Reporter : Nguyen Duong Thu Thuy

1. **Design Pattern**
2. **Singleton**

* Group: **Creational**
* The singleton pattern guarantees that only one instance of a class is instantiated.
* Access that variable anywhere in the project (Global Access)

**Code example:**

****

1. **Abstract Factory**

* Group: **Creational**
* Resolve: the problem of creating entire product families without specifying their concrete classes.
* Abstract Factory defines an interface for creating all distinct products but leaves the **actual product creation** to **concrete factory classes**. Each factory type corresponds to a certain product variety.
* What it does: create a Super factory in order to create other factories (Factory of Factory)

**Components:**

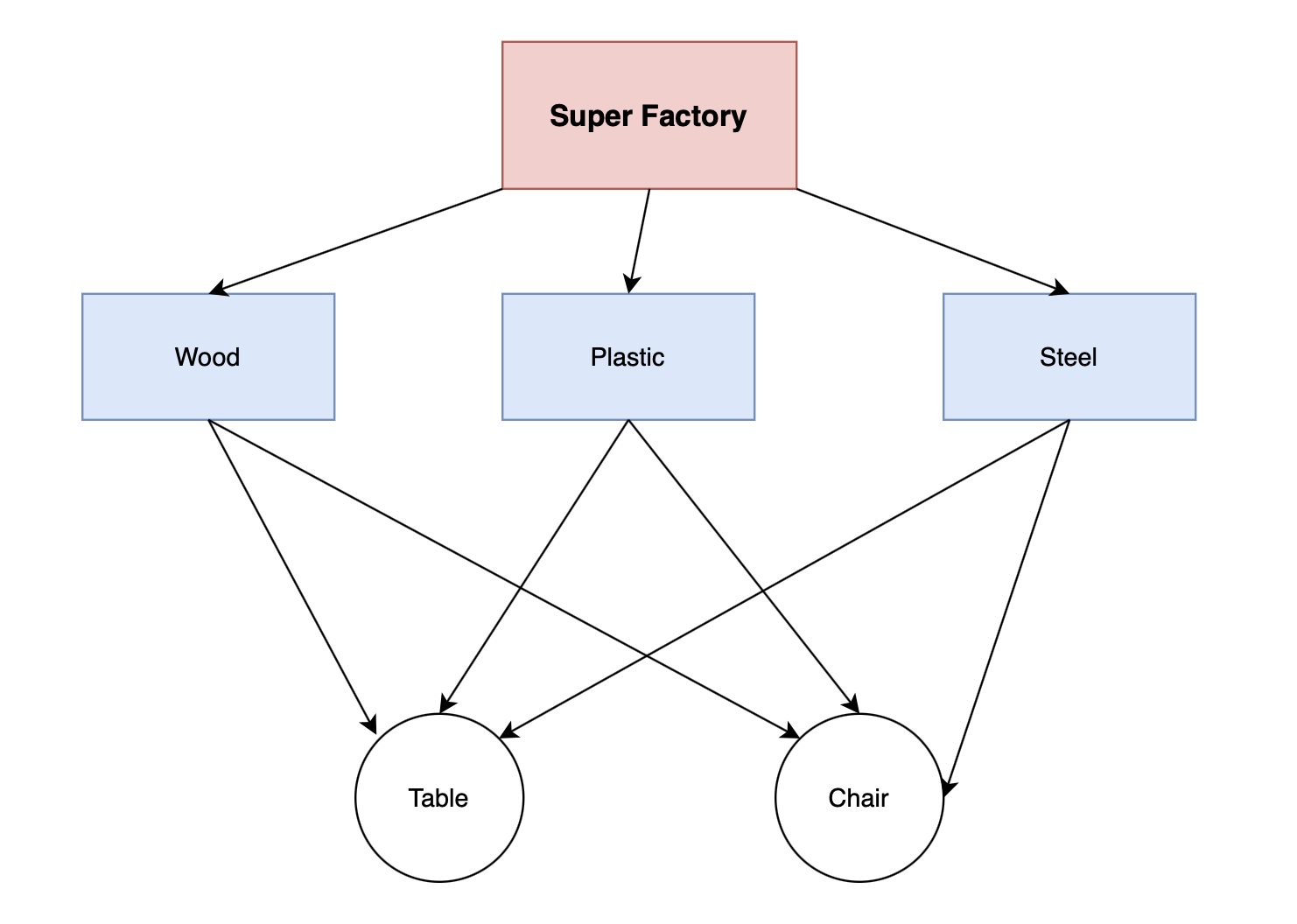
+ **AbstractFactory**: interface/abstract class/protocol with the methods to create other abstract objects

