Abstract Factory, Builder, Chain of Responsibility, Lazy Initialization

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 11

Abstract Factory Pattern

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

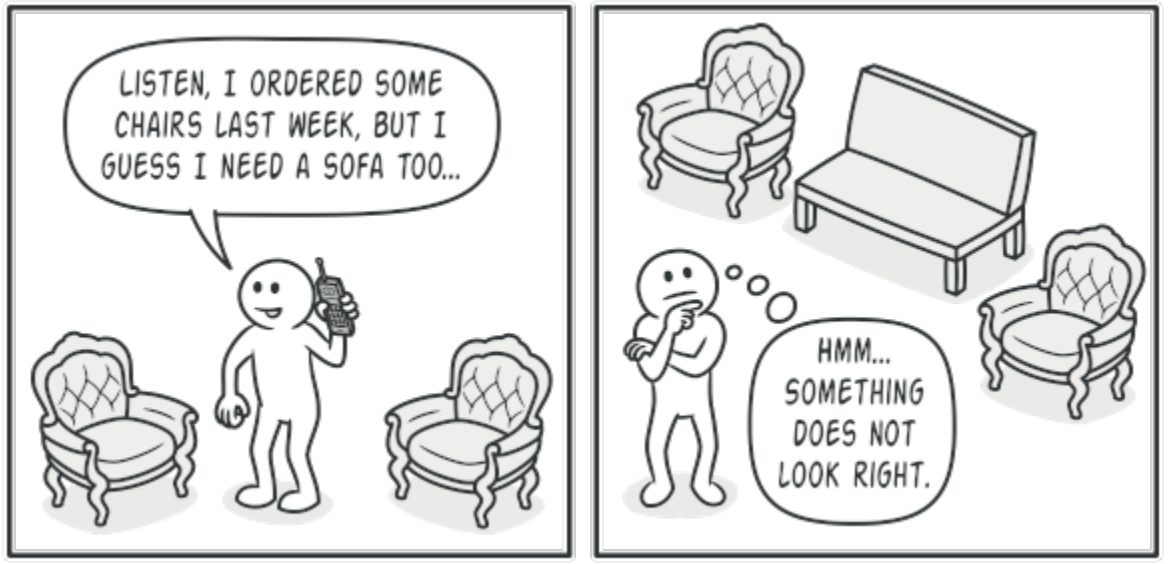
Create Families of Products.

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 an Abstract Factory pattern, each factory creates a **group of related products**.

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



Create Families of Products.

The Abstract Factory builds upon the Factory Pattern.

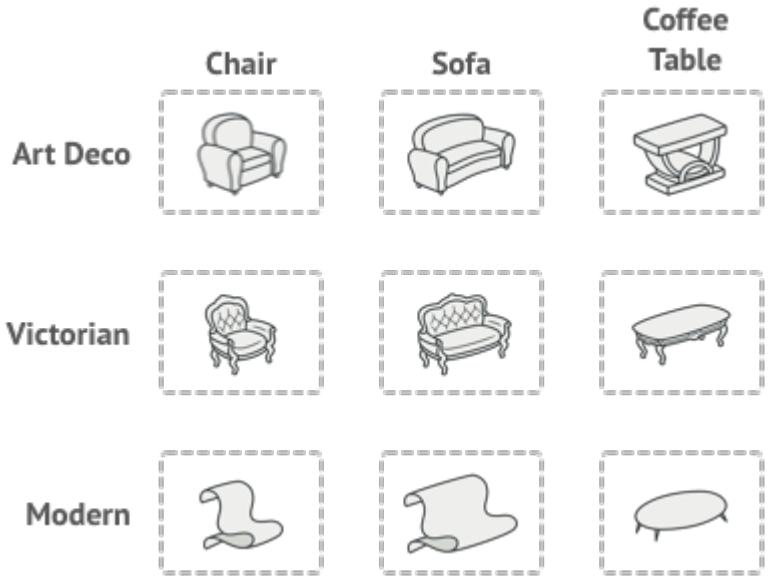
No. Making the base factory abstract is not enough to make an Abstract Factory Pattern.

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 + CoffeeTable

Product Family Variations:

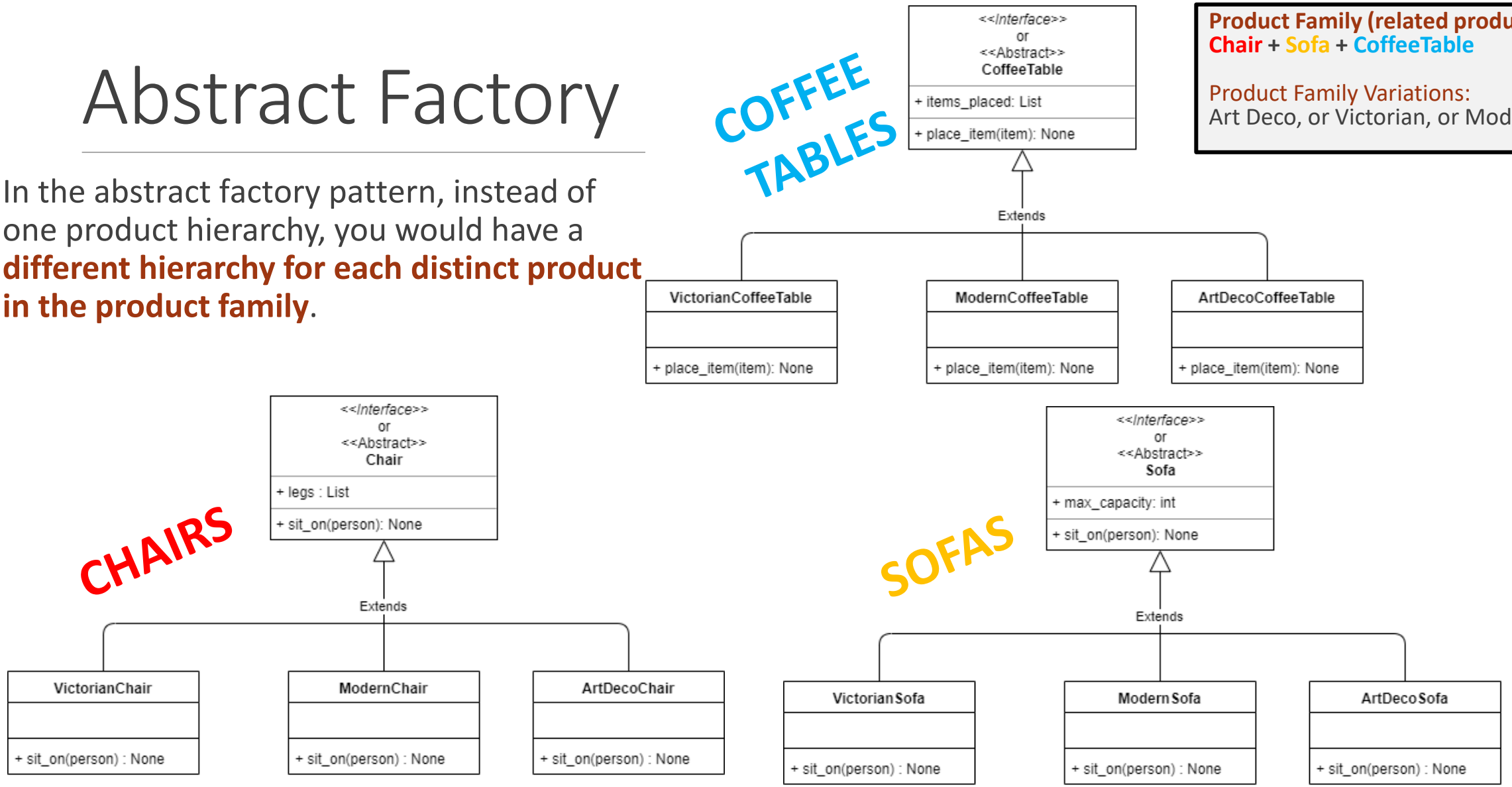
Art Deco, or Victorian, or Modern

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** + **CoffeeTable**

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



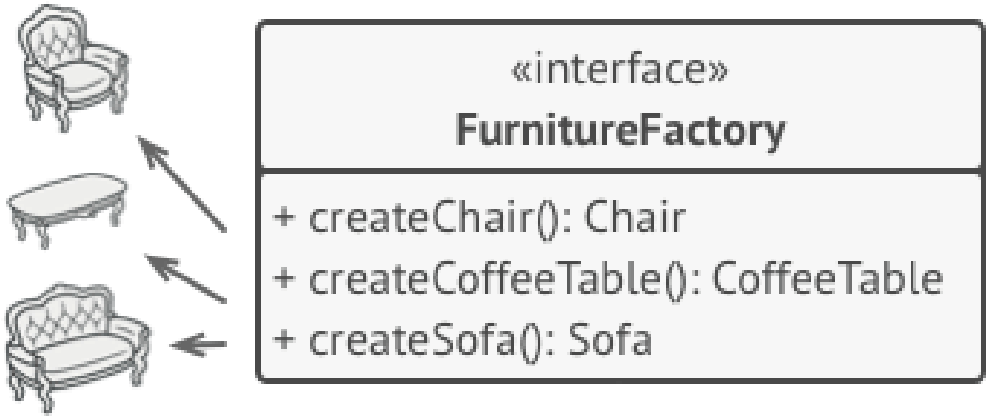
Abstract Factory

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

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

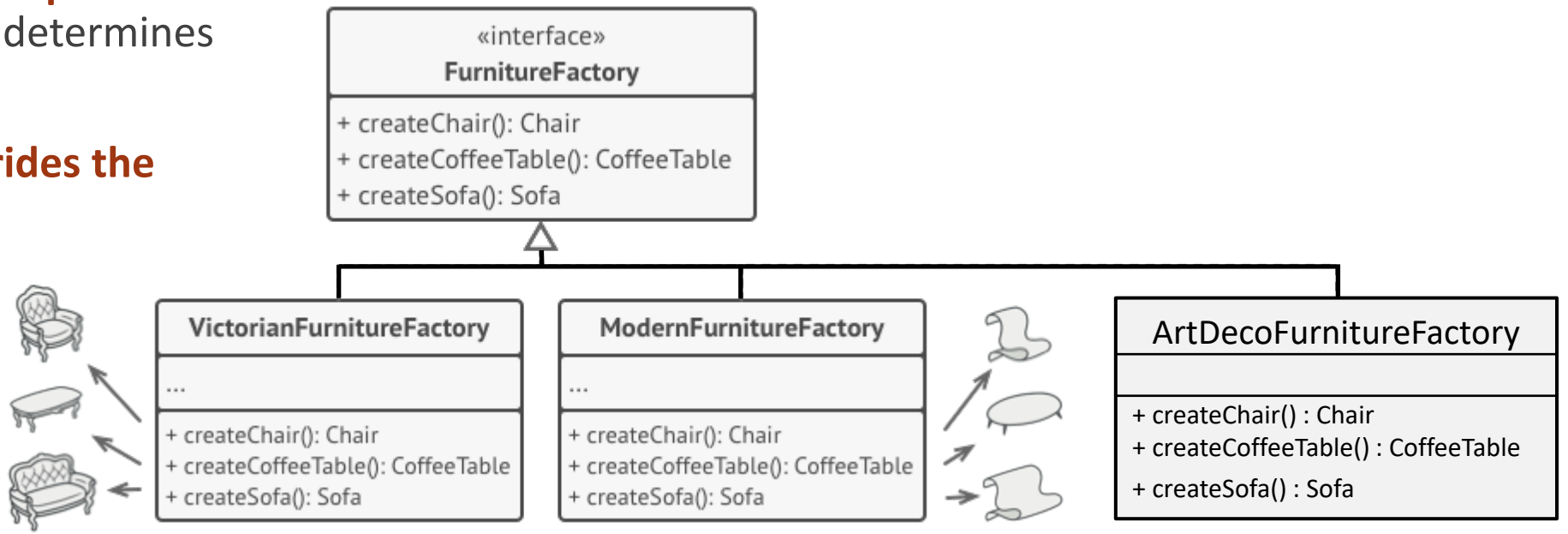
Product Family (related products):
Chair + Sofa + CoffeeTable

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.**

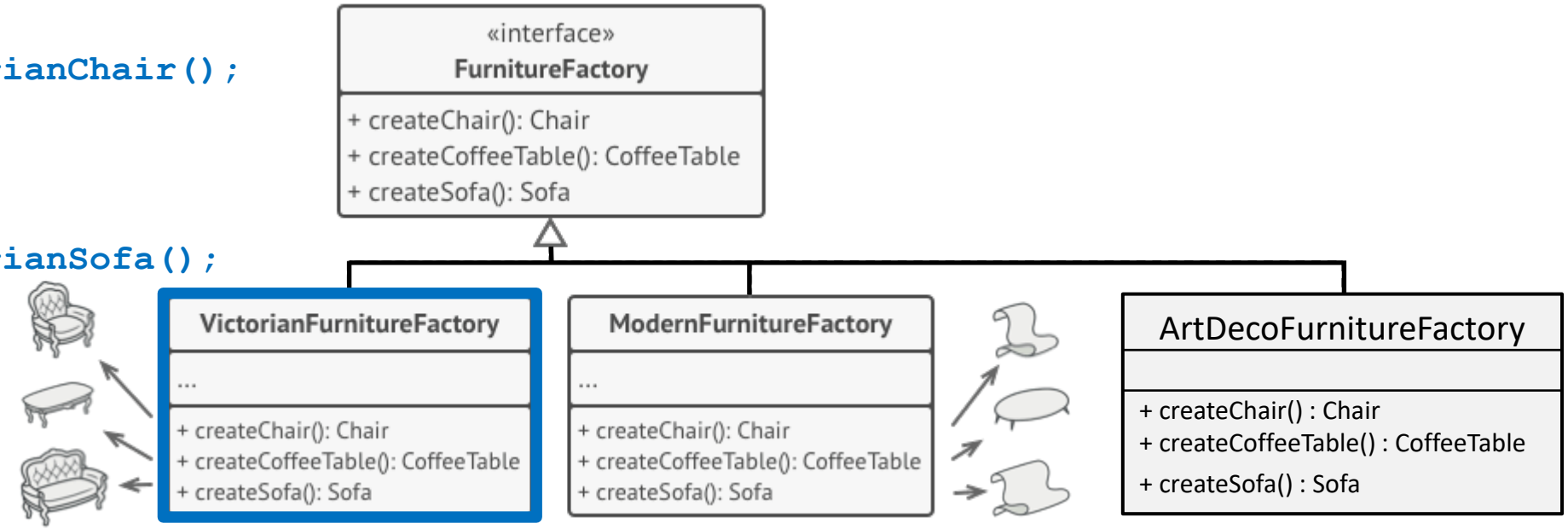


Abstract Factory

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

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

```
class VictorianFurnitureFactory : public FurnitureFactory {

  CoffeeTable* create_coffee_table() {
        return new VictorianCoffeeTable();

   }

  Chair* create_chair() {
        return new VictorianChair();

   }

  Sofa* create_sofa() {
        return new VictorianSofa();

   }

}
```



