Creational Patterns

LET’S START CREATING

Categorizing Design Patterns

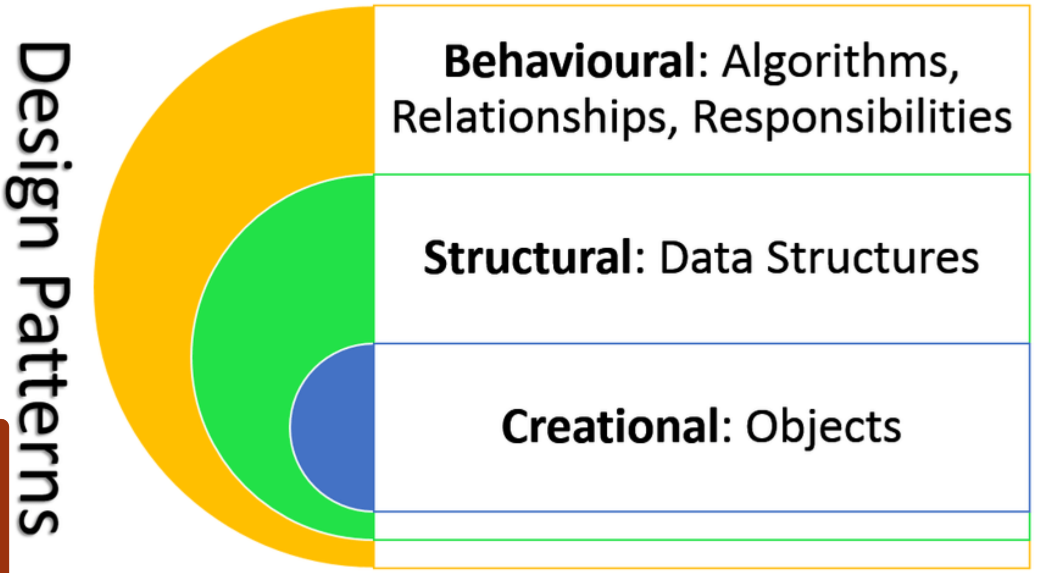
- ❑ **Behavioural**

Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- ❑ **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- ❑ **Creational** **(We are looking at these!)**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects


Design Patterns
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

Today we will be looking at 2 patterns…

Factory (Also known as Factory Method)

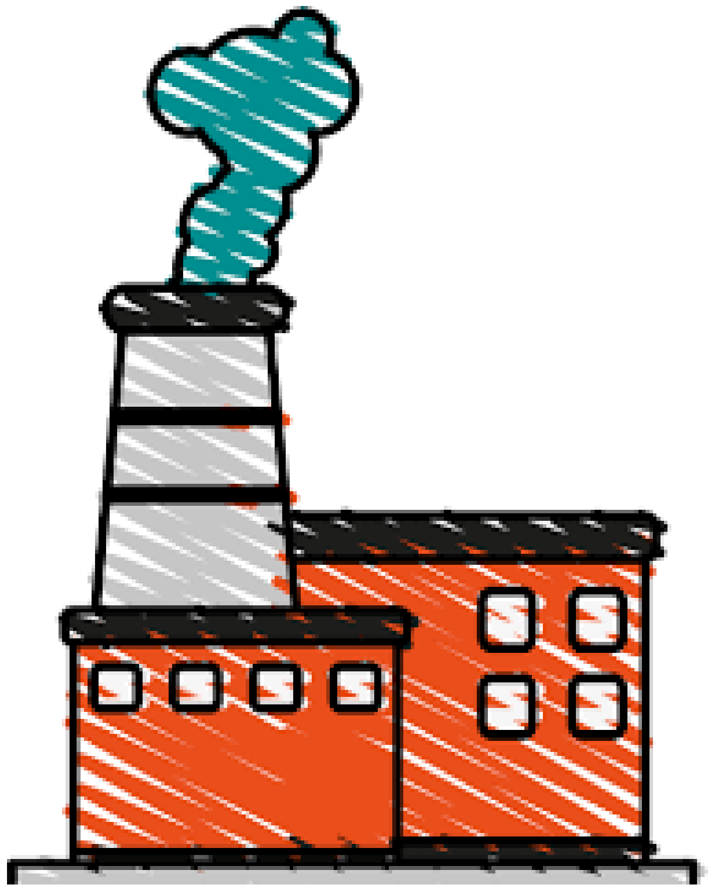
- When you want to separate creation code from client code.
- When you want a client or service to only depend on a base class for object creation and not each concrete class.

Abstract Factory

- A more evolved form of the Factory Pattern
- Allows for the creation of a group of related objects .
- When you want to create different variants of families of objects.

Factory Pattern

First let’s clarify the word Factory…



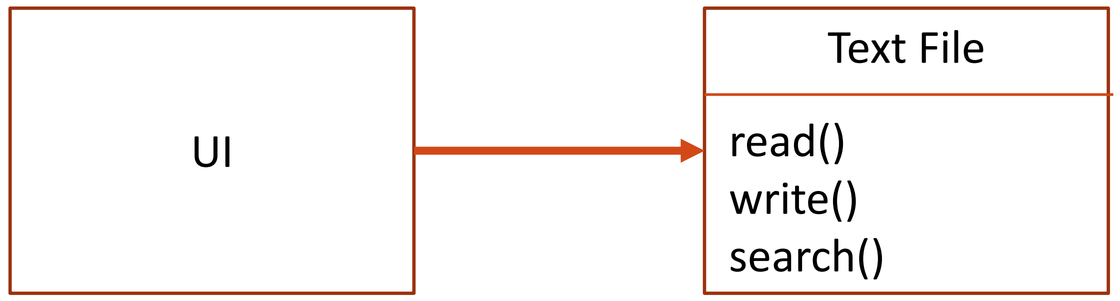
- Something that is responsible for making things
- In this case objects.
- The word factory is thrown around a lot, don’t mix it up with the Factory Pattern or the Abstract Factory Pattern. They are also responsible for making objects but are different.
- A Factory could refer to:
  - The pattern(s)
  - A class that creates objects
  - A static/class method that generates objects.

Some more lingo – Creation Method

- A method or function that is responsible for creating an object. It is a factory.
- Different from the constructor, although you can argue that a constructor is a creation method too. A creation method can be a wrapper around a constructor call. (Check the example below)
- A creation method doesn't have to make a new instances of a class, it can return a cached object or even re-use objects from a collection (Object Pool Pattern).
- Resist the urge to call it a Factory Method. The Factory Pattern is also known as the Factory Method Pattern and it can cause all sorts of confusion.

```
@classmethod
def generate_random_asteroid(cls, min_speed,
max_speed):
     speed = random.randomint(min_speed, max_speed)
     asteroid = cls()
     asteroid.speed = speed
     return asteroid
```

Remember the dependency inversion principle?



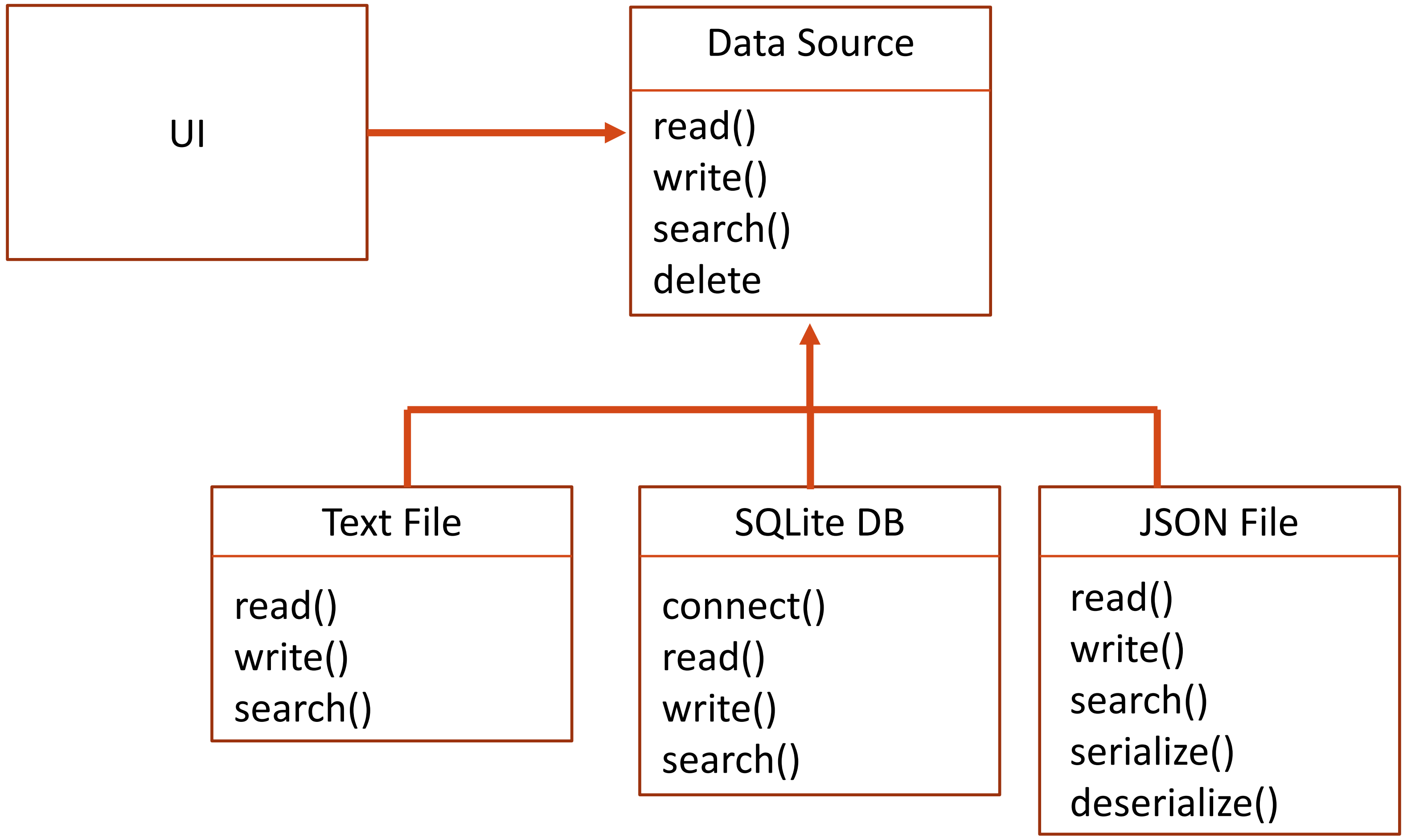
Say we were developing an app with a UI that was populated with some data from a text file.

After a few releases and years of development, for some reason we decide to switch out to a more secure source of data , perhaps an encrypted JSON file or even perhaps a SQLite Database. We would have to edit all the modules/classes that dealt with our app's UI!

Remember the dependency inversion principle?

