
COMP2511

Week 7 Tutorial

Last Tutorial

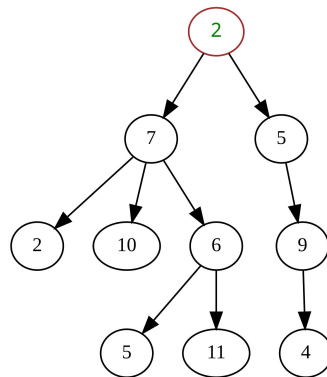
- Strategy Pattern
- Observer Pattern
- State Pattern

This Tutorial

- Composite Pattern
- Factory Pattern
- Abstract Factory Pattern

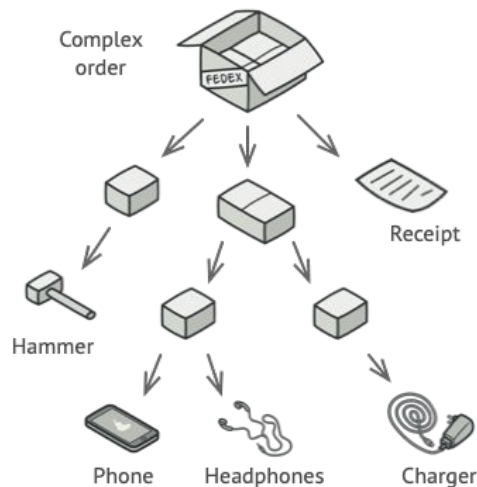
Composite Pattern

- The Composite pattern is a **structural** design pattern that organises objects into **tree structures**, i.e. recursive structures where each object is a node connected to zero or more subtrees (a tree can be null/empty).
- Terminology:
 - Nodes that have (non-empty) subtrees as children are referred to as **composites**.
 - Nodes that have no children are referred to as **leaves**.



Composite Pattern

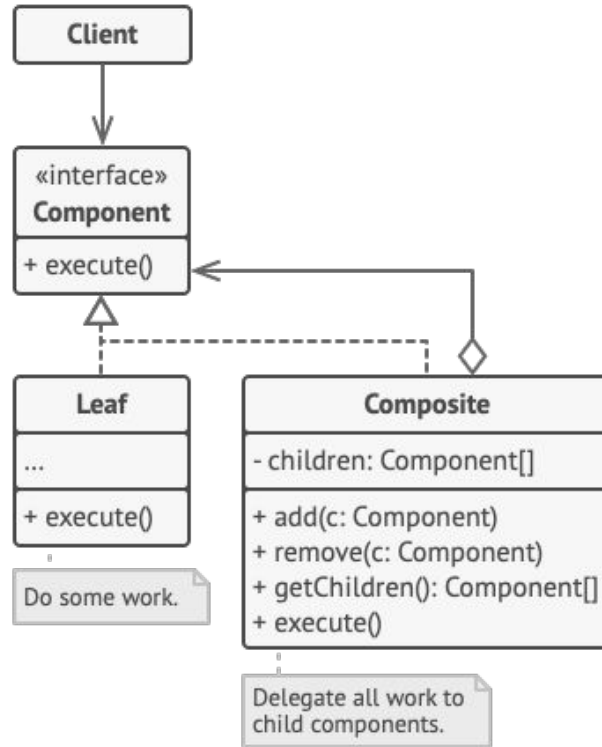
- Imagine you have a box of items, and you're trying to evaluate the value of everything inside the box. Inside that box, you could have individual items and more boxes, which in turn could have more items and boxes.



Composite Pattern

- This can be viewed in terms of a tree: boxes are **composite** objects, while the individual items are **leaves**.
 - The value of a box is the sum of all of the values of the items and boxes inside (the **recursive case**), and the values of the individual items are given (the **base case**).
- Opening every single one of the smaller boxes yourself to evaluate their values would require a lot of effort. It would be preferable if someone already did that work for you and stuck sticky notes on top of the boxes with their overall values, so you immediately know the value of those boxes as if they were a single item.
- Essentially, we want to treat composites and leaves the same. In OOP terms, this means that composites and leaves should implement a common **interface**.

Composite Pattern Implementation



Live Example

src/calculator (details in
README.md)

Factory Pattern

- The Factory pattern is a **creational** design pattern that provides interfaces to create objects, abstracting away the logic of using and calling constructors to another class or classes.
- The construction logic is delegated to classes called **factories**. Rather than calling constructors directly, objects get created by calling methods defined in a factory, which return objects of the required type.
- This is useful for situations where the logic required to construct an object is complicated and bloats the main program logic.
 - If you were writing a library, this would also be relevant if you don't want to expose the creation logic to clients.