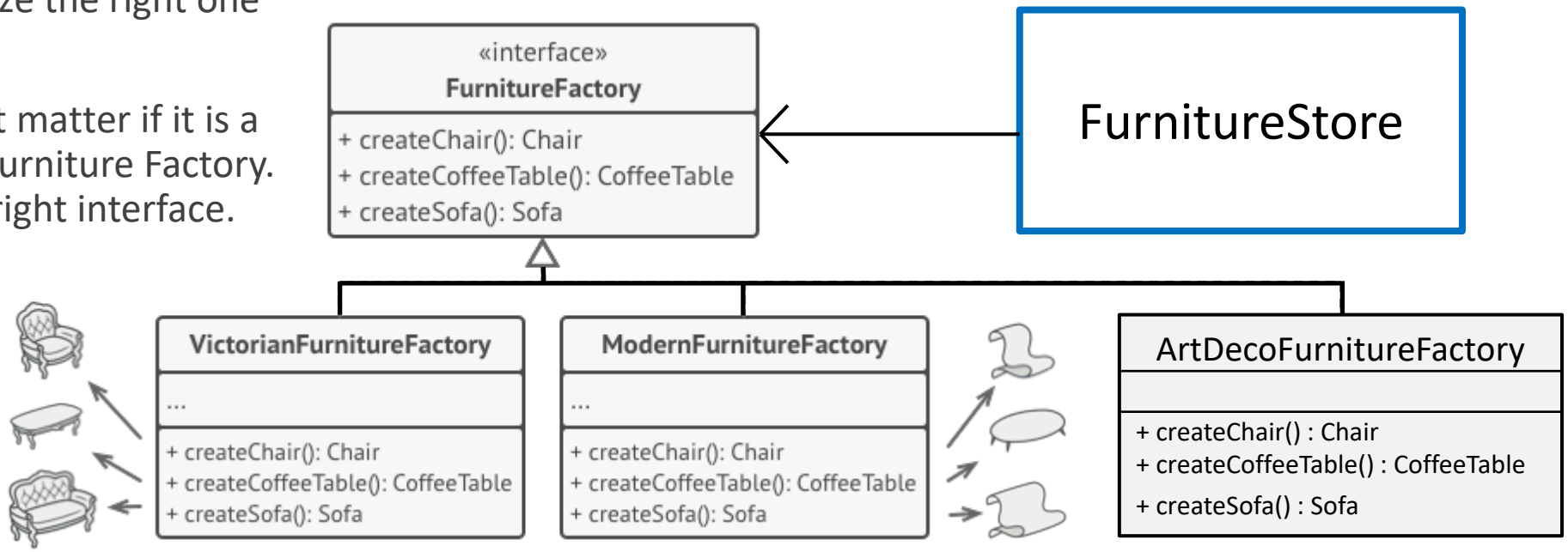
Abstract Factory

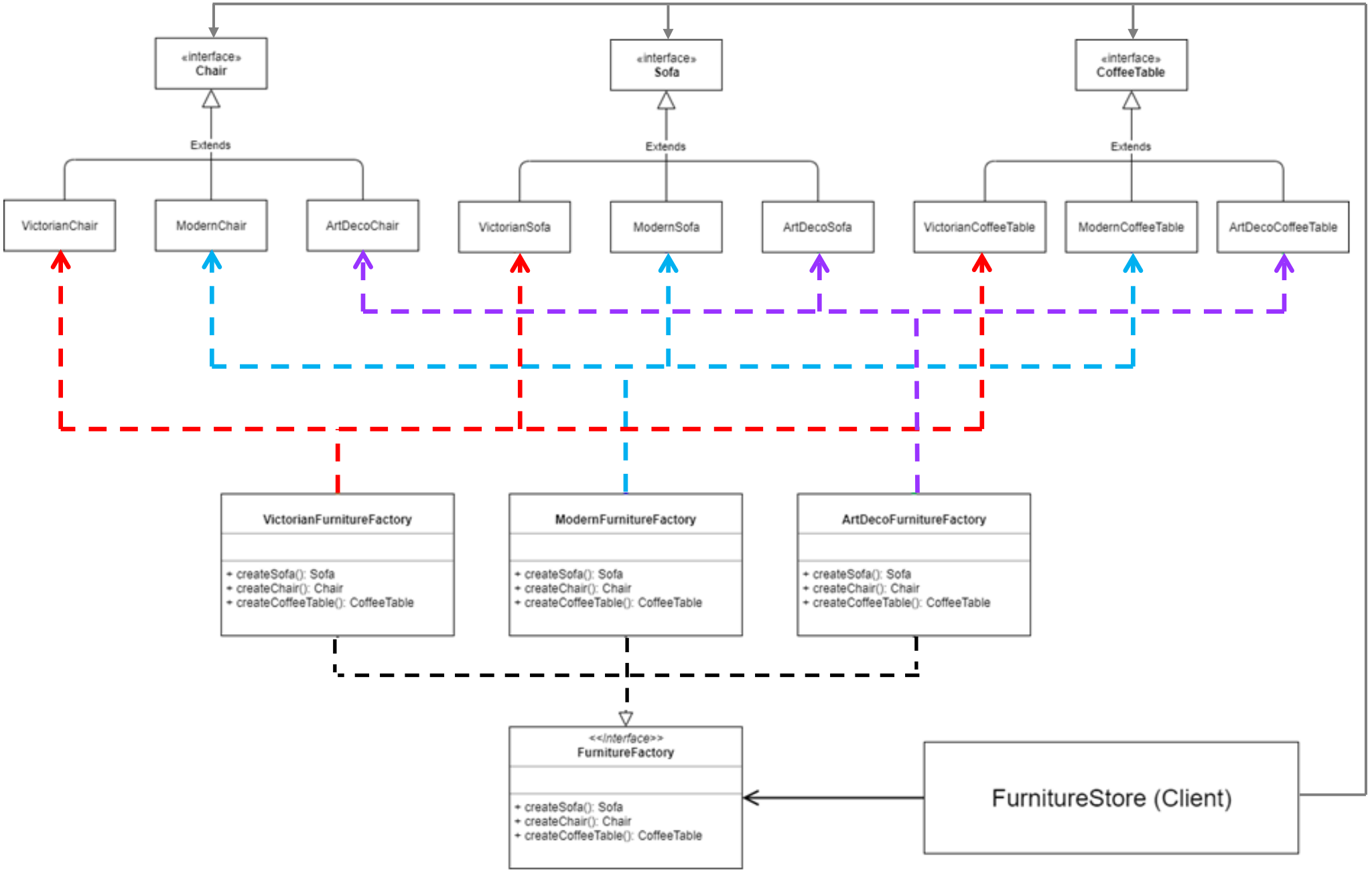
Now we just need to add in the **client**.

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.



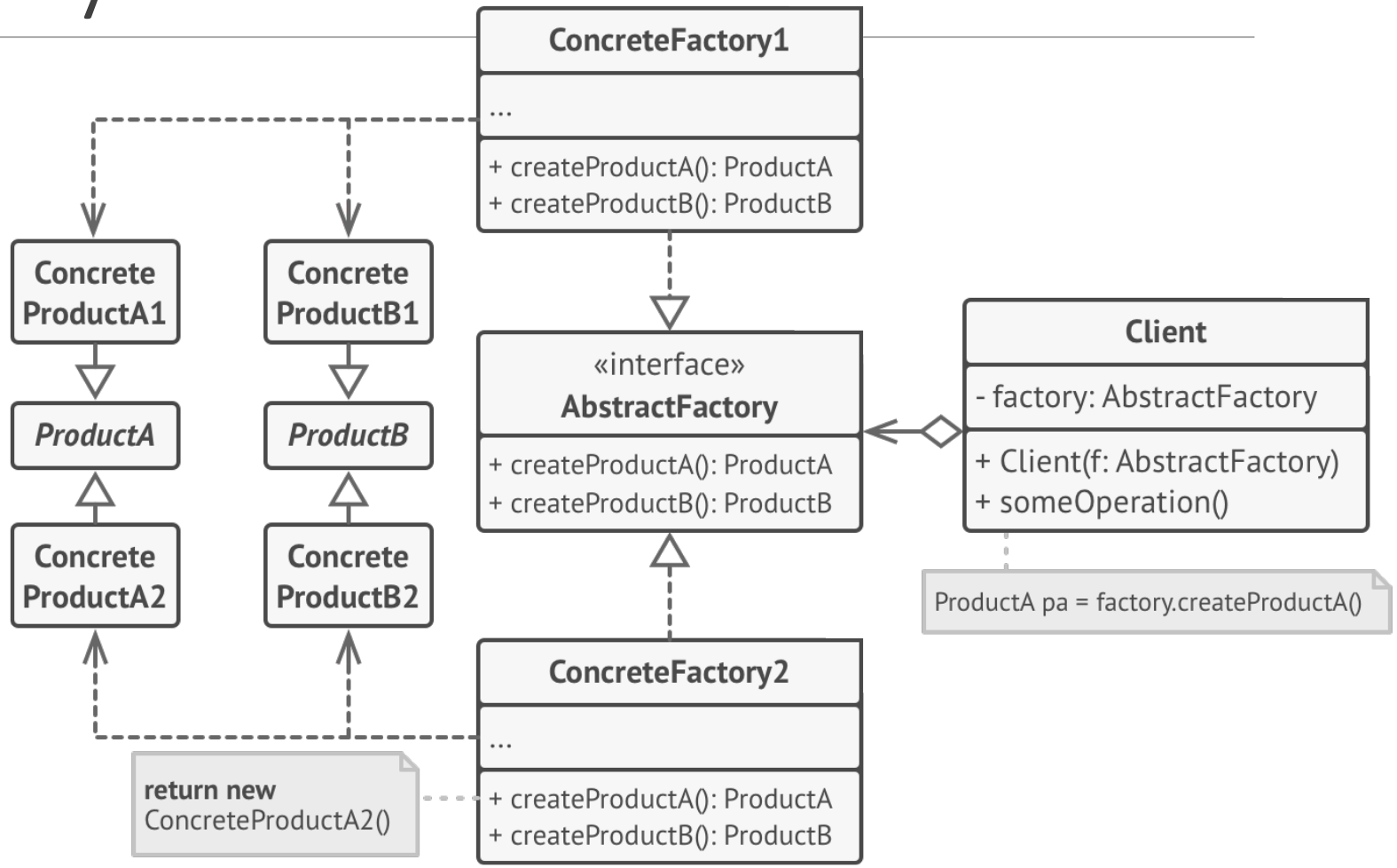


Abstract Factory – Putting it all together

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.



Abstract Factory – Let's Make one!

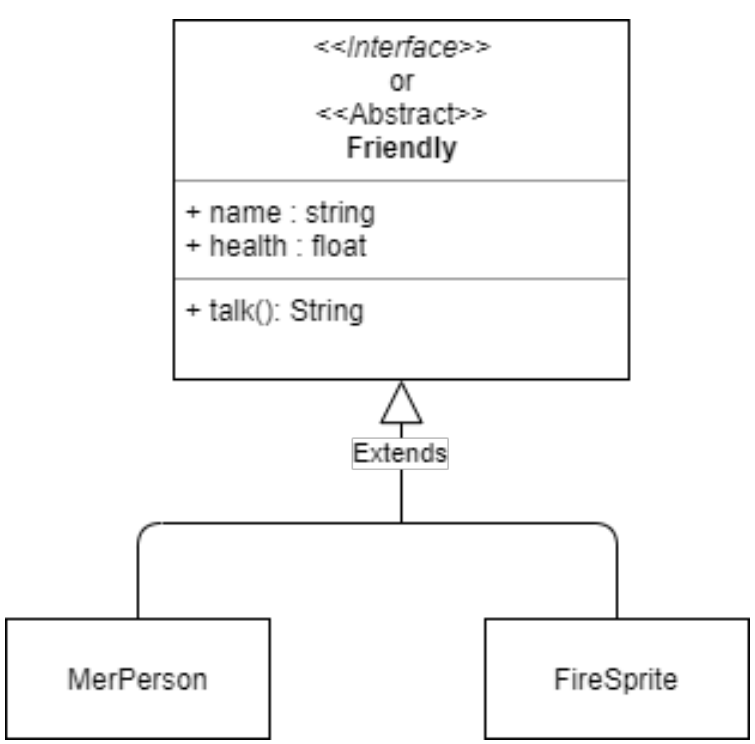
Say we were working on a game that wanted to spawn groups of characters (Friendlies, Enemy and Animals) in different worlds.