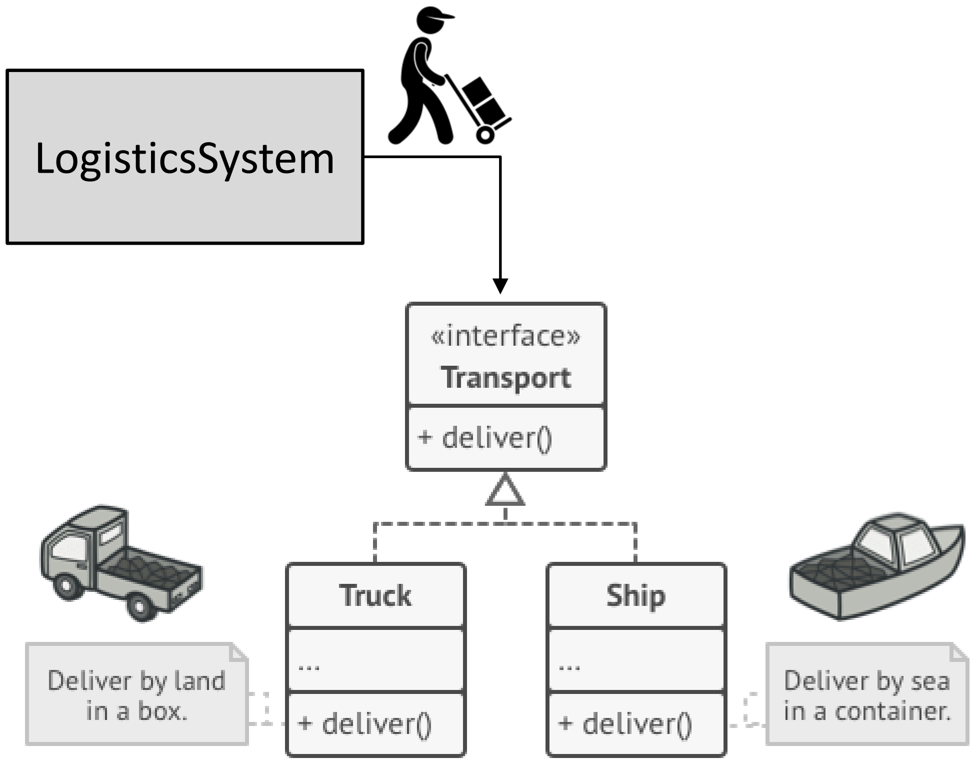
We can decouple our system to instead depend on a data source which is an **abstraction** that **hides** different data sources.

Our UI would just have to be **dependent on a common interface** provided by the Data Source and not be concerned with how that data source is implemented



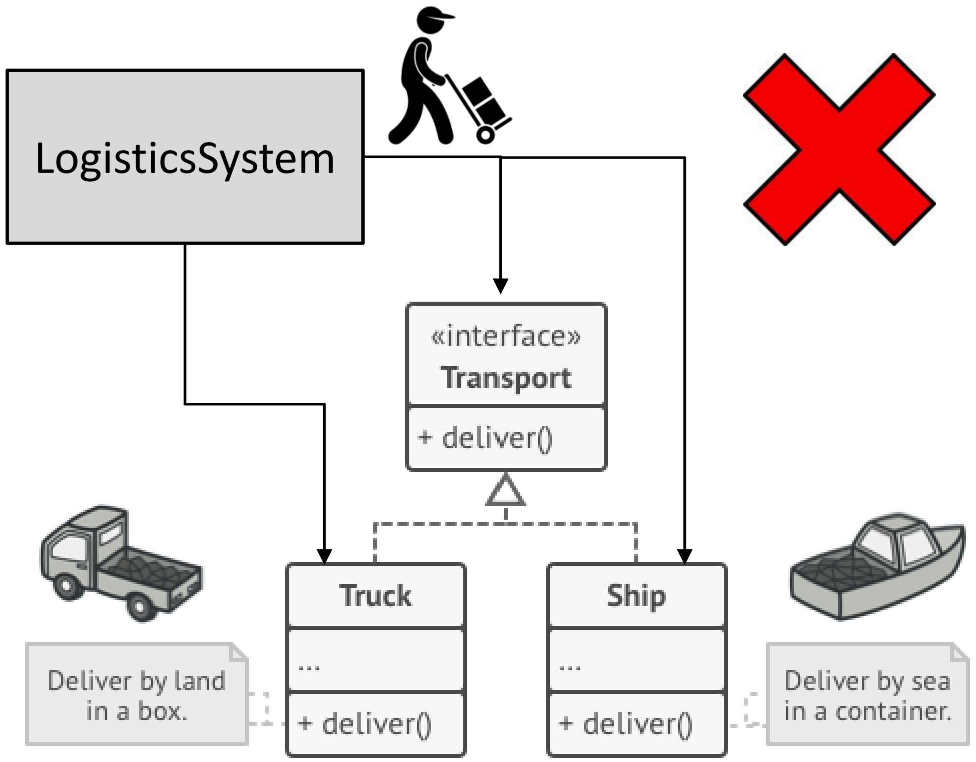
Factory Pattern

- A way to decouple client code that uses an object or a service, from the code that creates an object or a service.
- Consider a scenario where a system is managing the logistics of delivering goods.
- If inheritance is done correctly, all a Logistics System should only care about depending on the Transport Interface, not the concrete transports (Truck or Ship). This follows **the Dependency Inversion Principle**.



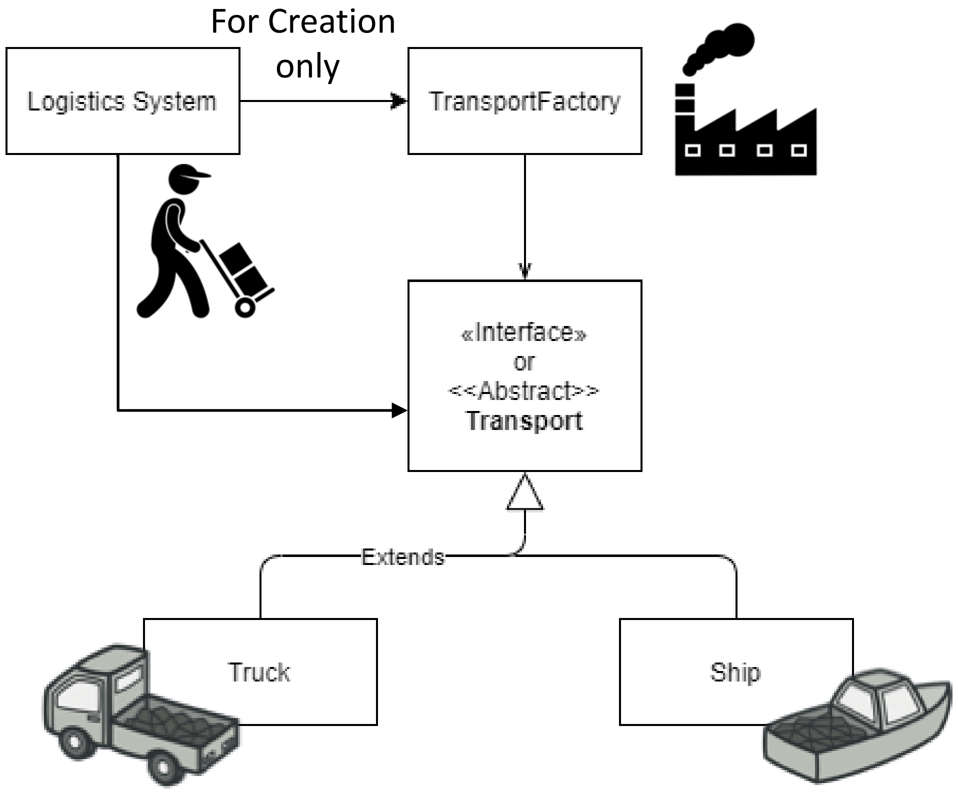
Factory Pattern

- But at some point we need to **create the concrete class**. If the Logistics System did that, then it would need to call the Truck() or Ship() constructors.
- This would make it dependent on the concrete classes and break the Dependency Inversion Principles. A high level module is now dependent on a low level module.



Factory Pattern

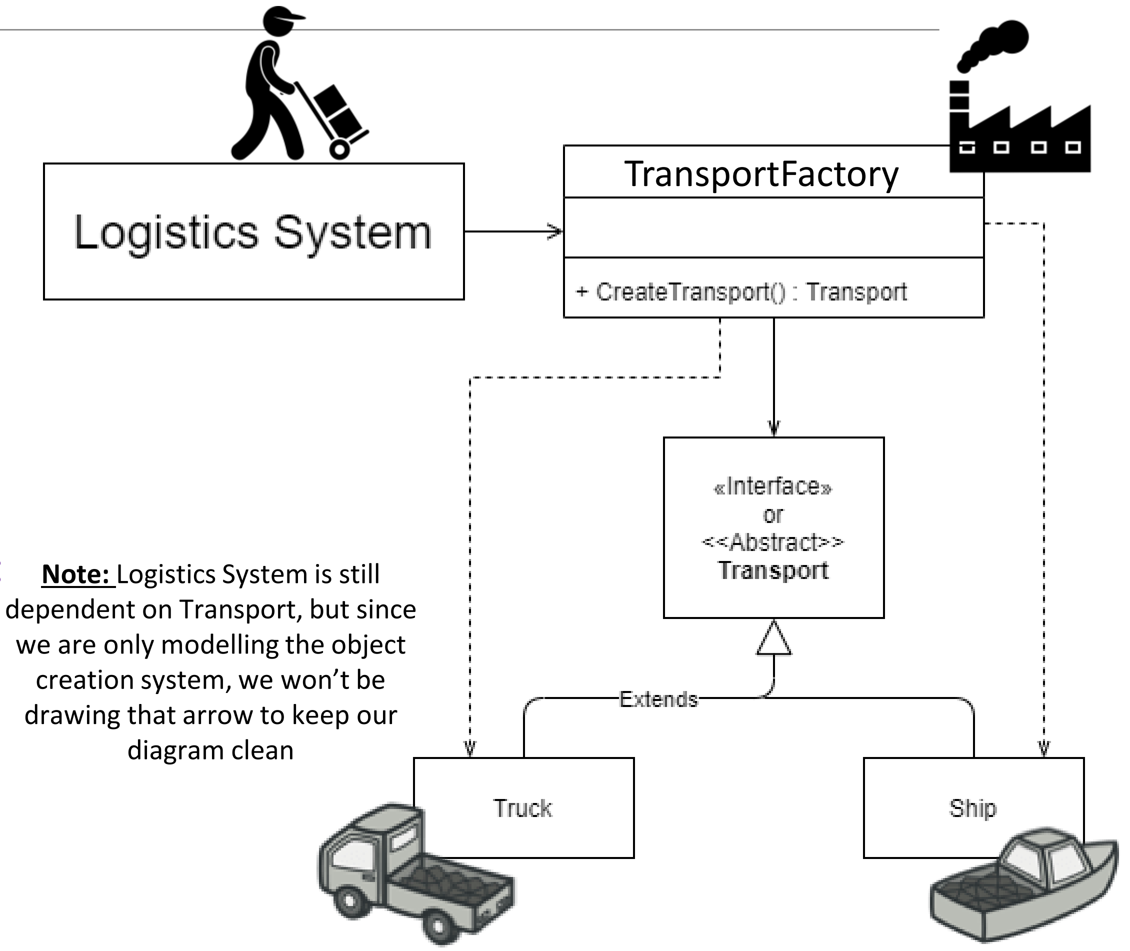
- So when it comes to creating these objects we need another entity to handle the responsibility of creating a specific type of Transport.
- We need a factory which will create a concrete transport and return it to the Logistics System.
- As far as the Logistics System is concerned. It doesn't necessarily know that it is receiving a Truck, Ship, or Train. All it needs to know is that it is receiving a Transport.