+ **ConcreteFactory**: contains the methods to create concrete objects

+ **AbstractProduct**: interface/abstract class/protocol to define the abstract object

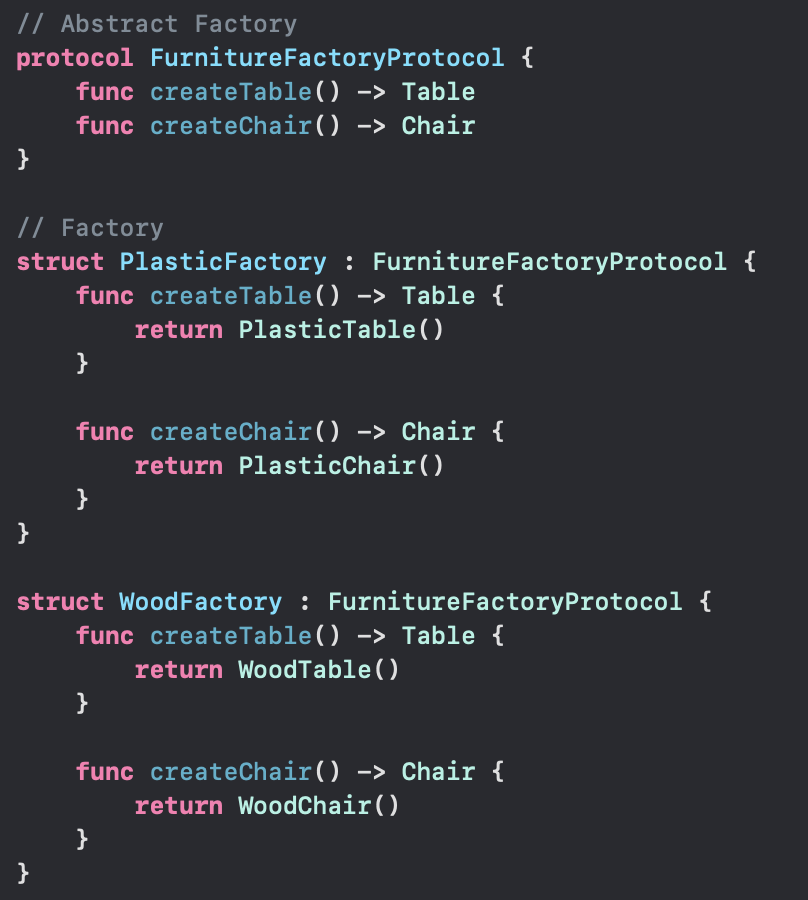
+ **Product**: concrete implementions (methods) that are pre-defined in AbstractProduct