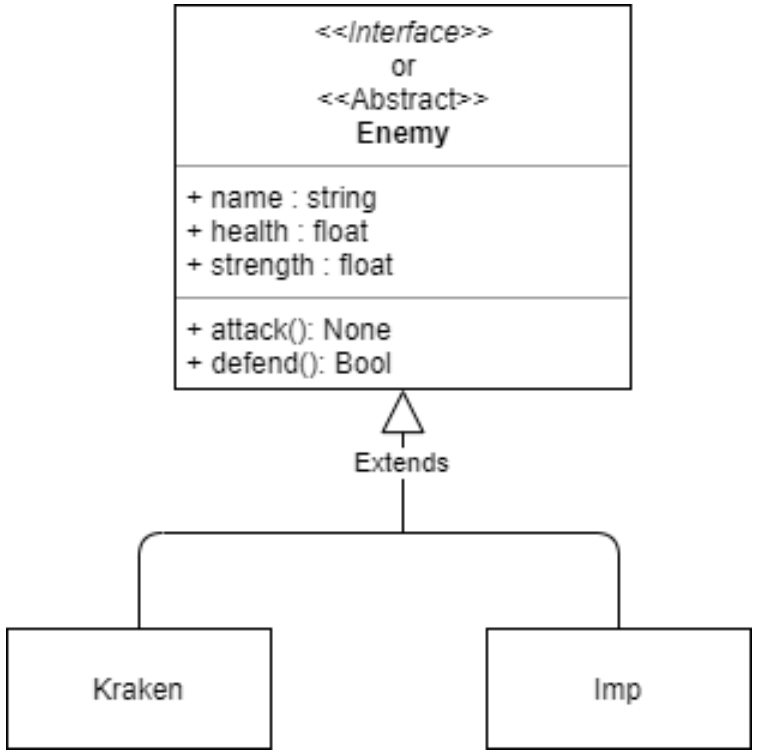
Each world would have characters following various themes.

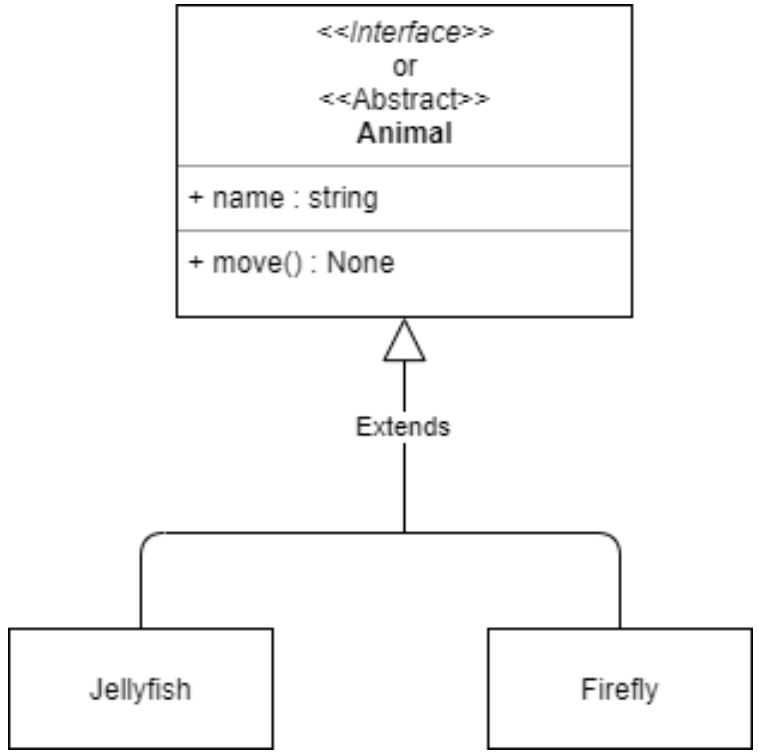
The first step is to create a matrix identifying our **Product Families** (groups of related objects) in one dimension, and **varieties** (themes) in another dimension.

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.

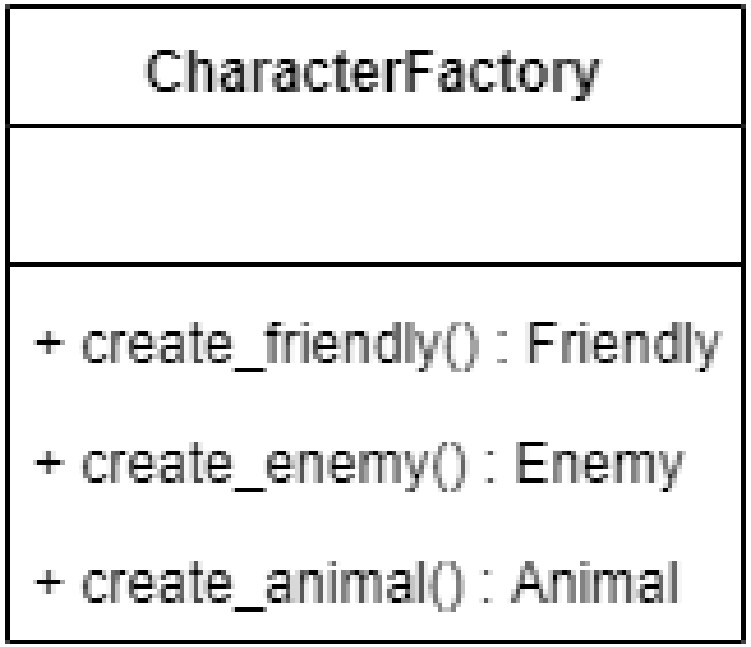




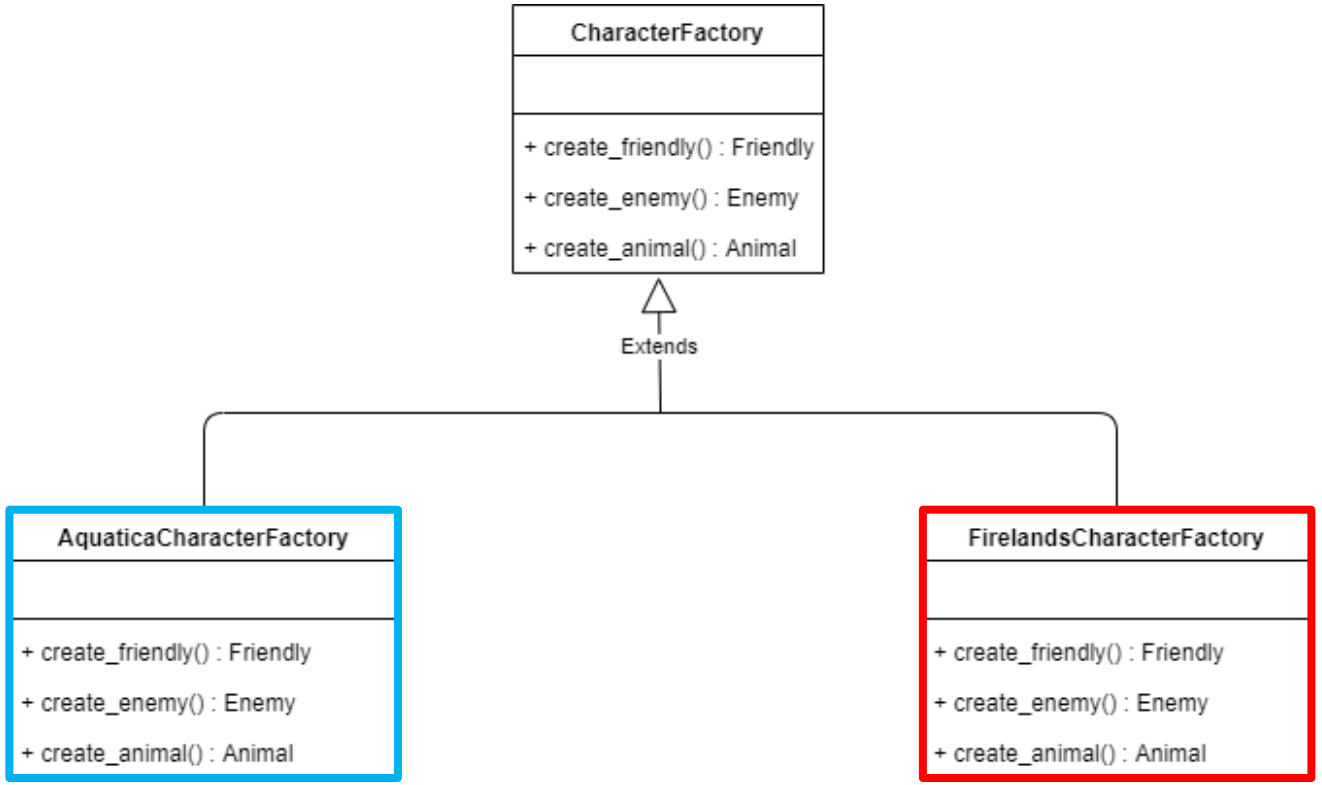


Theme/Product	Friendlies	Enemies	Animals
Aquatica	MerPerson	Kraken	Jellyfish
Firelands	FireSprite	Imp	Firefly

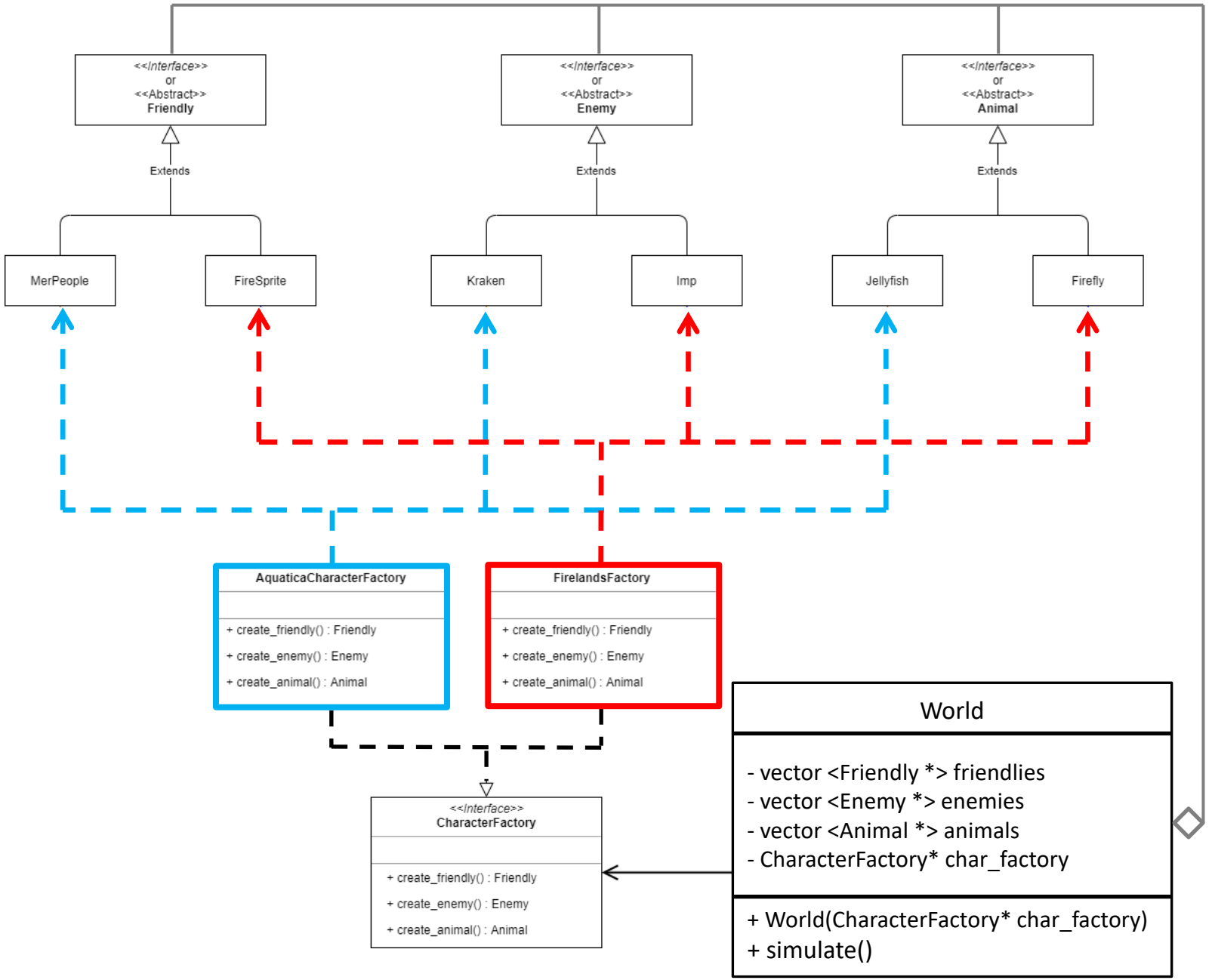
Step 2: Define a Factory base class that can create a Product Family



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



Step 4: Put it all together



Game_abstract_factory.cpp

Abstract 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 have to add new ones in the future.

Abstract 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.