Factory Pattern

- This still isn't enough. All we have done is move the call to our Truck(), Ship() or Train() constructor to the TransportFactory. This isn't that useful.
- Say our TransportFactory has a CreateTransport() method. Now this method will have a massive if-else or switch statement block where it will be dependent on the concrete Truck and ship Classes

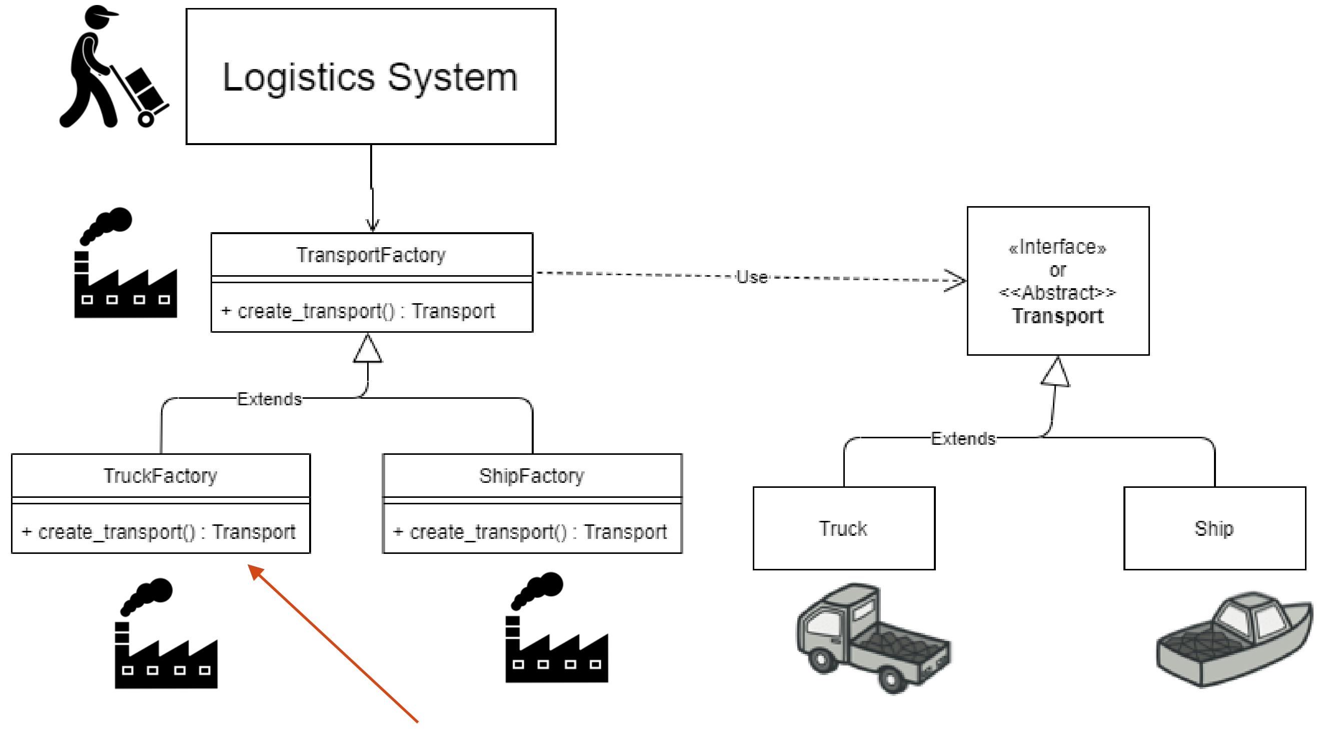
```
def CreateTransport(self) -> Transport:
        if self.current_type == Ship:
                return Ship()
        else:
                return Truck()
```



Note: Logistics System is still dependent on Transport, but since we are only modelling the object creation system, we won't be drawing that arrow to keep our diagram clean

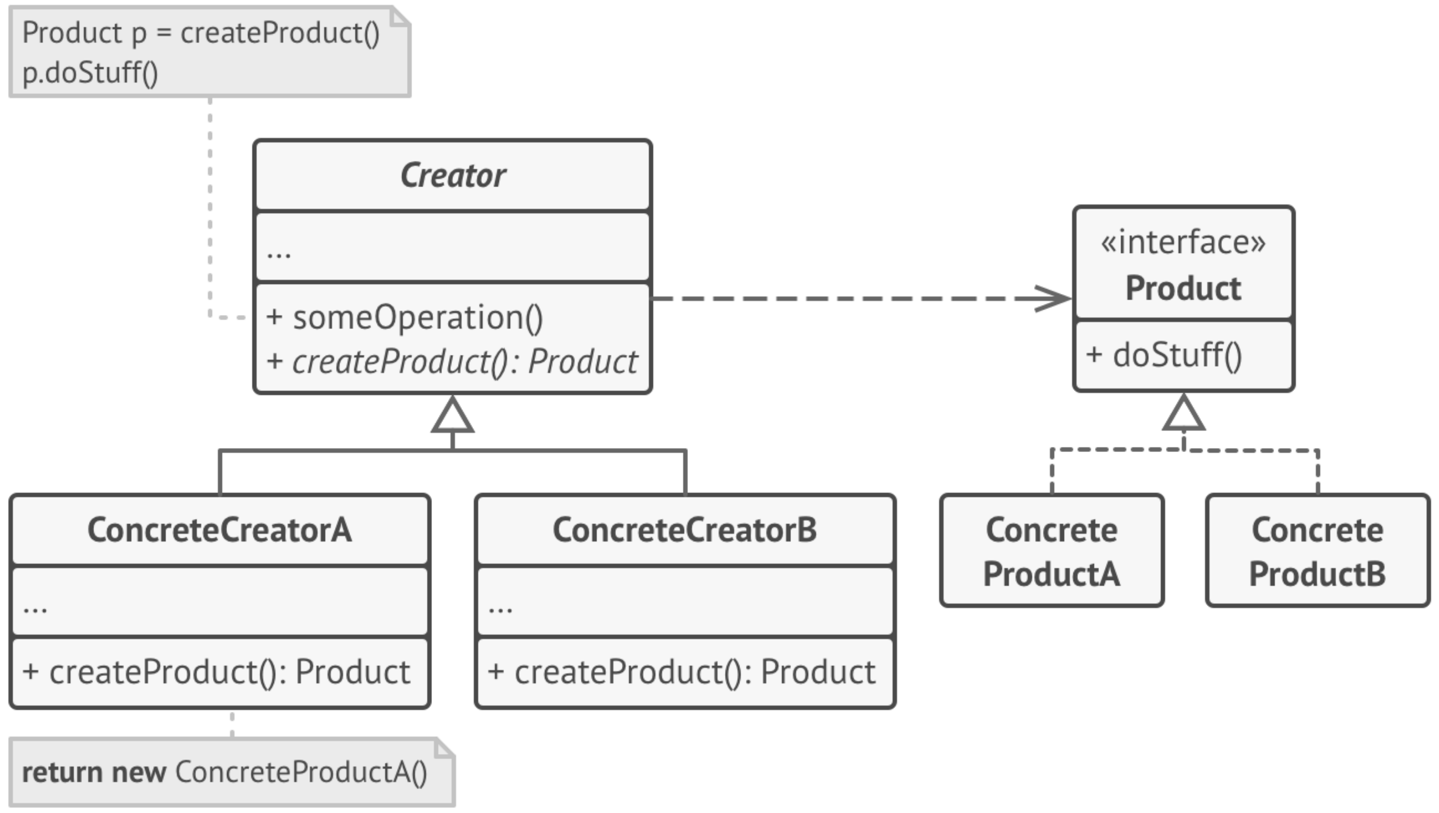
Factory Pattern

- That's better.
- We can simply inherit from TransportFactory and override the create_transport() method.
- Hey, doesn't this kind of look like the **Bridge Pattern**?
- It draws from the same philosophy. We have two split hierarchies that are easier to maintain. But we can't mix and match combinations here.
- All we need to do is supply the Logistics System with the right factory. (Which can be done by a controller).
- As far as the Logistics System is concerned it needs an object with the TransportFactory interface.



```
def CreateTransport(self)-> Transport:
    return Truck()
```

Factory Pattern



Let’s Build a Factory Pattern Together.

Let’s say we need to implement a Factory Pattern for a website hosting a forum messaging board. This website creates a new user object when the user logs in, signs up to be a member or views the forum as a guest.

Our system can deal with two kinds of users. A **Guest** and a **Member**.