+ **Client**: the objects that use AbstractFactory and AbstractProduct

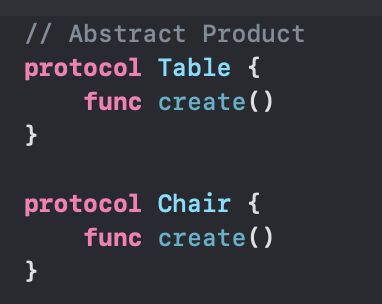


**Code Example**

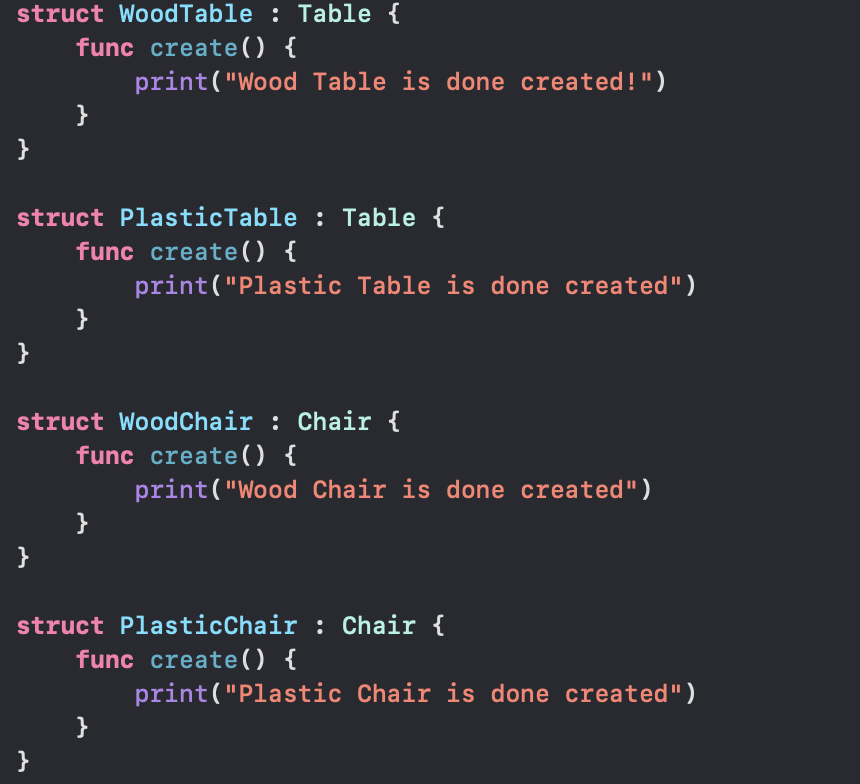
* Abstract Factory & Concrete Factory



* Abstract Product



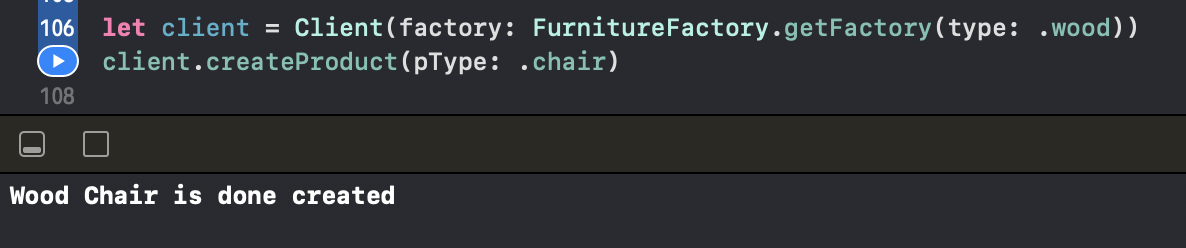
* Concrete Product



* Client



* Output:



1. **Decorator**

* Group: **Structural**
* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
* Acts like a wrapper for the current class. Anytime the object needs to add a new feature/function, the current object will be wrapped in a new wrapper (decorator class)
* **Composition** instead of **Inheritance**

**Components:**

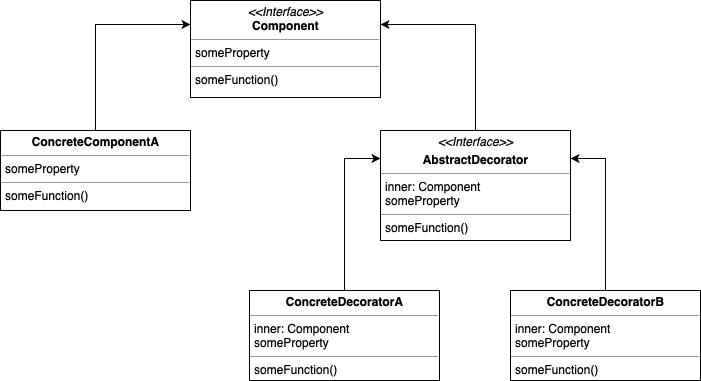
+ Component: an interface/protocol defining all the common methods

+ ConcreteComponent: implement all the abstract methods of Component

+ Decorator: an abstract class that keeps a reference to the current object, and at the same time sets up the methods implementation of Components