Builder

WHEN YOU WANT CUSTOMIZATION AND COMPLEXITY

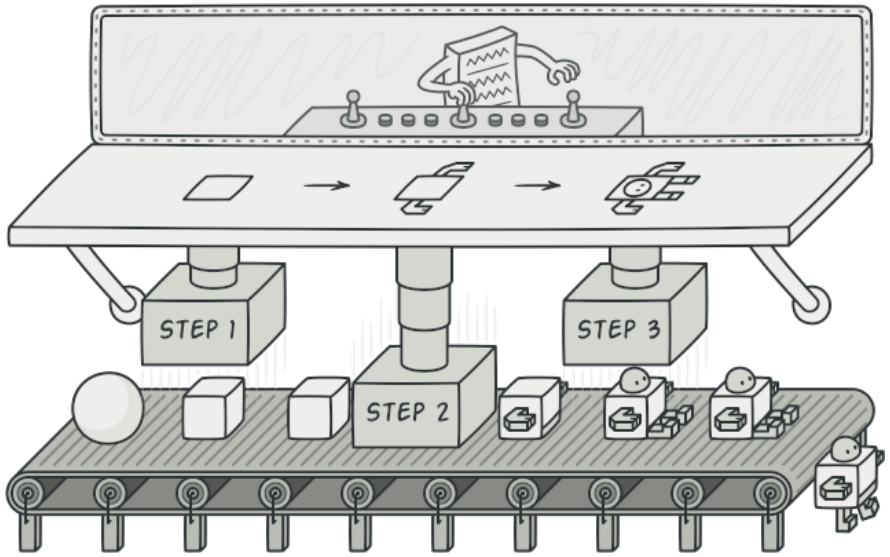
Builder

Often we need to initialize objects with a number of different parts, nested fields, **optional attributes** and behaviour.

This can cause either:

- The **constructor of an object to become bloated**. **Lots of parameters**, most of which may not even be used often.
  - `MyClass(param1, param2, param3, param4, etc…)`
- A complex inheritance hierarchy to model the optional features.

The Builder Pattern allows us to **break this construction code into a separate class** that decouples independent blocks of code that can be mixed and matched.



Builder

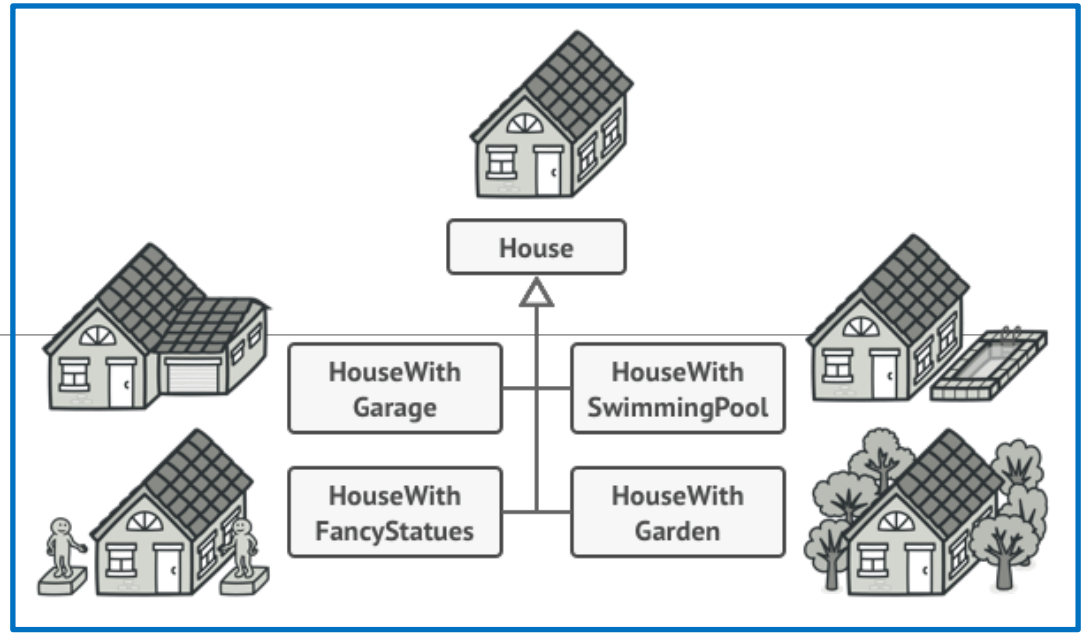
Say we were modeling and representing Houses.

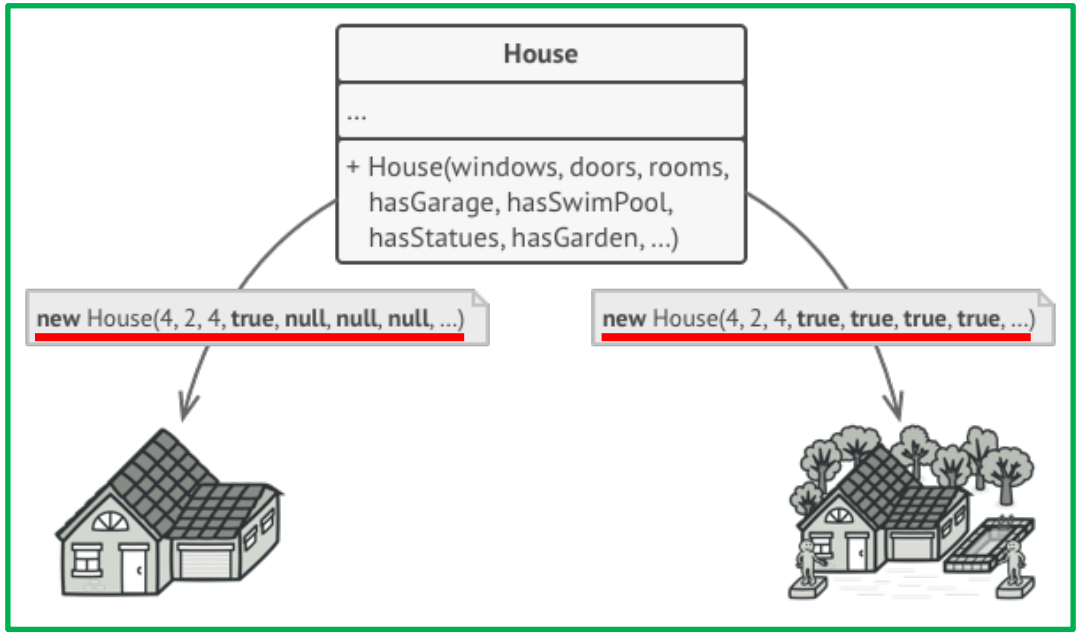
Houses have a lot of components and optional parts.

- Door, walls, roof, windows, etc.

We can use **inheritance** to model all these cases.

Or we can avoid inheritance and have a **single class with a lot of parameters** in its constructor to track all the components. This leads to **ugly constructor calls**.





Builder

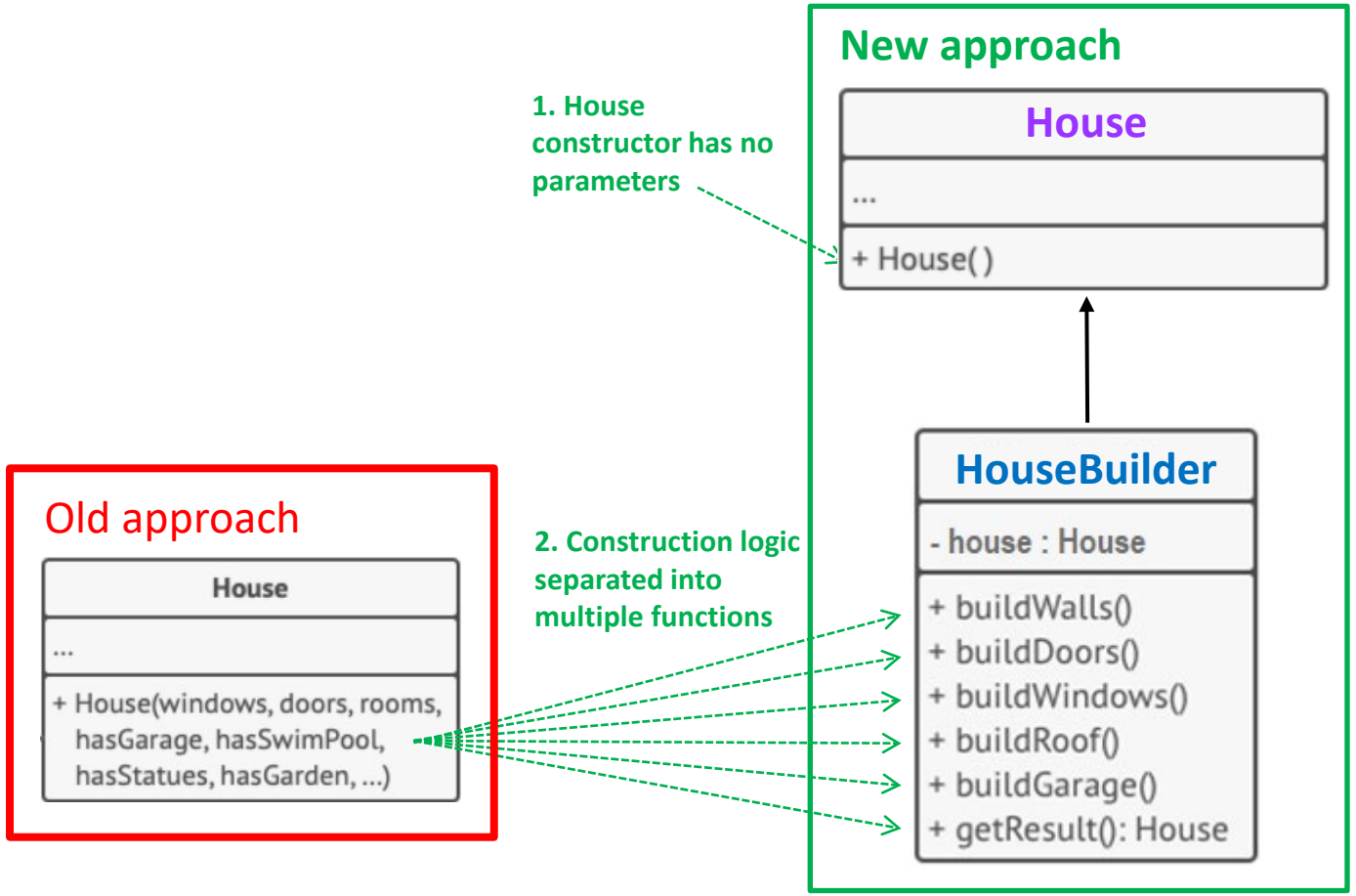
If the construction of an object is so complex, let's separate that out into its own class (**HouseBuilder**).

This would ensure the **Single Responsibility Principle**.

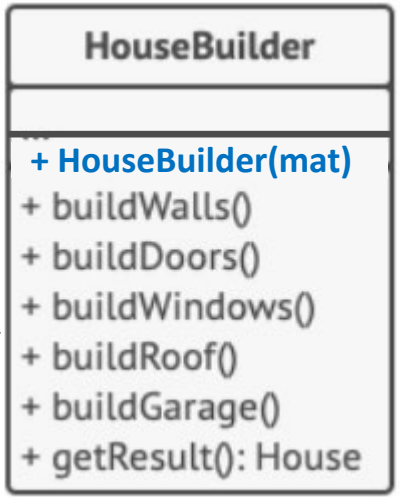
A class would be split into itself (**House**) and its builder class (**HouseBuilder**) with all of its construction code.

We can have a separate call for each different component that our object is aggregated/composed off.

Calling different combinations of these methods would lead to different types of homes.



Builders and Variations



This is a really powerful pattern.

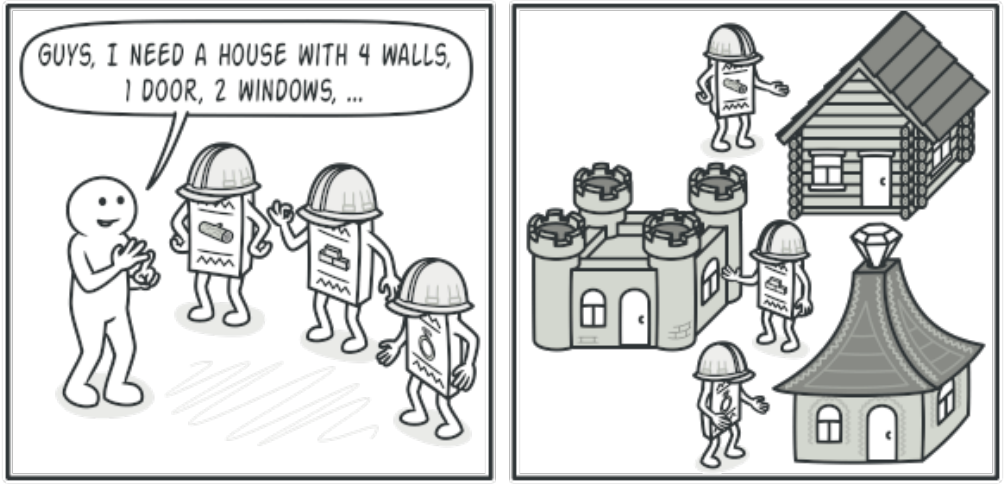
We can customize the Builder Class to take in its **own parameters and options**.

This can allow us to instantiate different Builder objects that execute the construction differently.

For example, we can have our builder class accept a parameter for the material we can build the house out of.