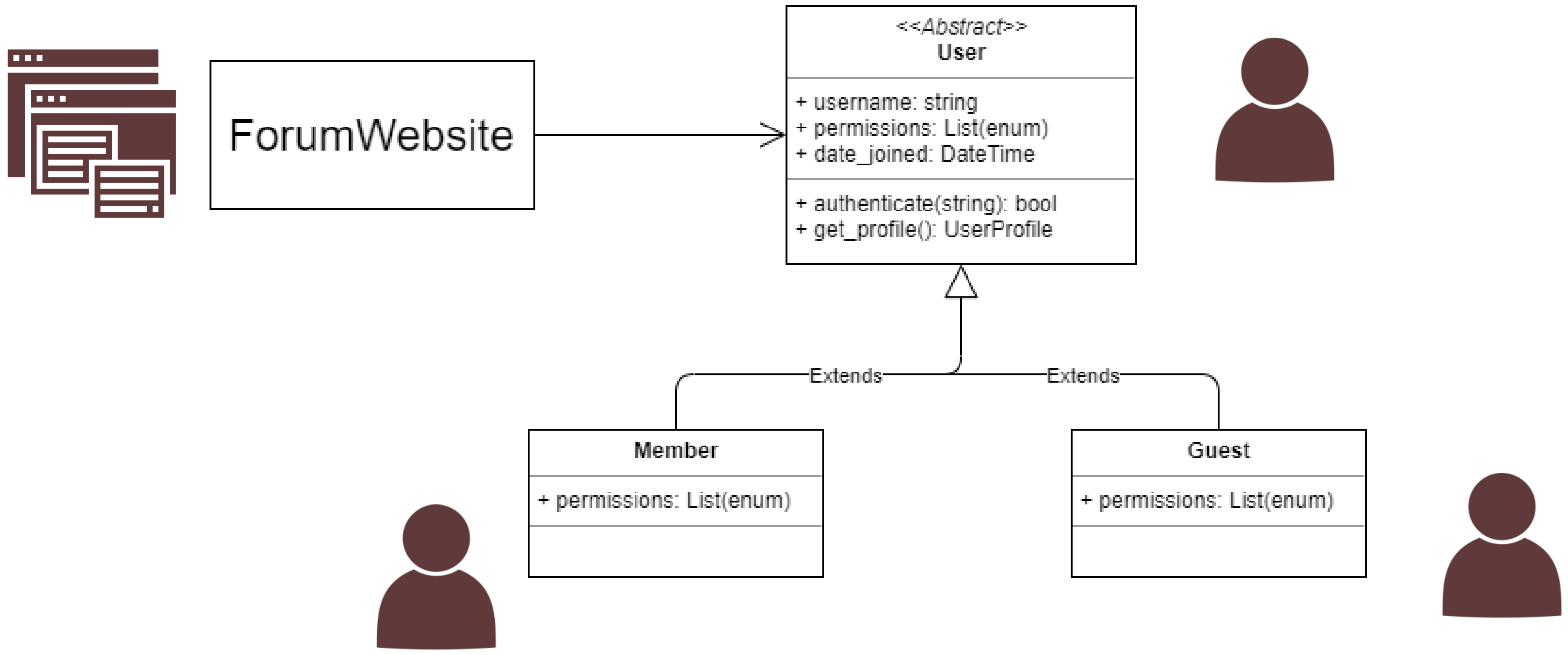
A Guest has the following **permissions** (this is our attribute, it is a tuple):

- Read Posts
- Like
- Share Posts
- Flag Posts

A Member can do everything a guest can, but they can also **write** posts.

Step 1: Let’s create a UML Class diagram depicting **the User hierarchy.**

Step 1: Define a product hierarchy



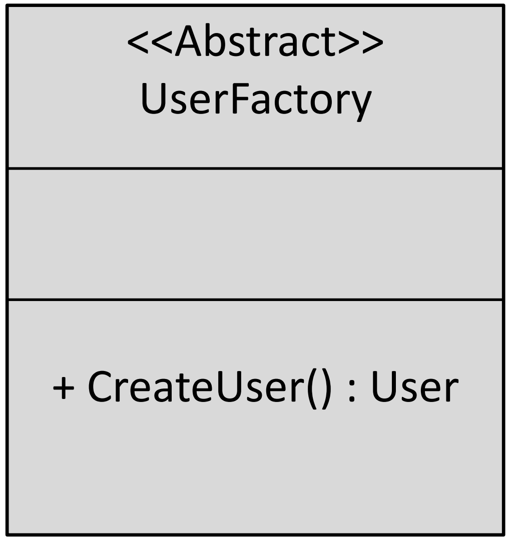
ForumWebsite
<<Abstract>>
User
+ username: string
+ permissions: List(enum)
+ date_joined: DateTime
+ authenticate(string): bool
+ get_profile(): UserProfile
Extends
Extends
Member
+ permissions: List(enum)
Guest
+ permissions: List(enum)

Step 2: Define a factory class

- Let’s create a UserFactory class.
- Add it to the UML diagram
- Specify the creation method.

Step 2: Define a factory class

- In the Factory Pattern, a Factory class doesn't need to solely be responsible for creating objects. It can be a larger class with a creation method.
- **The base factory doesn't need to be abstract**. It can create a default product. In our case, let's make it abstract though.
- For example a UserFactoryClass might also keep track of how many user accounts are currently active.

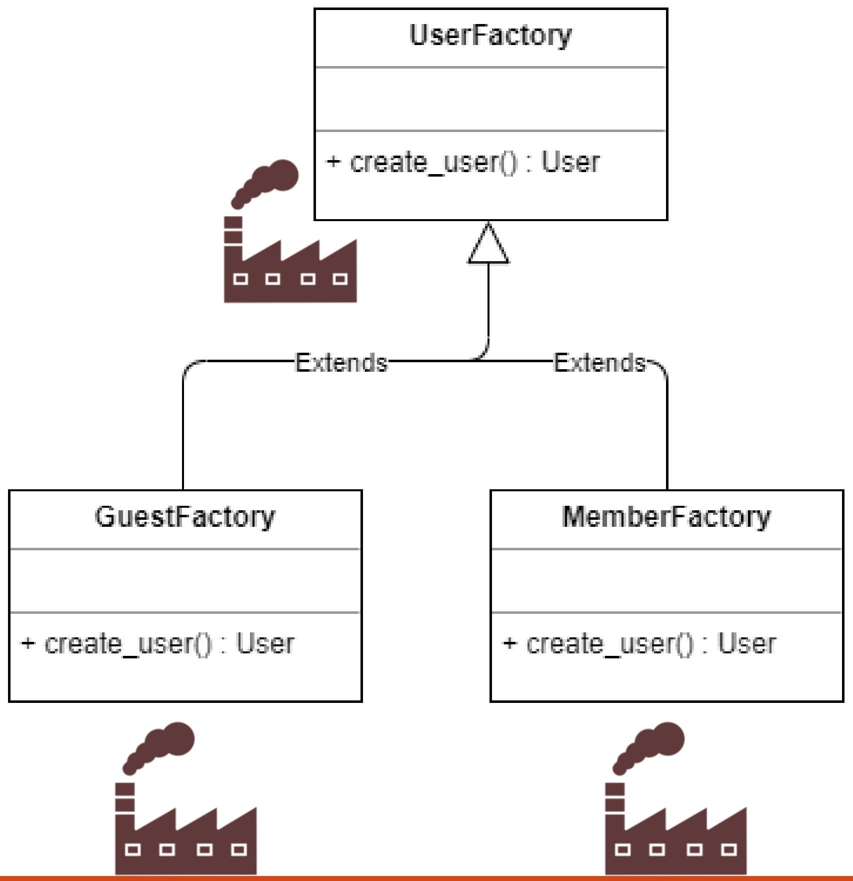


Step 3: Create a Factory hierarchy and override the creation method.

The title says it all.

We want to create our concrete factories in this step.

Each concrete factory will create a new object of it's corresponding user type and return it.



Step 4:
Integrate it into our Forum Website

Bring it all together.

Add a client class, and mark the associations between Client, Factories and Users.

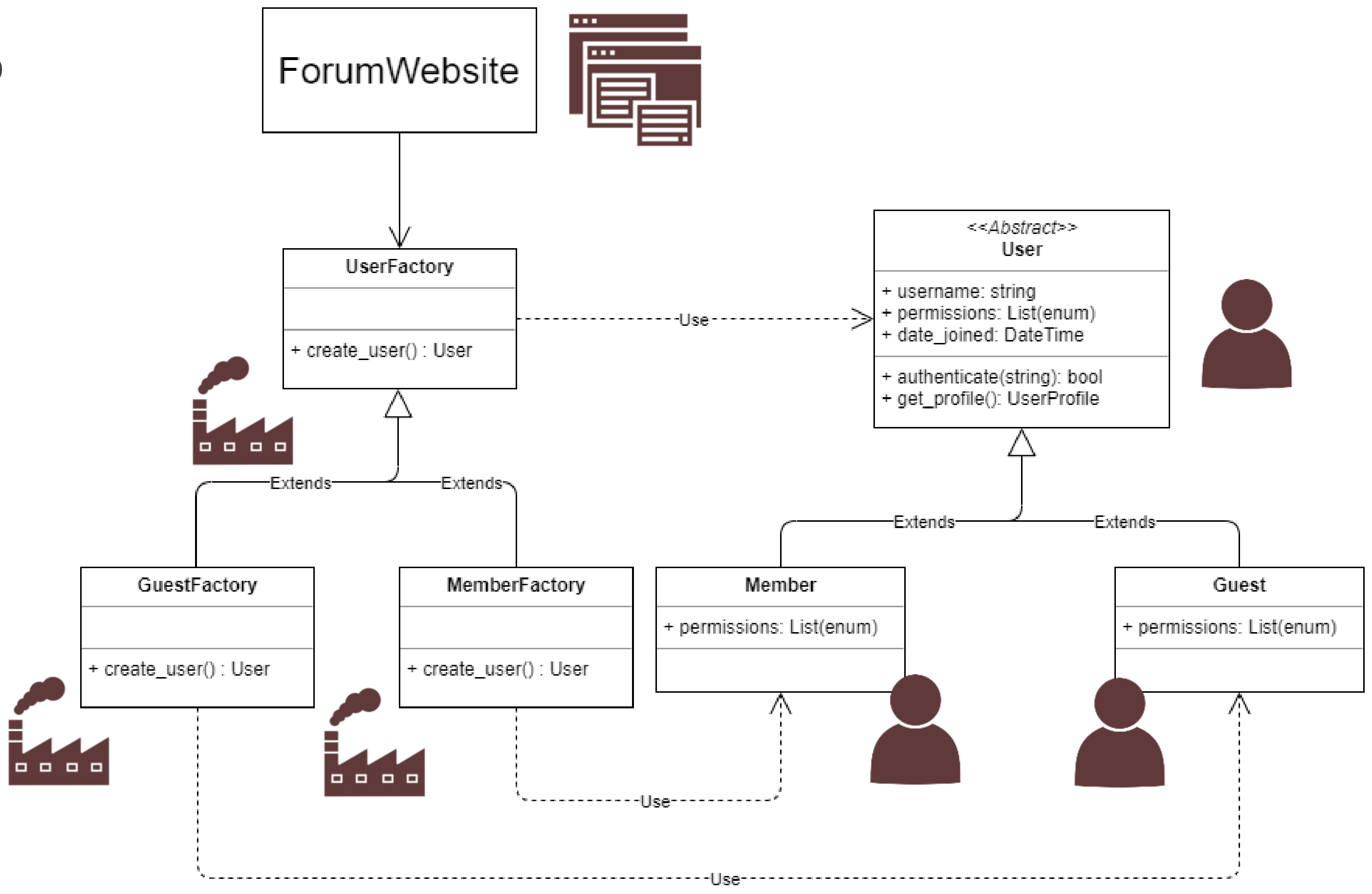
Step 4: Integrate it into our Forum Website

We can have a ForumController that prompts the user for the type of user to create

And that's it! We have a working Factory Pattern.