Factory Pattern

- **Disclaimer:** In this course, the main use of the factory pattern is moving complicated construction logic away from the 'important' classes into a factory class. This is different to how you will see the factory pattern defined in most other resources.
- The 'more typical' presentation of the factory pattern emphasises the ability to create objects without having to specify their concrete types.
 - For example, instead of doing `Car c = new Car()` or `Train t = new Train()`, we could define a `createVehicle()` method in an interface and make `CarFactory` and `TrainFactory` classes that implement the interface. Then if `Car` and `Train` both inherit `Vehicle`, we can do `Vehicle v = factory.createVehicle()`, where the concrete type (`Car` or `Train`) depends on the type of factory used. This decouples the code from dealing with concrete types.
 - We will see more of this when we look at the Abstract Factory pattern.

Live Example










src/characters (details in
README.md)

Abstract Factory Pattern

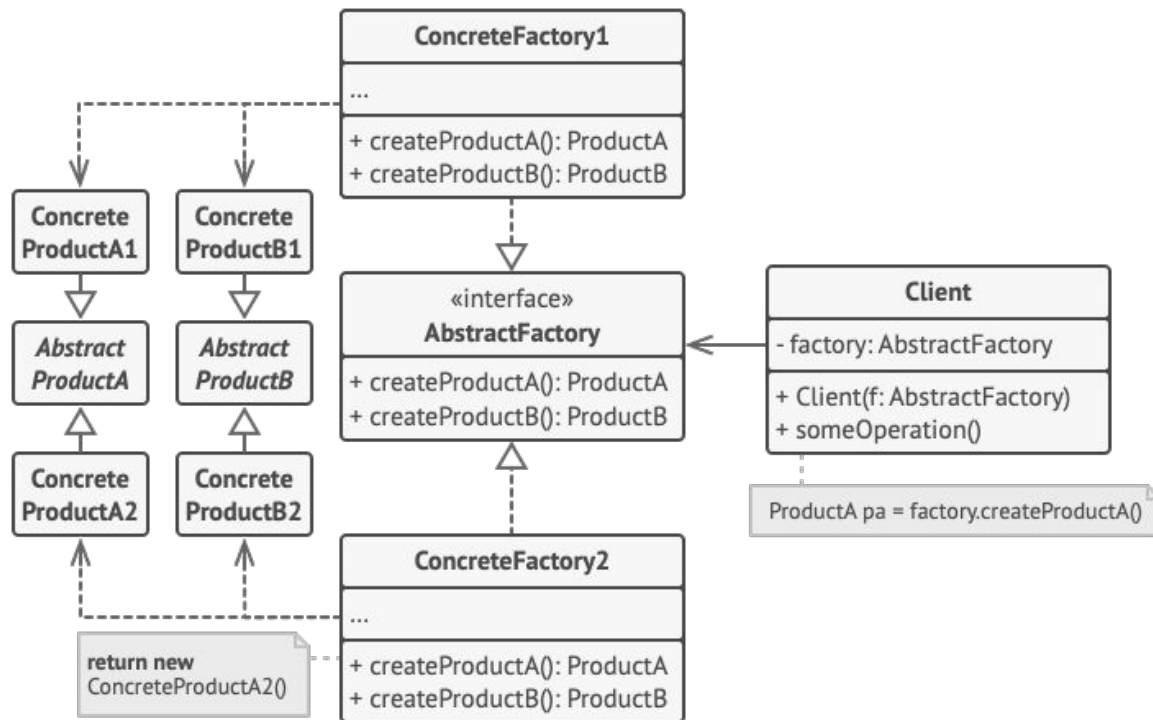
- The Abstract Factory Pattern is a **creational** design pattern that provides interfaces to create **families of related objects** without specifying their concrete classes.
 - This is very similar to the Factory Pattern (as the name suggests). The difference is that the Abstract Factory pattern is relevant when you are creating a group of objects that can be categorised into different *variants*.
 - For example, consider a shirt, pants and shoes. This is a group of related objects, and you can further classify them into different categories - colour, size, brand etc.
 - The Abstract Factory pattern suggests that instead of creating only one factory to handle the creation of these products, you should make a **distinct** factory for each category, and an interface that each of the factories implement. If you were categorising clothing by colour, you could have a `ClothingFactory` interface with `createShirt()`, `createPants()`, `createShoes()` methods, and then concrete `RedClothingFactory`, `GreenClothingFactory` etc. classes which each provide their own implementation of those methods.

Abstract Factory Pattern - Categories Example

- In this example, all of the products here (chair, sofa, coffee table) are different types of furniture; these can be further classified into different styles (Deco, Victorian, Modern).
 - Which different factories would you have? What should each factory be able to produce?

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

Abstract Factory Pattern Implementation



Live Example

src/characters2 (details in
README.md)