We can then make 3 builder objects:

```
HouseBuilder* builder_wood = new HouseBuilder("Wood");
House* cabin = builder_wood->build_house();

HouseBuilder* builder_stone = new HouseBuilder("Stone");
House* castle = builder_stone->build_house();

HouseBuilder* builder_paper = new Housebuilder("Paper");
House* origami_house = builder_paper->build_house();
```



Builder and Director

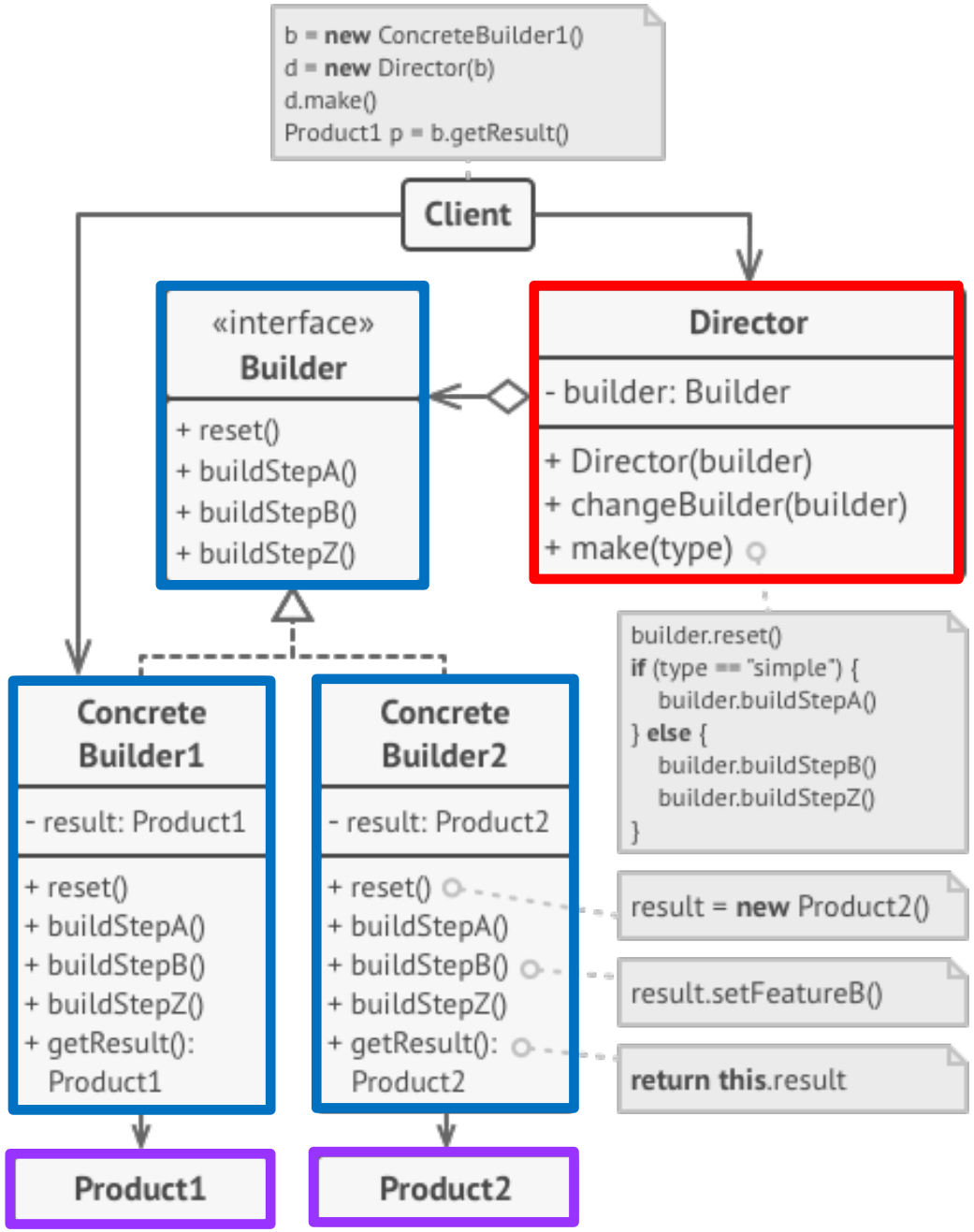
We can **optionally** implement a manager class called a **Director** that can be responsible for different subroutines that call different methods in a builder class.

This Director class is optional and usually makes sense when working with complex objects that can be decoupled into different pieces/components that can be mixed and matched.

We **can even inherit** from a Builder class to create different variations of builders if the complexity demands it.

```
struct Director {
    Builder* builder;
    Director(Builder* builder) : builder(builder) {}

    House* build_townhouse() {
        builder->build_walls();
        builder->build_roof();
        builder->build_garage();
        builder->build_swimming_pool();
        builder->build_second_story();
        builder->build_door();
        builder->build_windows();
        return builder->getProduct();
    }

    House* build_apartment() {
        builder->build_walls();
        builder->build_door();
        builder->build_windows();
        return builder->getProduct();
    }
};
```

Builder Pattern: UML

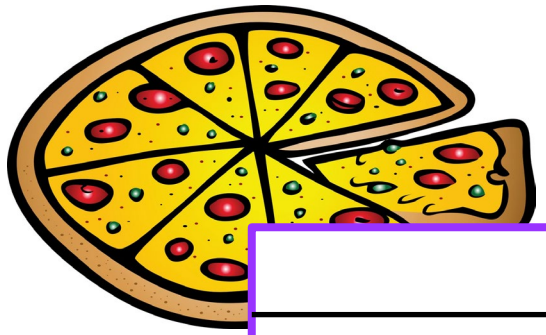


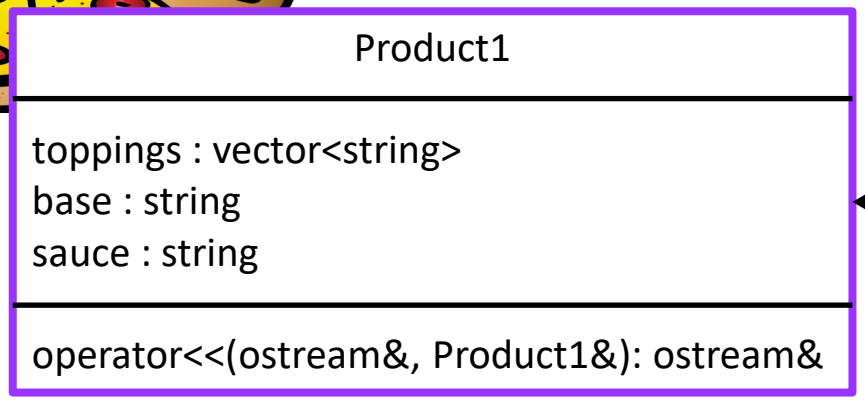
Builder Pattern

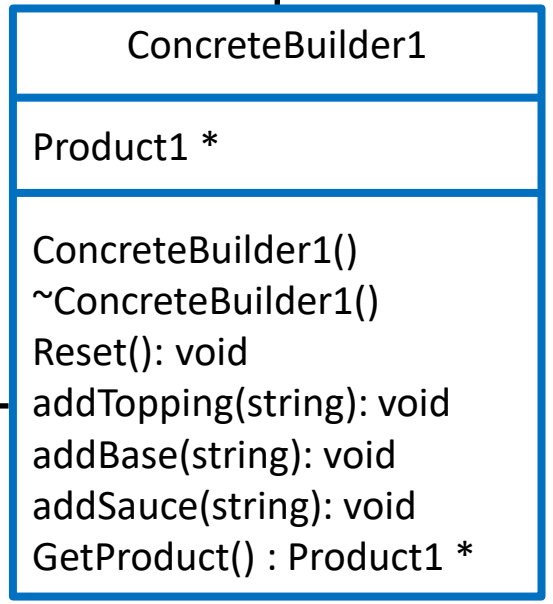
Let's look at some code:

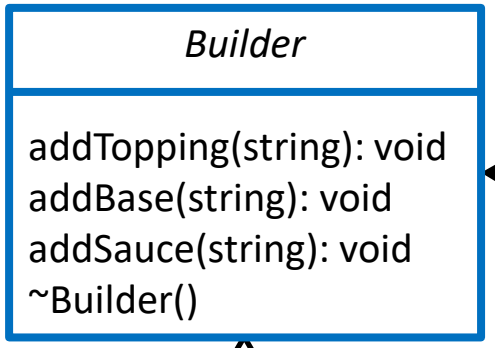
bad_pizza.cpp

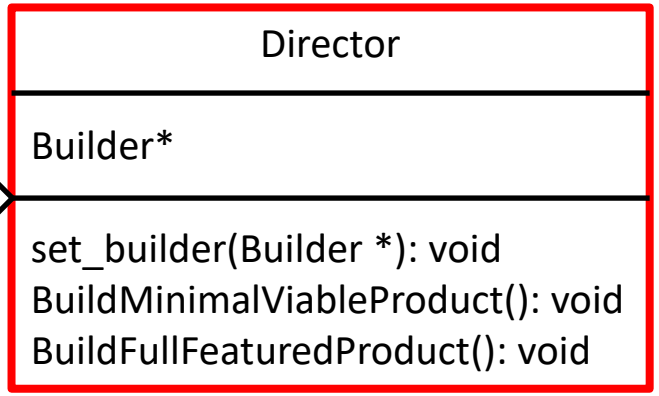
builder_sample_code.cpp











Builder Pattern:
Why and When do we use it

- When you want to get rid of numerous constructors

- When a product is built out of many parts that can be mixed and matched. This allows you to build variations while (possibly) avoiding inheritance hierarchies
- Single Responsibility Principle comes to life as we separate construction logic from the product logic. That is, we **decouple** complex **construction logic** from the **product**.

Builder Pattern – Disadvantages

- Code complexity increases as we introduce more classes.
- Does not work well for products that cannot be broken down into independent parts.

Categorizing Design Patterns

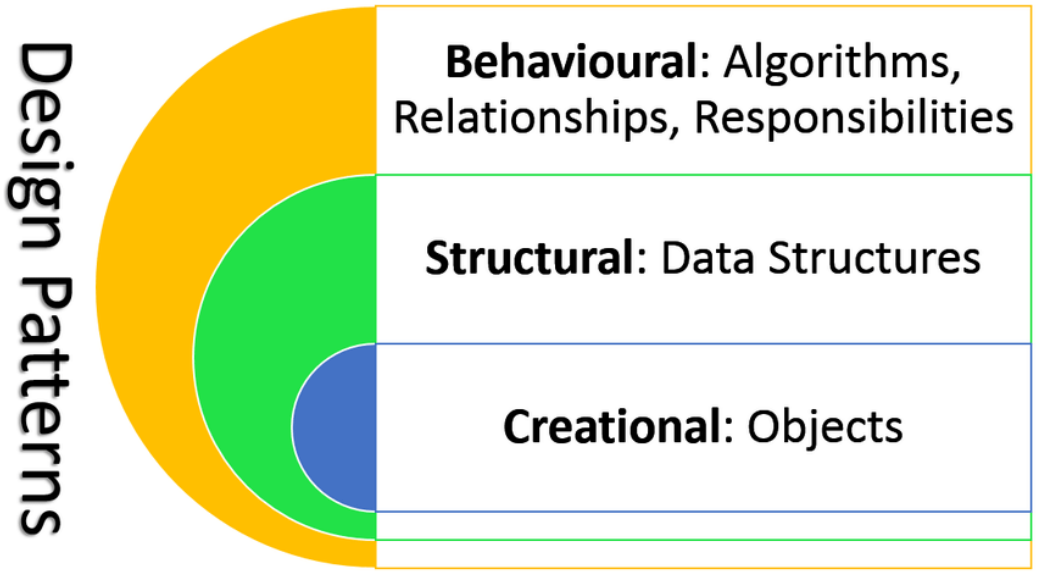
- **Behavioural (We are looking at these!)**

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

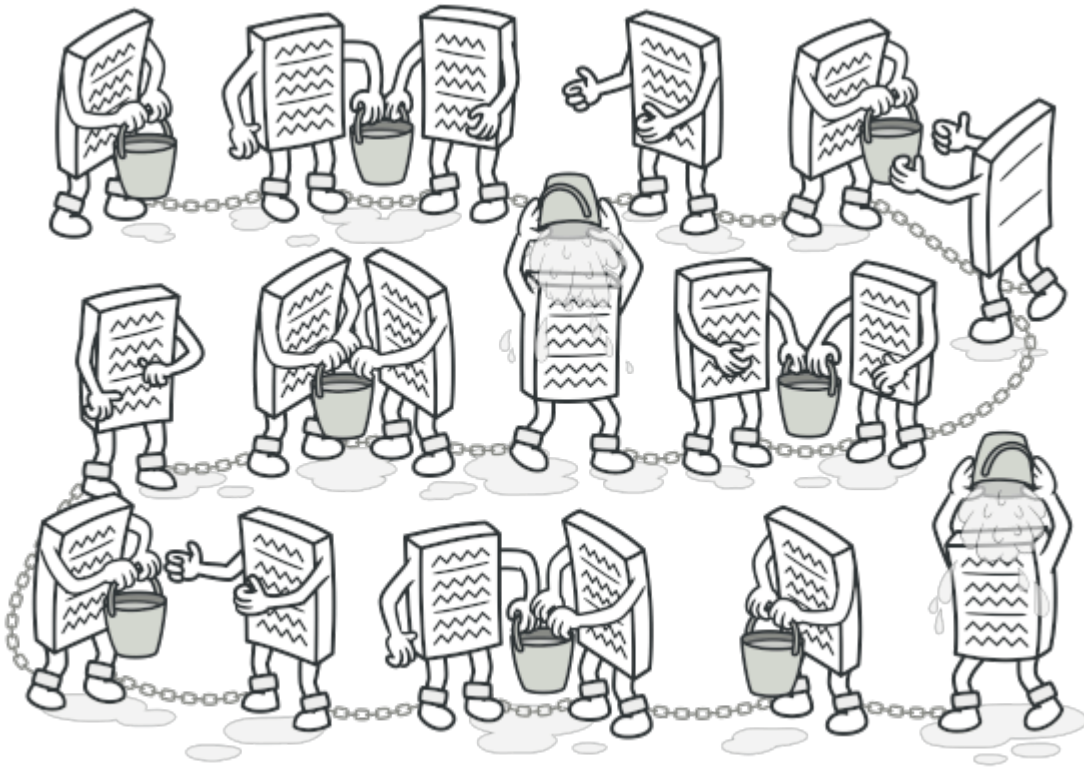
- **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational**