It's pretty straight forward if you break it down right?

forum_user_factory.py



Factory Pattern: Why and When do we use it

- When we want to provide a separate interface for object creation where each subclass can alter the type of object created.
- Adheres to the Single Responsibility, Open Closed, Liskov Substitution and Dependency Inversion Principle.
- Use the Factory Pattern when the exact type and dependencies of the objects you need are unknown or susceptible to change. If we want to add a new product, all we need to do is create a new Factory and override the creation method in it.
- Use this pattern when you want to separate and encapsulate the creation process. This is handy if we want to cache or pick objects from a pool, etc.

Factory Pattern – Disadvantages

- There are a lot of classes in play. This can make the code complex and hard to debug.
- Sometimes the classes can get artificial. You may decide to create a whole new subclass for a very minor change in the object creation process.

Abstract Factory Pattern

WHEN YOU NEED TO CREATE FAMILIES OF OBJECTS INSTEAD OF A SINGLE OBJECT.

Create Families of Products.

The Abstract Factory builds upon the Factory Pattern.

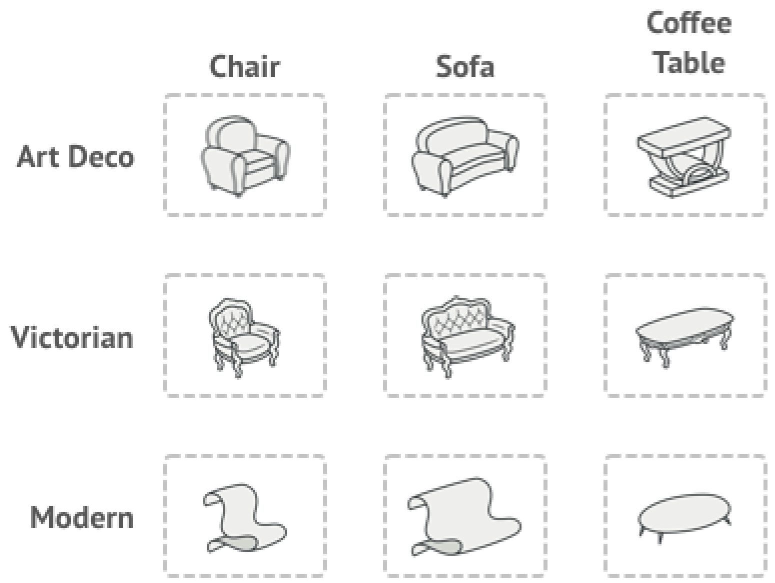
No. Making the base factory abstract is not enough to make a Abstract Factory Pattern.

So here's a new scenario.

Say we have a bunch of objects **that are related to each other** somehow, they make sense together. Let's call this a **"product family"**.

Now say there are **different variations** of these product families.

The Abstract Factory Pattern modifies the Factory Pattern to accommodate this type of complexity.



Product Family (related products):

Chair + Sofa + Coffee

Product Family Variations:

Art Deco, or Victorian, or Modern

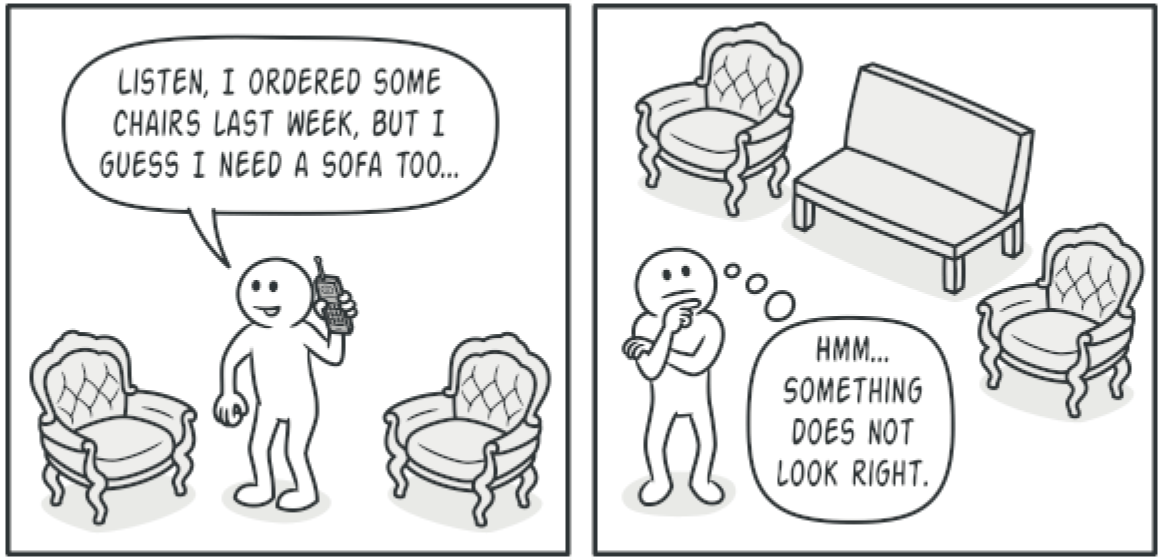
Create Families of Products.

Abstract Factory is a creational design pattern that lets you produce **families of related objects without specifying their concrete classes**.

In a Factory pattern, each factory creates a single product.

In a Abstract Factory pattern, each factory creates a group of products that are **related**.

This is useful if our code **requires objects in specific groups** that are **compatible** with each other.

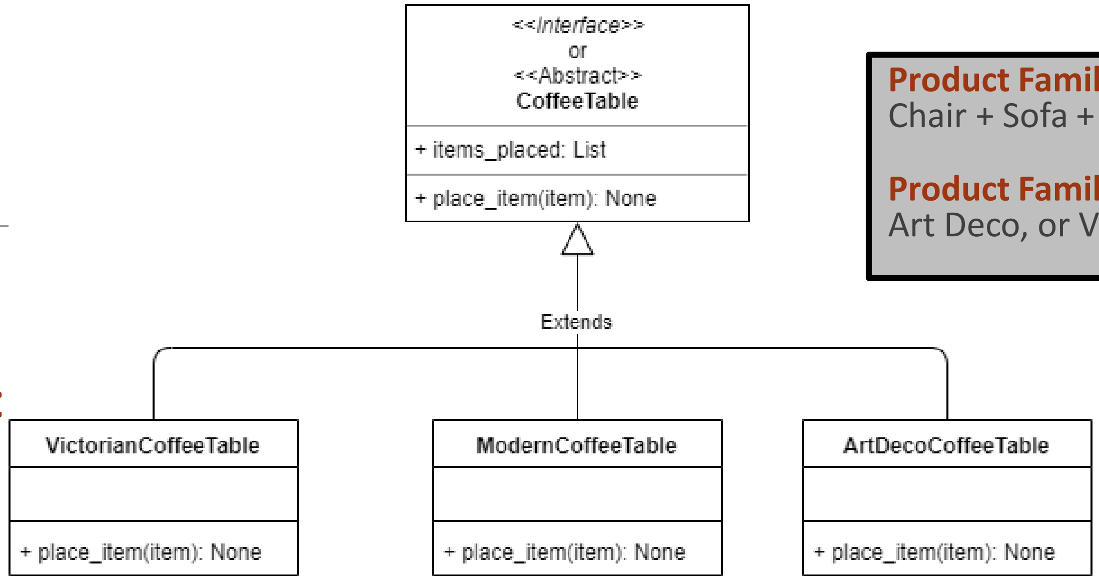

LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

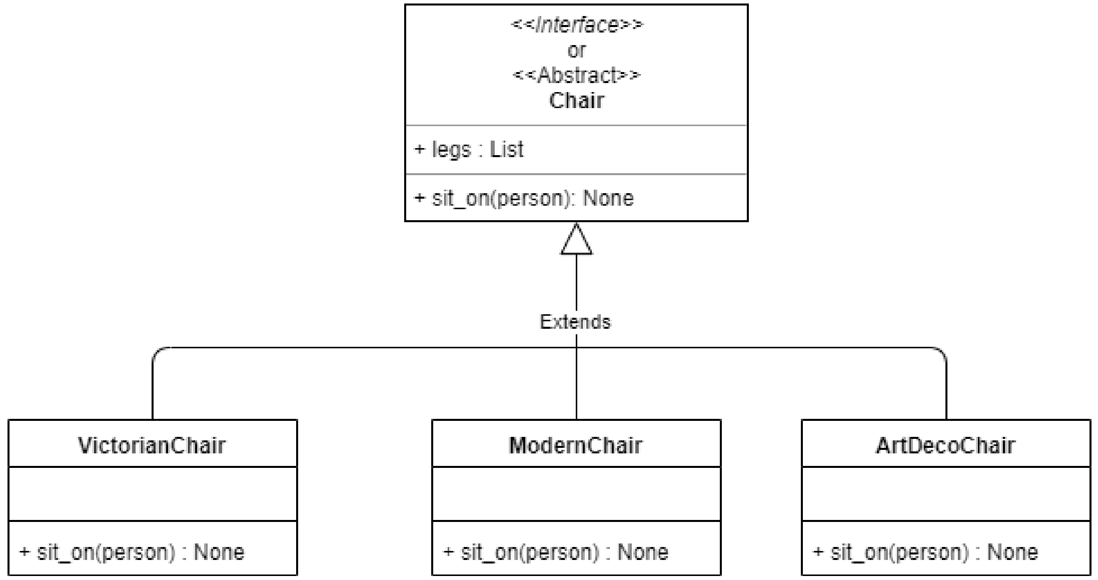
Abstract Factory

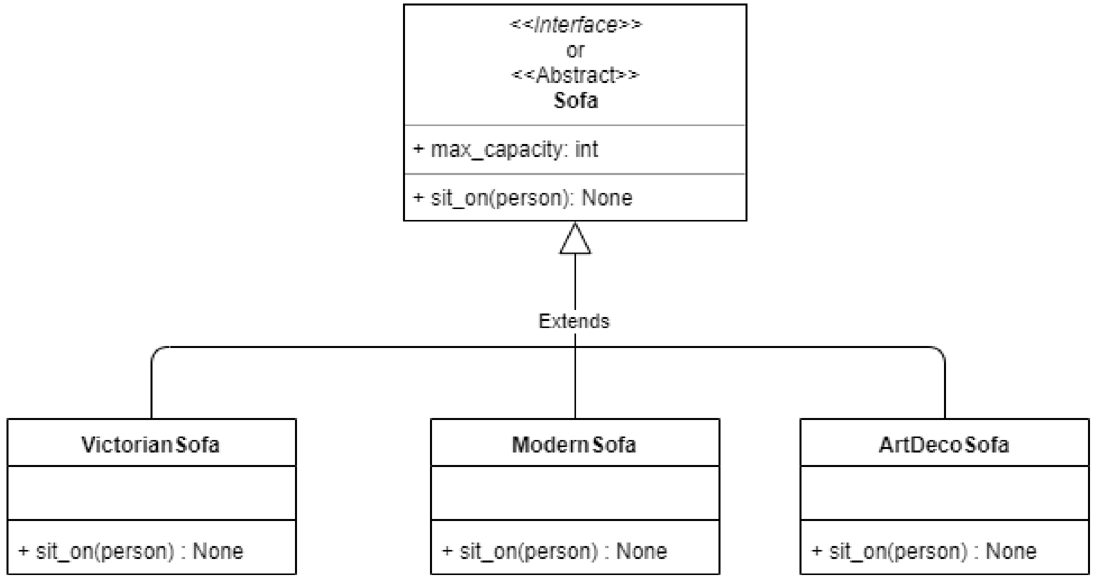
In the abstract factory pattern, instead of one product hierarchy, you would have a **different hierarchy for each distinct product in the product family**.

Product Family (related products):
Chair + Sofa + Coffee

Product Family Variations:
Art Deco, or Victorian, or Modern







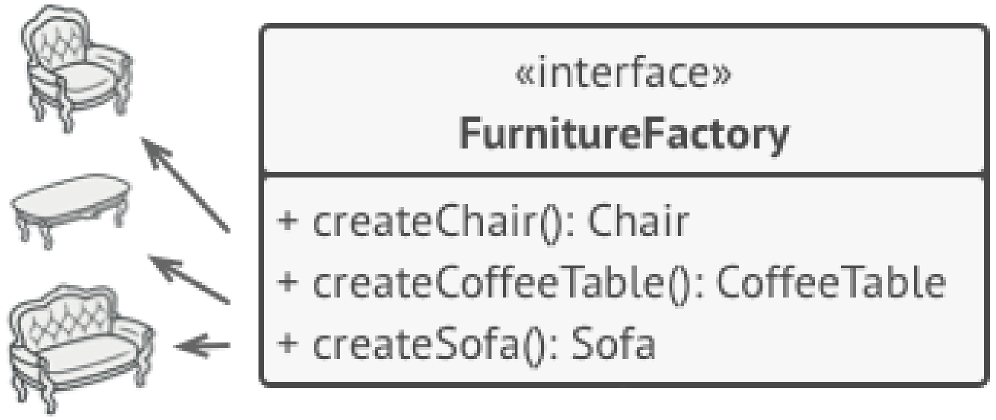
Abstract Factory

Product Family (related products):
Chair + Sofa + Coffee

Product Family Variations:
Art Deco, or Victorian, or Modern

Instead of having a factory that creates one item, we would have **a factory that creates a product family**.

A factory creates an instance of each object in the family and those objects are guaranteed to be compatible.



Abstract Factory

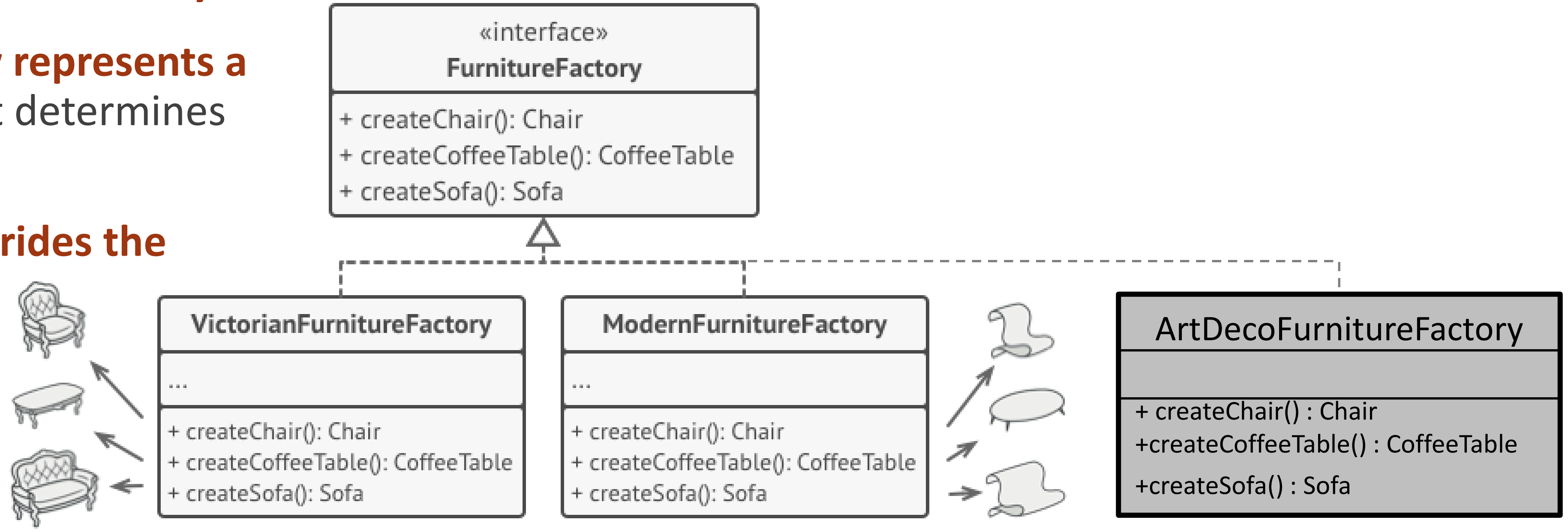
Product Family (related products):
Chair + Sofa + Coffee

Product Family Variations:
Art Deco, or Victorian, or Modern

Just like the factory pattern, we have to extend our Base Factory class (Furniture Factory) and create a **Concrete Factory.**

Each Concrete Factory now represents a theme/variety. This is what determines the **compatibility**.

Each Concrete Factory **overrides the creation methods** to return objects from the **same Product Family.**



Abstract Factory

Product Family (related products):
Chair + Sofa + Coffee

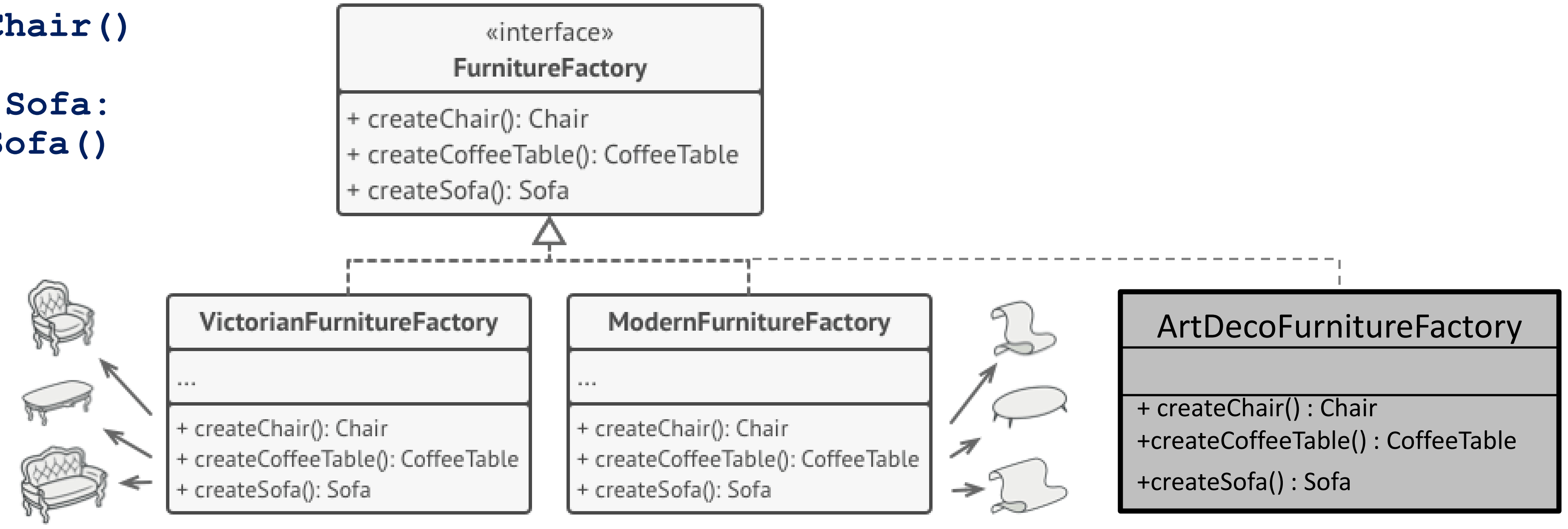
Product Family Variations:
Art Deco, or Victorian, or Modern

```
Class VictorianFurnitureFactory(FurnitureFactory):

    def create_coffee_table() -> CoffeeTable:
        return VictorianCoffeeTable()

    def create_chair() -> Chair:
        return VictorianChair()

    def create_sofa() -> Sofa:
        return VictorianSofa()
```