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



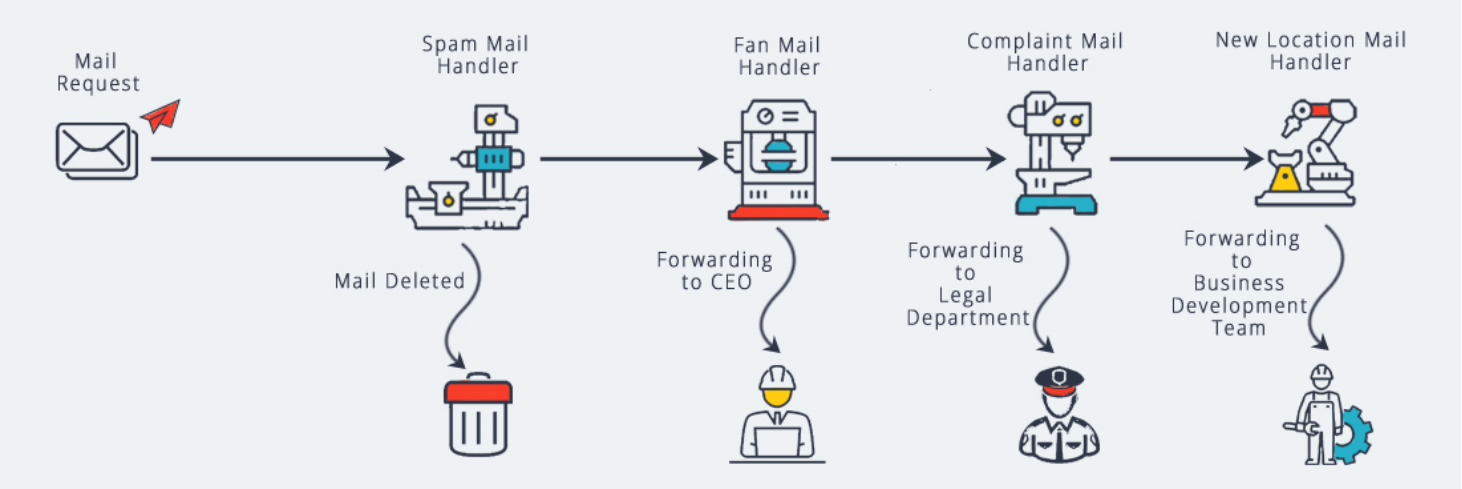
Chain of Responsibility



WHEN YOU WANT TO HAVE A LOT OF RESPONSIBILITIES BUT STILL BE FLEXIBLE

Chain of Responsibility

- When an object or a set of objects needs to undergo different “steps” of processing.
- These could be checks, validation, formatting, security, setup, etc.
- Set up a series of **Handlers**. Each handler is unique but **implements the same interface** and does something to a **Request**.
- Each Handler **has a reference to another handler** and it may, depending on its code, pass on the request to another handler.



Chain of Responsibility - Scenario

Say there was an old fashioned school that used paper forms.

Students had to fill out an **Enrolment Form** every semester listing the courses they wanted to enrol in.

This form undergoes a series of validation checks, and gets processed by hand.

Student Enrolment Form

Hogwarts School of Witchcraft and Wizardry

This form needs to be filled by anyone students in years 3 – 7. Students in years 1 and 2 have their subjects predetermined.

Name: _____

Student ID: _____

Age: _____

Course Code	Course Name	Semester # (1-2)

Student Enrolment Form

- Eventually (Finally!) the school went digital with their administrative tasks.
- Forms were processed by their administrative system.
- The code to process their enrolment form kind of looked like this.
- This is terrible.

```
pair<string, bool> process_enrolment(EnrolmentApplicationForm student_form) {
    bool validated = true;
    StudentRecord student_record = database.get(student_form.studentId);

    if(student_record.name != student_form.name)
    {
        validated = false;
        return make_pair("invalid record", validated);
    }

    if(student_form.fees_paid) {
        validated = false;
        return make_pair("fees not paid", validated);
    }
    //etc
}
```

Student Enrolment Form

- The system also processes other similar forms.
- Many of the forms share different **validation checks**.
- Sometimes the order in which they are done is changed depending on the form.
- Right now the code is duplicated and difficult to maintain. We have several big classes with redundant code.

```
pair<string, bool> process_enrolment(EnrolmentApplicationForm student_form) {
    bool validated = true;
    StudentRecord student_record = database.get(student_form.studentId);

    if(student_record.name != student_form.name)
    {
        validated = false;
        return make_pair("invalid record", validated);
    }

    if(student_form.fees_paid) {
        validated = false;
        return make_pair("fees not paid", validated);
    }
    //etc
}
```

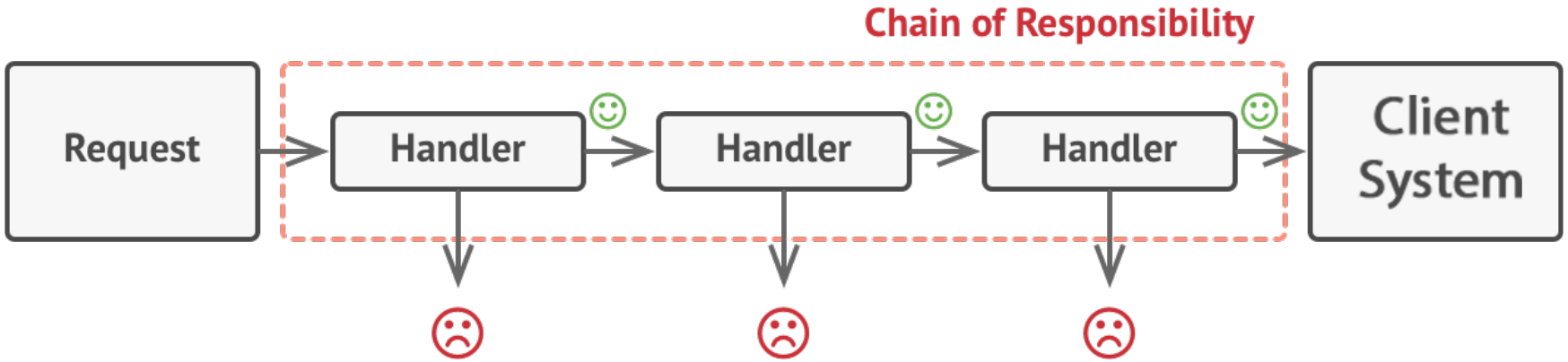
Chain of Responsibility

The Chain of Responsibility pattern **separates** these different **processing steps into different classes**. Each class is a **Handler**

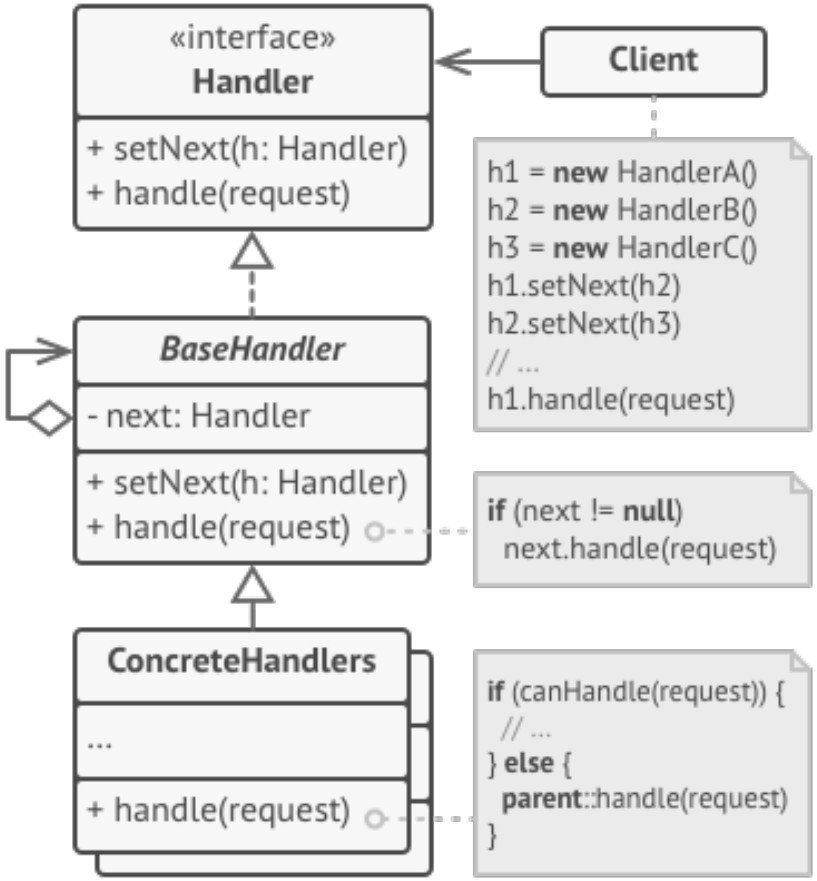
All these classes share the same interface, that is how one handler can pass on the request to another handler.

It is kind of like a **linked list of responsibilities** where each handler forms a node. We can arrange a different list for different scenarios.

The execution may stop midway and exit the chain if the Handler deems it necessary



Chain of Responsibility



Requirements:

- Each Handler implements the **same interface.**
- Base Handler is an **Optional** parent class that can hold some duplicate code (such as **setNext(h: Handler)**)
- Each handler implements a **handle(request) method** which is where they carry out their specific code.

Let’s implement our enrolment form!

Or at least a part of it.

Let’s draw out the UML diagrams

Enrolment Application Forms

- **Student Account**

Enrolment System

3 **handlers** to verify application forms:

- **StudentValidationHandler**
- **CheckCourseOfferingHandler**
- **FeesHandler**

Cheesy Teamwork Image for motivation

Let's implement our enrolment form!

Or at least a part of it.

Let's draw out the UML diagrams

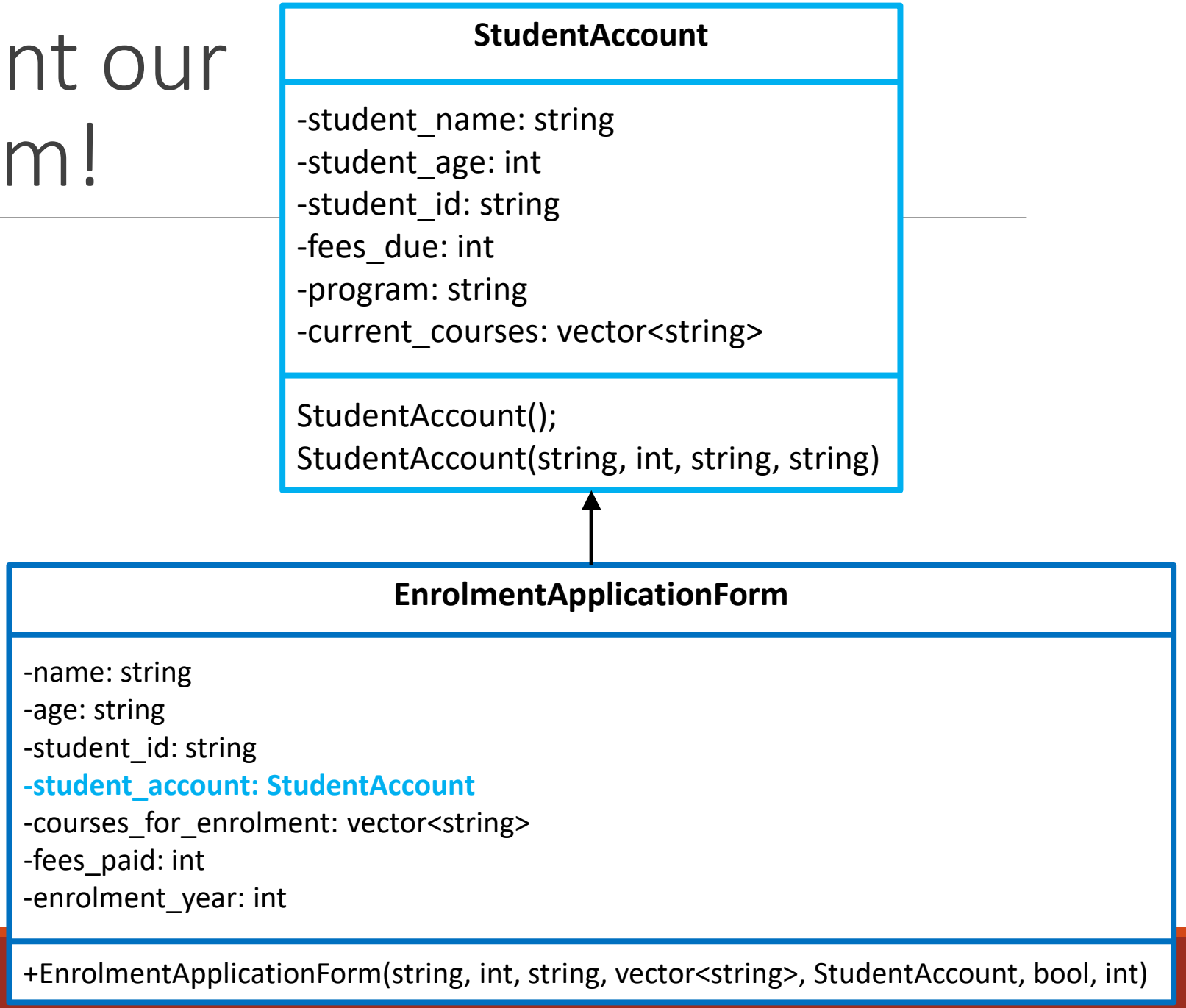
Enrolment Application Forms

- **Student Account**

Enrolment System

3 handlers to verify application forms:

- StudentValidationHandler
- CheckCourseOfferingHandler
- FeesHandler



Let's implement our enrolment form!