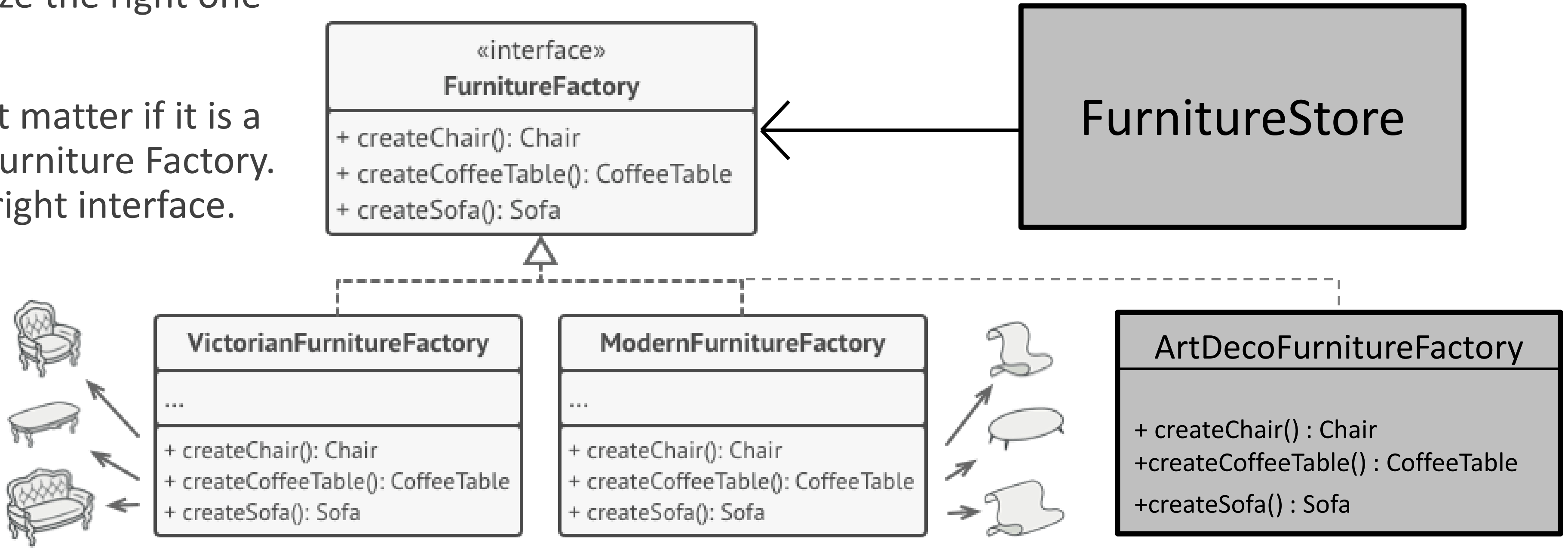
Abstract Factory

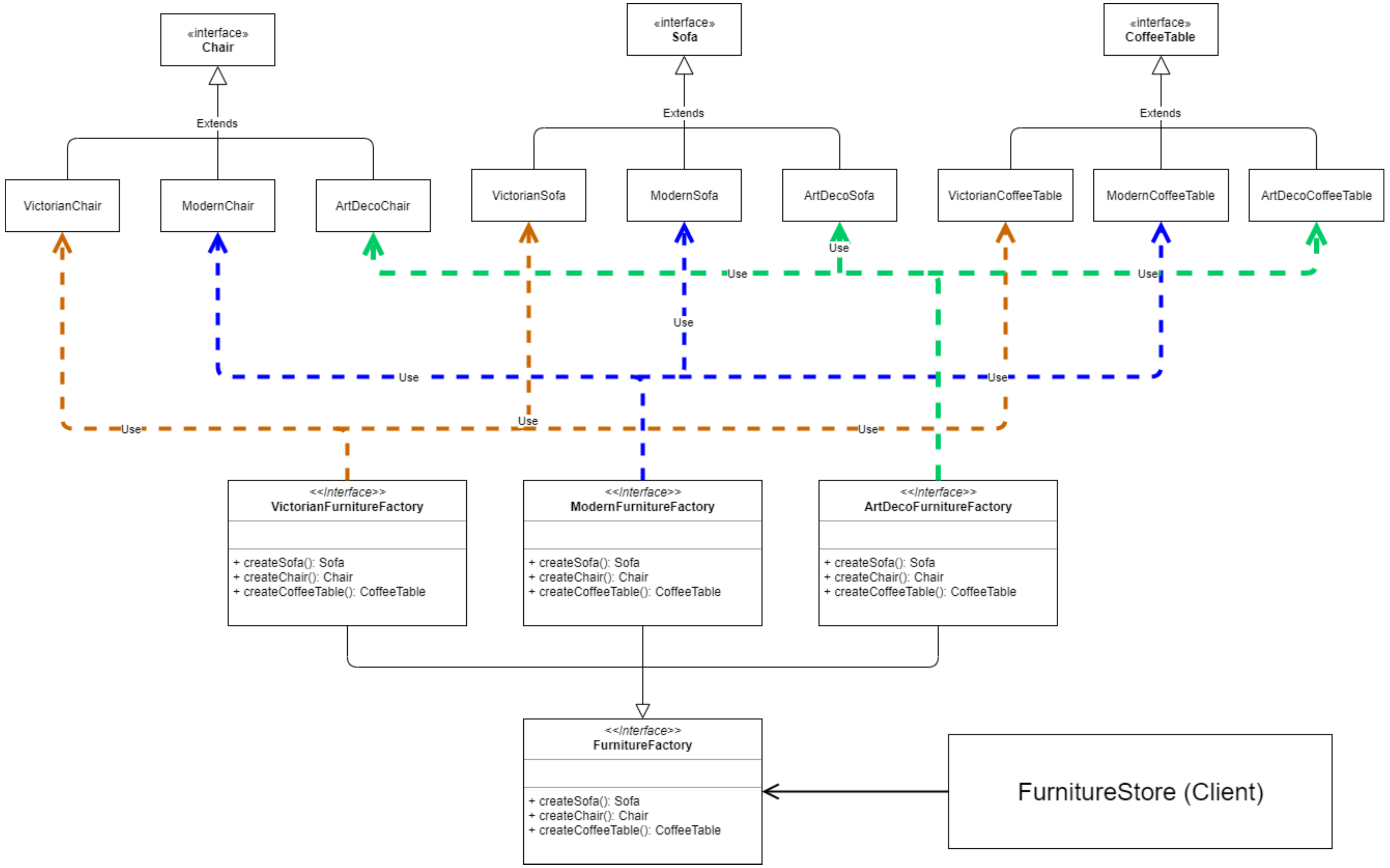
Now we just need to add in the client.

The client expects an object with FurnitureFactory Interface.

We can have a controller initialize the right one and pass it on to the client.

Once with the client it shouldn't matter if it is a Victorian, Modern or ArtDeco Furniture Factory. What matters is that it has the right interface.



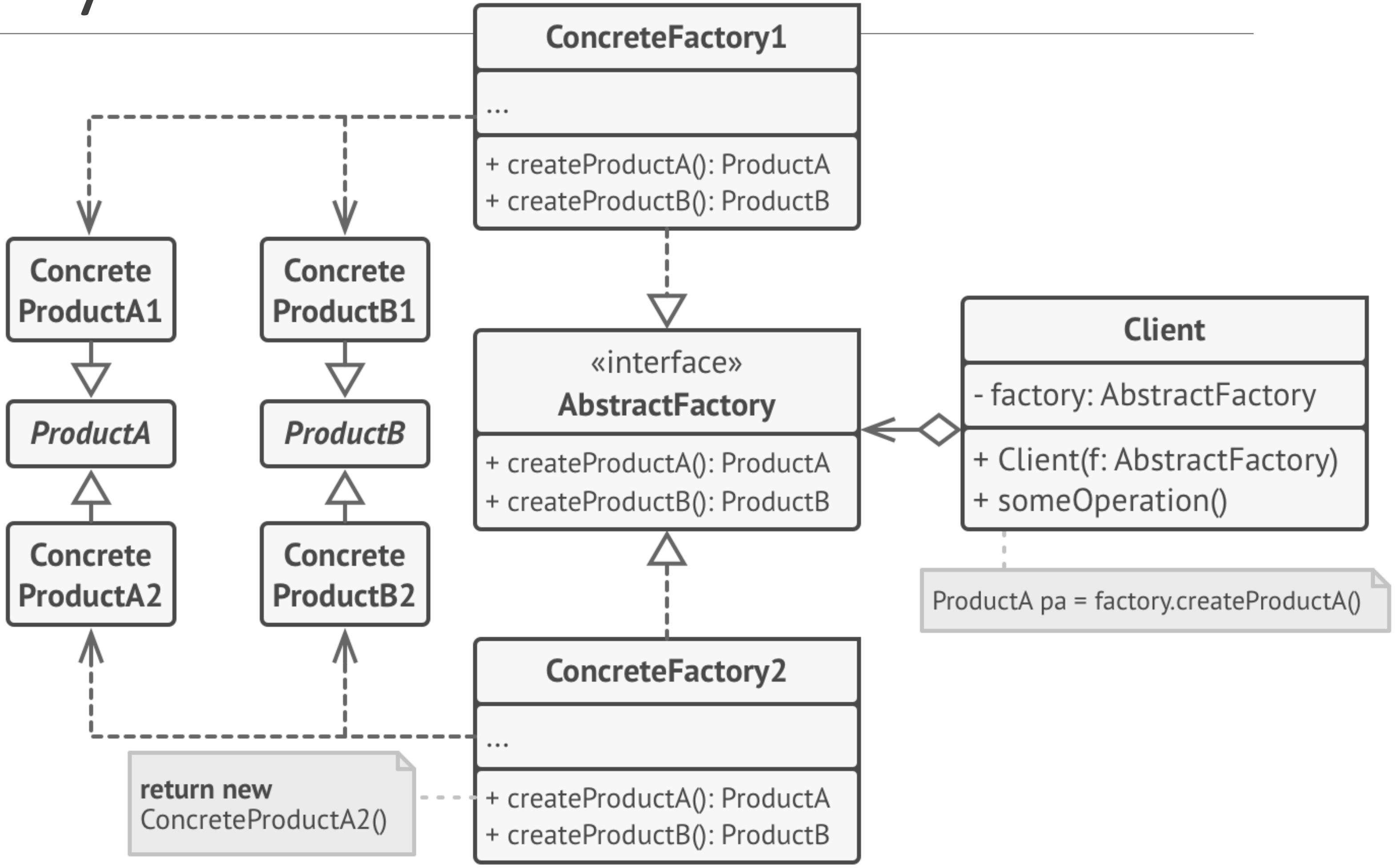


Abstract Factory – Putting it all together

Abstract Factory

It's essentially very similar to the Factory Pattern.

It just looks complicated since we are dealing with more classes due to the large number of products.


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Abstract Factory – Let’s Make one!

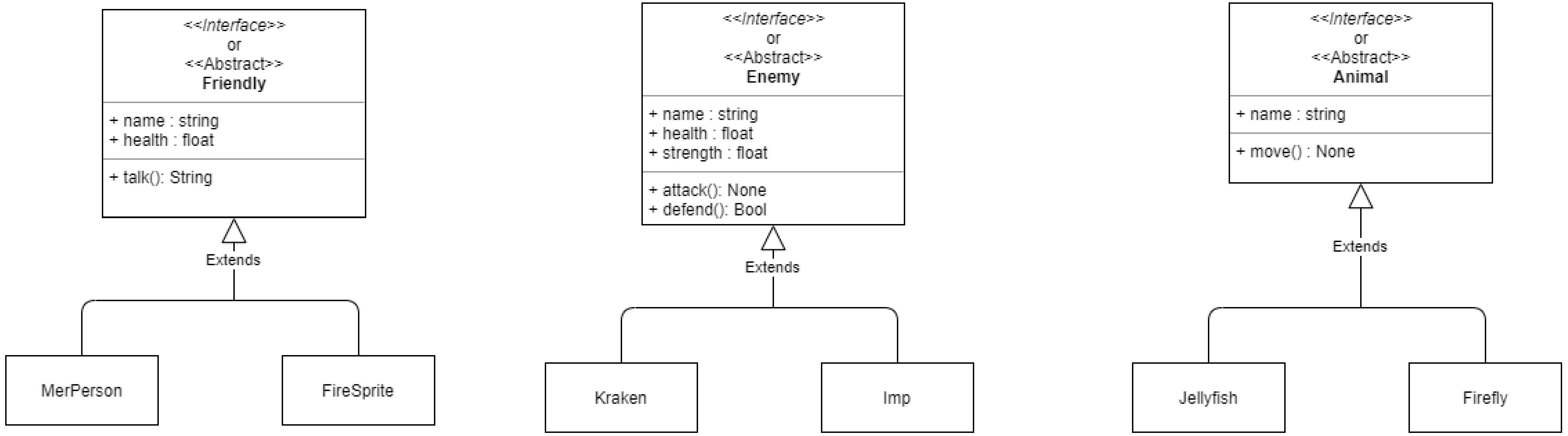
Say we were working on a game that wanted to spawn groups of characters (Friendlies, Enemy and Animals) in different worlds.

Each world would have characters following a various theme.

The first step is to create a matrix identifying our **Product Families** (groups of related objects) in one dimension, and **varieties** (themes) in another dimensions.

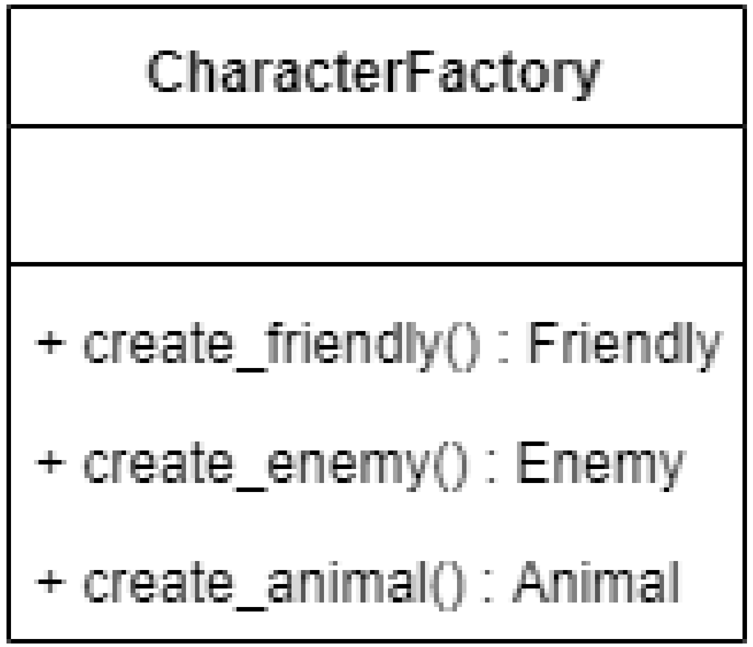
Theme/Product	Friendlies	Enemies	Animals
Aquatica	MerPerson	Kraken	Jellyfish
Firelands	FireSprite	Imp	Firefly

Step 1:For each product in the product family, define its inheritance hierarchy.

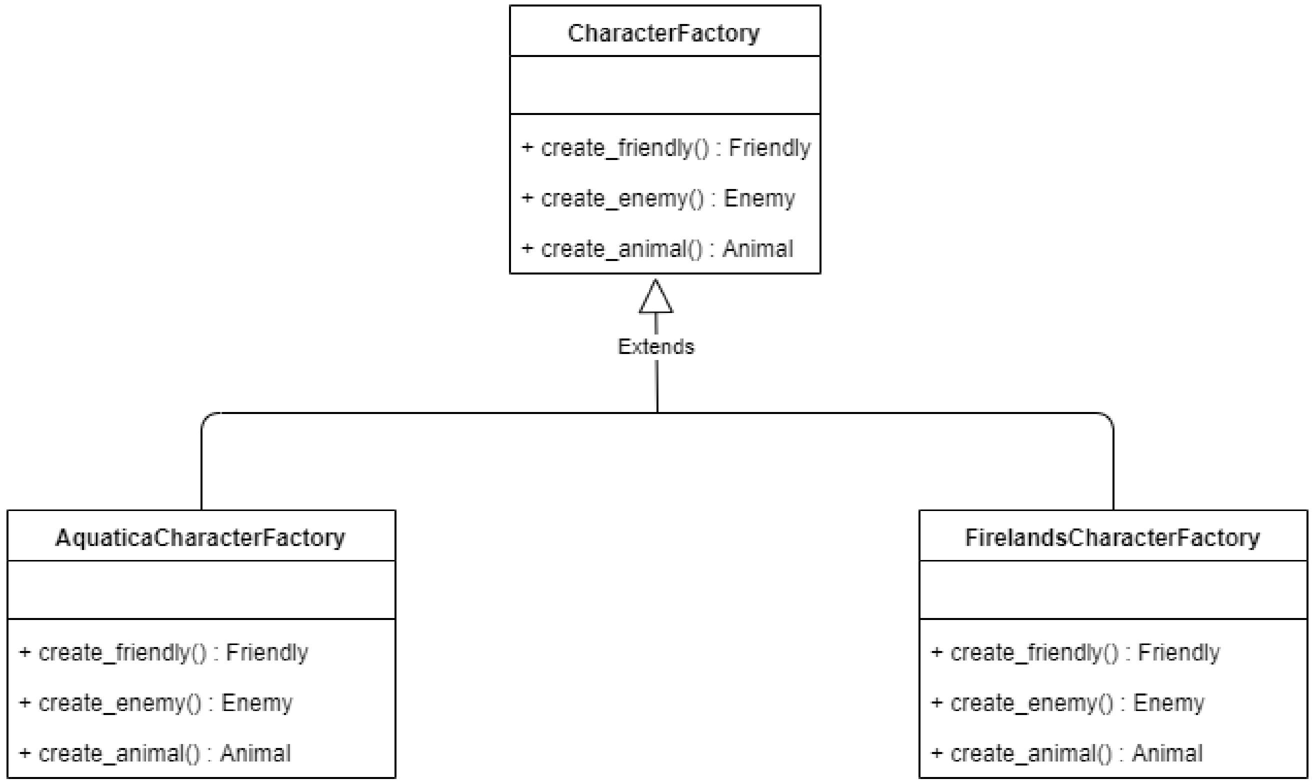


Theme/Product	Friendlies	Enemies	Animals
Aquatica	MerPerson	Kraken	Jellyfish
Firelands	FireSprite	Imp	Firefly

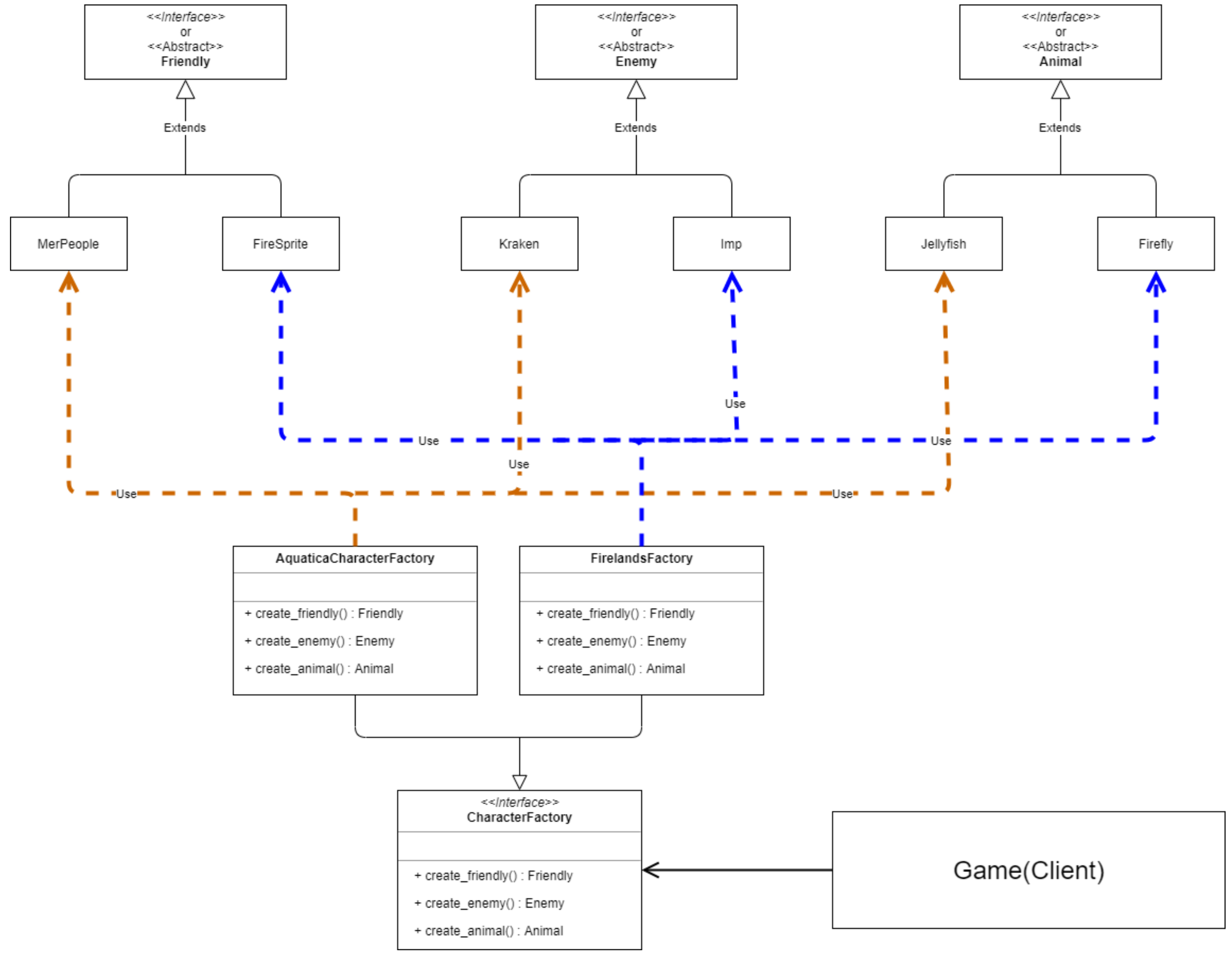
Step 2: Define a Factory base class that can create a Product Family



Step 3: Extend the Factory base class with Concrete Factories that can create Varieties.


CharacterFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal
Extends
AquaticaCharacterFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal
FirelandsCharacterFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal

Step 4: Put it all together



<<Interface>> or <<Abstract>> Friendly
<<Interface>> or <<Abstract>> Enemy
<<Interface>> or <<Abstract>> Animal
Extends
Extends
Extends
MerPeople
FireSprite
Kraken
Imp
Jellyfish
Firefly
Use
Use
Use
Use
Use
Use
AquaticaCharacterFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal
FirelandsFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal
<<Interface>> CharacterFactory
+ create_friendly() : Friendly
+ create_enemy() : Enemy
+ create_animal() : Animal
Game(Client)

Game_abstract_factory.py

Factory Pattern:
Why and When do we use it

- Use this pattern if the code needs to work with families of related objects.
- Single Responsibility, Open/Closed Principle, Liskov Substitution and Dependency Inversion Principle.
- If the exact number of Product Families are unknown and you will have to add in new ones in the future.

Factory Pattern – Disadvantages

- There are a lot of classes and interfaces in play. This can make the code complex and hard to debug.
- Sometimes the classes can get artificial. You may decide to create a whole new subclass for a very minor change in the object creation process.

Recap: Factory and Abstract Factory

Factory (Also known as Factory Method)

- When you want to separate creation code from client code.
- When you want a client or service to only depend on a base class for object creation and not each concrete class.

Abstract Factory

- A more evolved form of the Factory Pattern
- Allows for the creation of a group of related objects .
- When you want to create different variants of families of objects.

That’s it for today!

Quiz on Friday

Material from last week and week 6