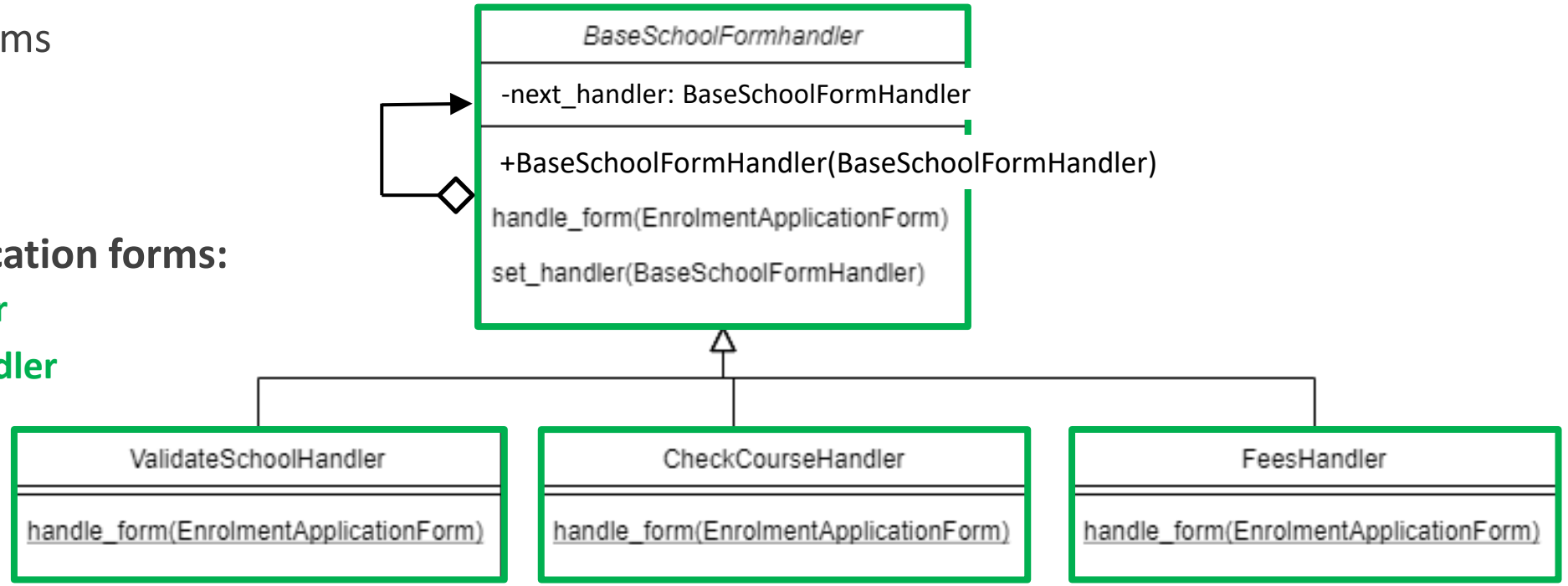
Or at least a part of it.

Let's draw out the UML diagrams

Enrolment Application Forms
- Student Account

Enrolment System

3 handlers to verify application forms:
- **StudentValidationHandler**
- **CheckCourseOfferingHandler**
- **FeesHandler**



Let’s implement our enrolment form!

Or at least a part of it.

Let’s draw out the UML dia

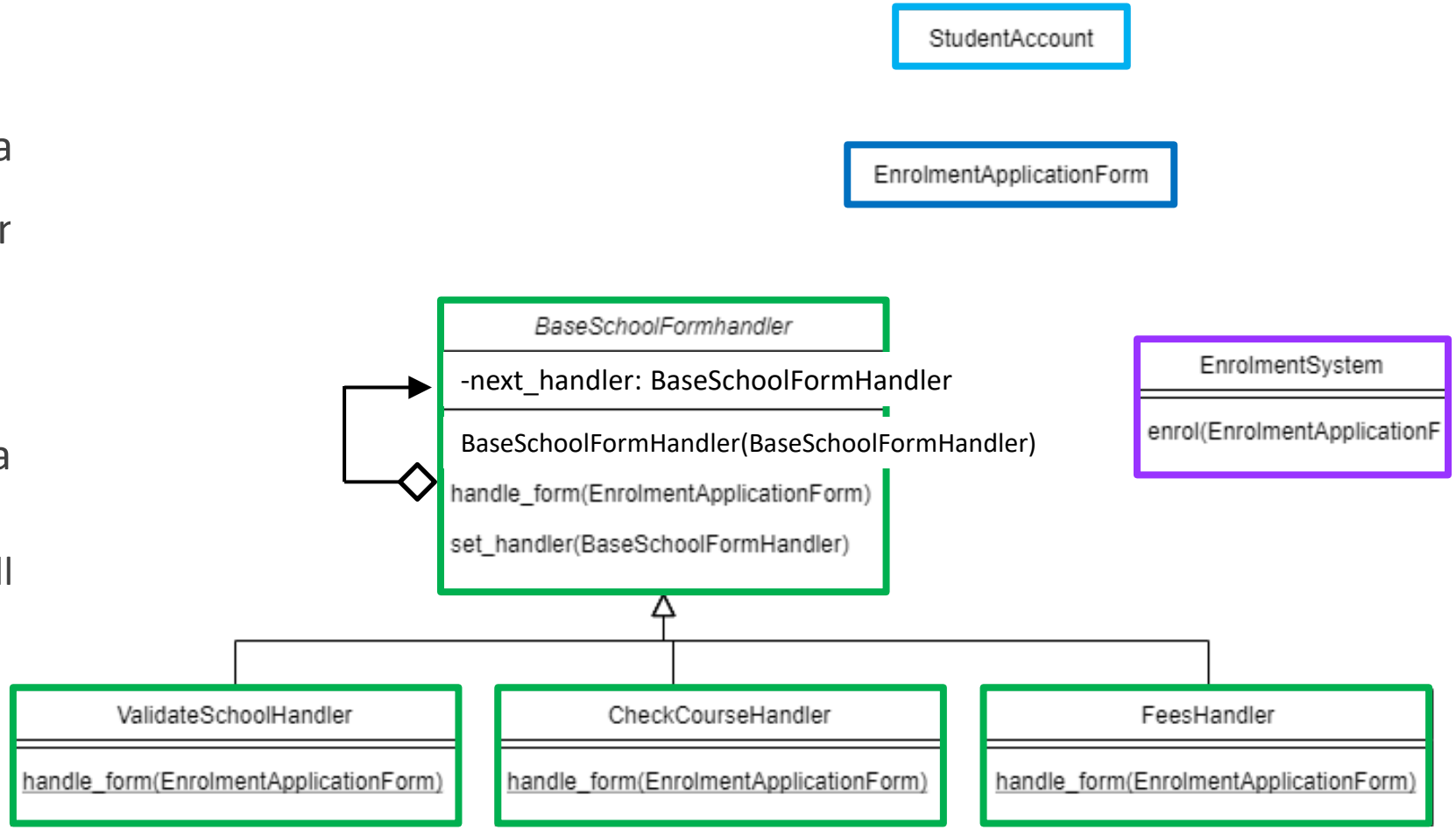
Enrolment Application Forr

- Student Account

Enrolment System

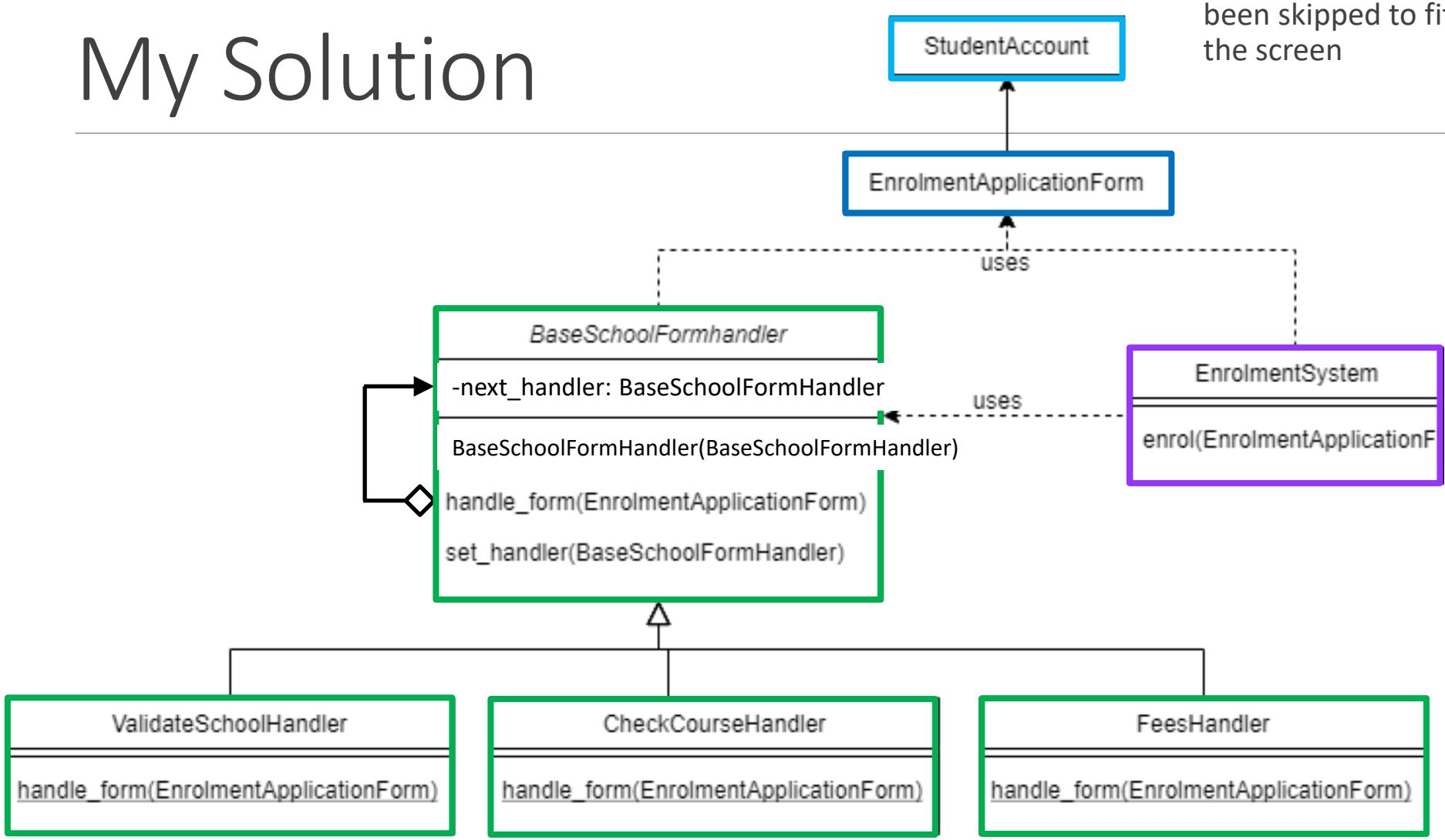
3 handlers to verify applica

- StudentValidationHandler
- CheckCourseOfferingHandl
- FeesHandler

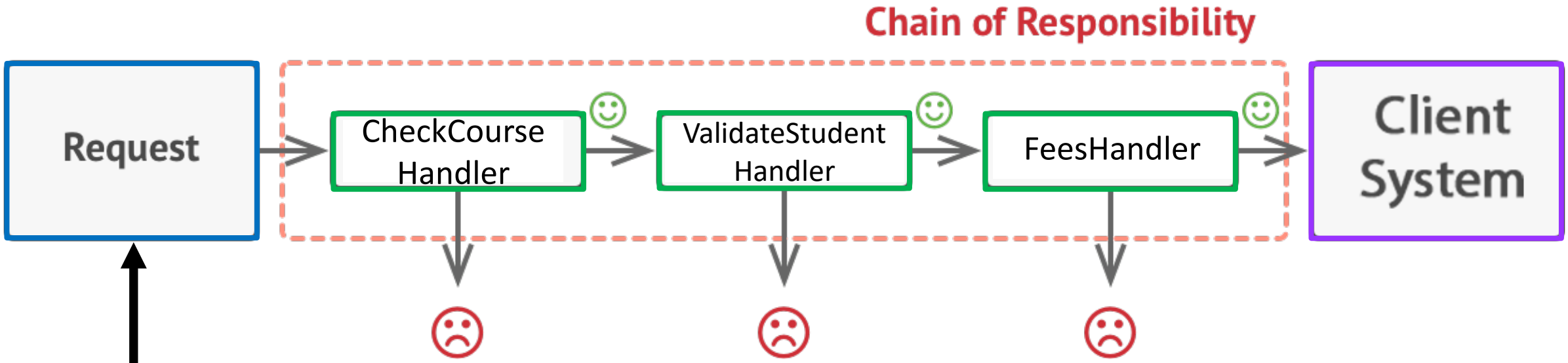


My Solution

NOTE: These would have attributes in them, but have been skipped to fit it all on the screen



My Solution



Client System
1. Create student account and forms
2. Send form to enrolment system

chain_responsibility.cpp

Chain of Responsibility: Why and When do we use it

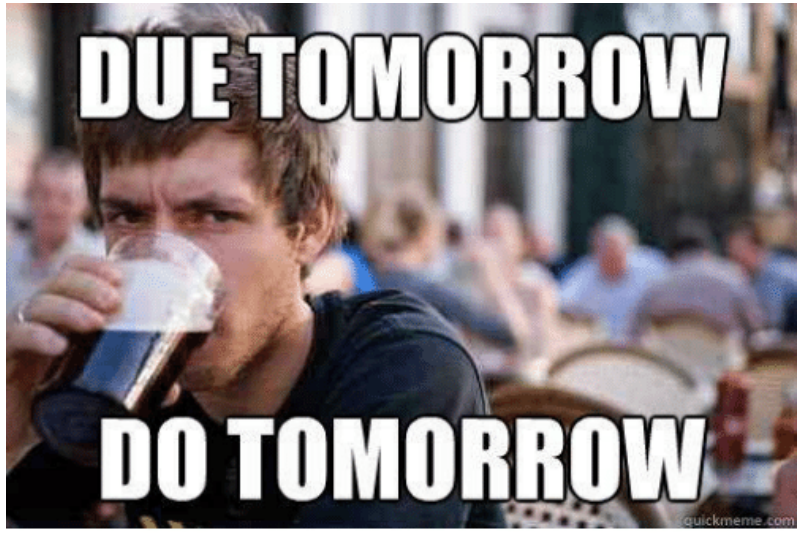
- If your program is expected to process different kinds of requests in various ways.
- When you need to do something in a particular order.
- Can control the order of request handling if the sequence and ordering of request-processing is not known beforehand and needs to be determined at run-time.
- Single Responsibility Principle. Each handler does one thing. We have decoupled **classes that invoke operations** (E.g. **EnrolmentSystem**) from **classes that perform operations** (the **handlers**)
- Open/Closed Principle. We can introduce new handlers without modifying existing handlers or client code.

Chain of Responsibility – Disadvantages

- More classes to maintain.
- Some requests may end up unhandled. This may happen if we don’t set up the ordering of handlers properly.

Lazy Initialization

Lazy Initialization



The art of procrastination.

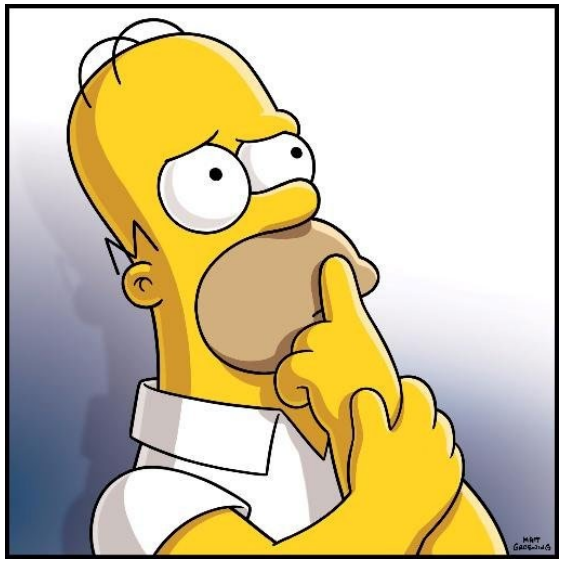
- Lazy Initialization is probably the easiest design pattern out there. It is also known as Lazy Loading.
- The **Lazy Initialization pattern** mandates that we **don't initialize an object/resource unless we need it.**
- Why tax the system unnecessarily?
- This is usually applied to objects that are **expensive to initialize** either due to memory or computation costs.
- In simple words. **We procrastinate on initializing an expensive resource until we need it**. Laziness is the hallmark of every good programming.

Lazy Initialization

I could continue talking about it.

But let's look at some code: lazy_initialization_sample_code.cpp

Cars are great, but where can I actually use this?



Ok I hear you!

Contrived examples are great for understanding but not good for application.

So, hypothetically speaking.

Just hypothetical.

Say we were writing an application that had to maintain a connection or a web session or something like that.

If we are not sending any requests, should we keep the web session open?

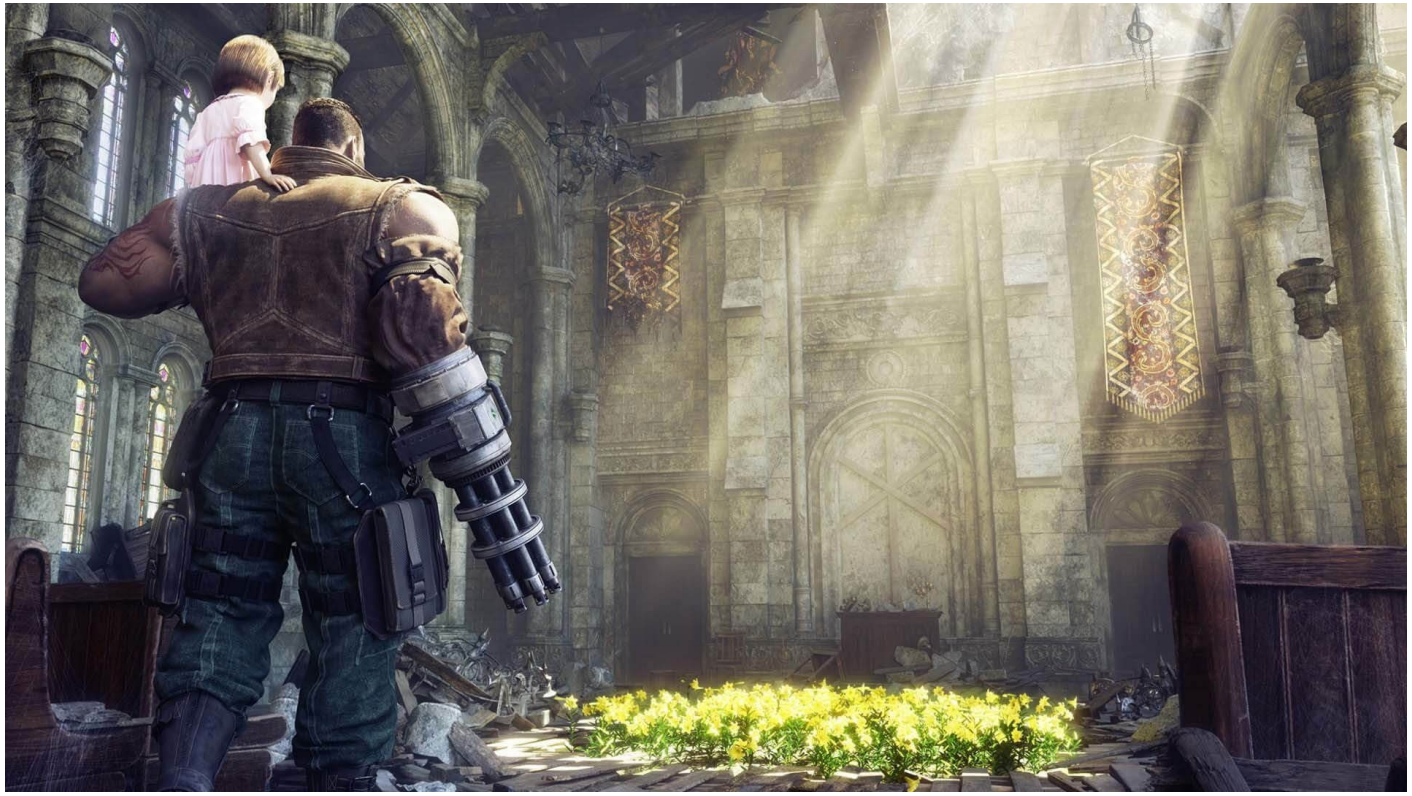
Or should we open one when we need it and say, close it if there has been no activity in 3-5 minutes or so?

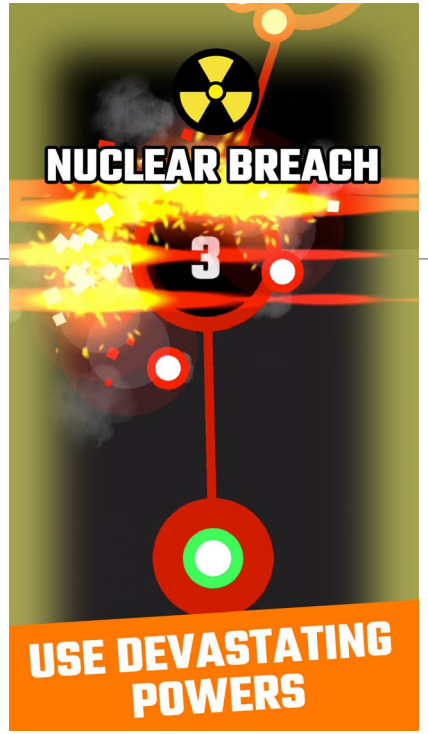
We can lazily initialize our web session and close it after a period of inactivity.

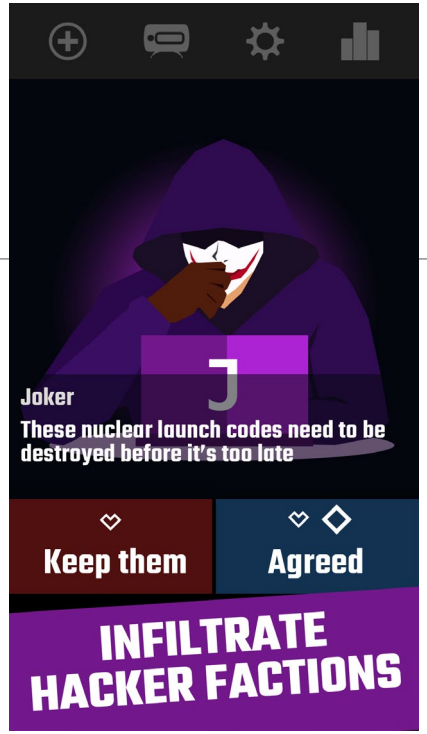
Examples

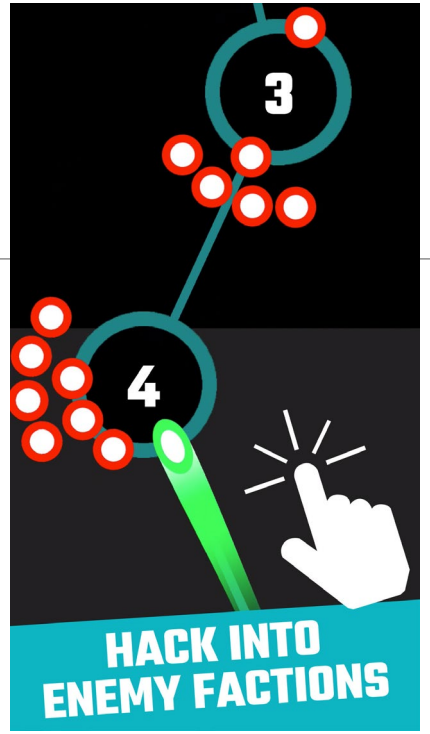
Lazy Initialization is used when rendering complex environments in games for example.

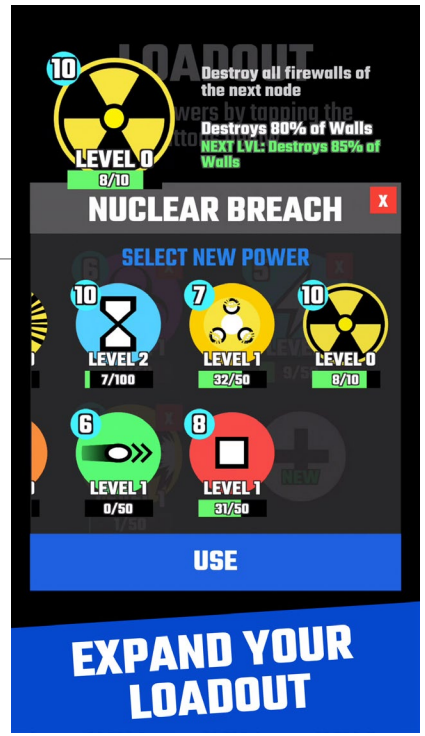
If the player is not looking at an area, don’t keep it in memory and render it.











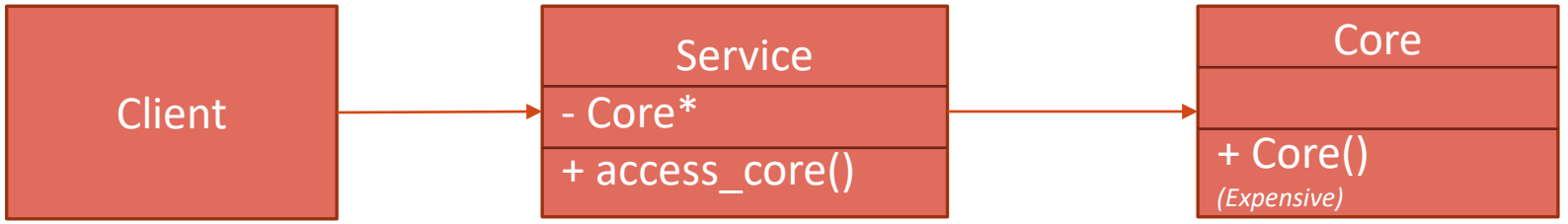
I've used lazy initialization in my games. Games have many screens with UI components. The player won't always see every screen when they play the game

Don't load and save all screens in memory when the game starts

Load screens when the player needs to access them

Lazy Initialization

- That's it.
- That's lazy initialization.
- Let's look at its advantages and disadvantages.
- Also, it can be implemented in so many ways, there really isn't much point in a UML diagram.
- But if you insist. Here is a possible UML diagram.



Lazy Initialization

Advantages:

- Dedicate resources (memory / processing) on a on-demand basis.

Disadvantages:

- Not a good choice if the object needs to be initialized every time it is accessed (assuming it is accessed frequently).