+ ConcreteDecorator: implement all the methods of Decorator

+ Client: that uses Component



**Code Example:**

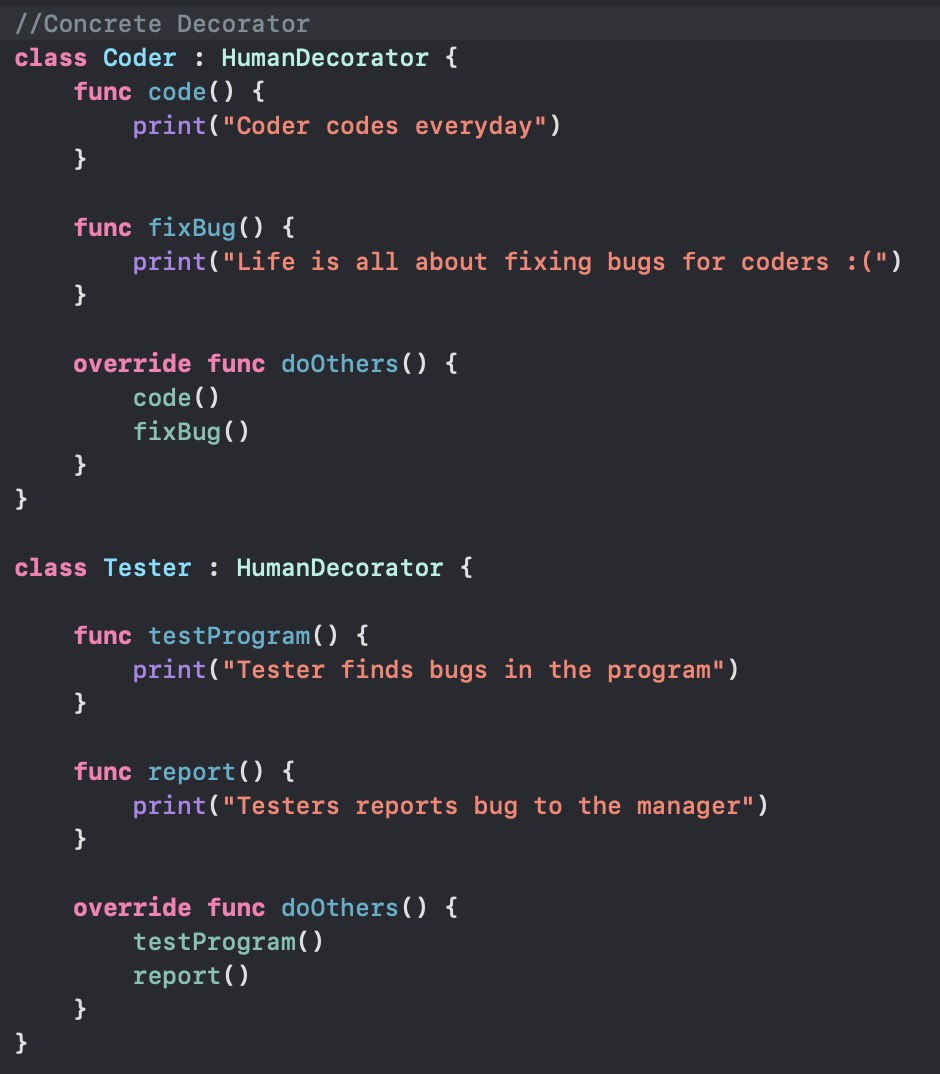
* Component & ConcreteComponent



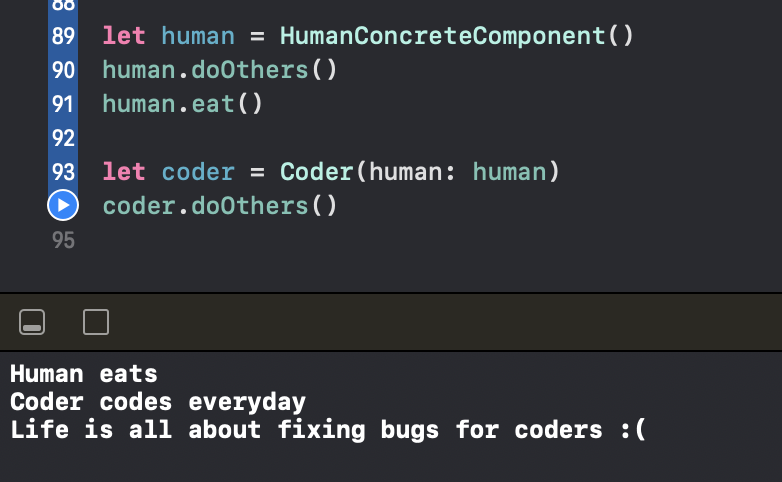
* Decorator



* ConcreteDecorator



* Output:



1. **Strategy**

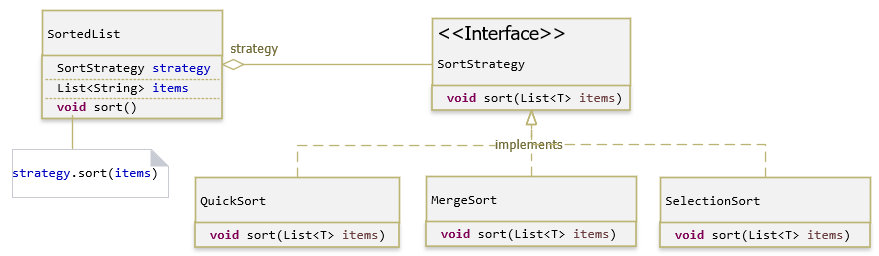
* Group: Behavior
* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it
* What Strategy do: detach a function out of the object. Defined a set of ways of that function. Pick out the best one out of all -> set the function for the object

**Components:**

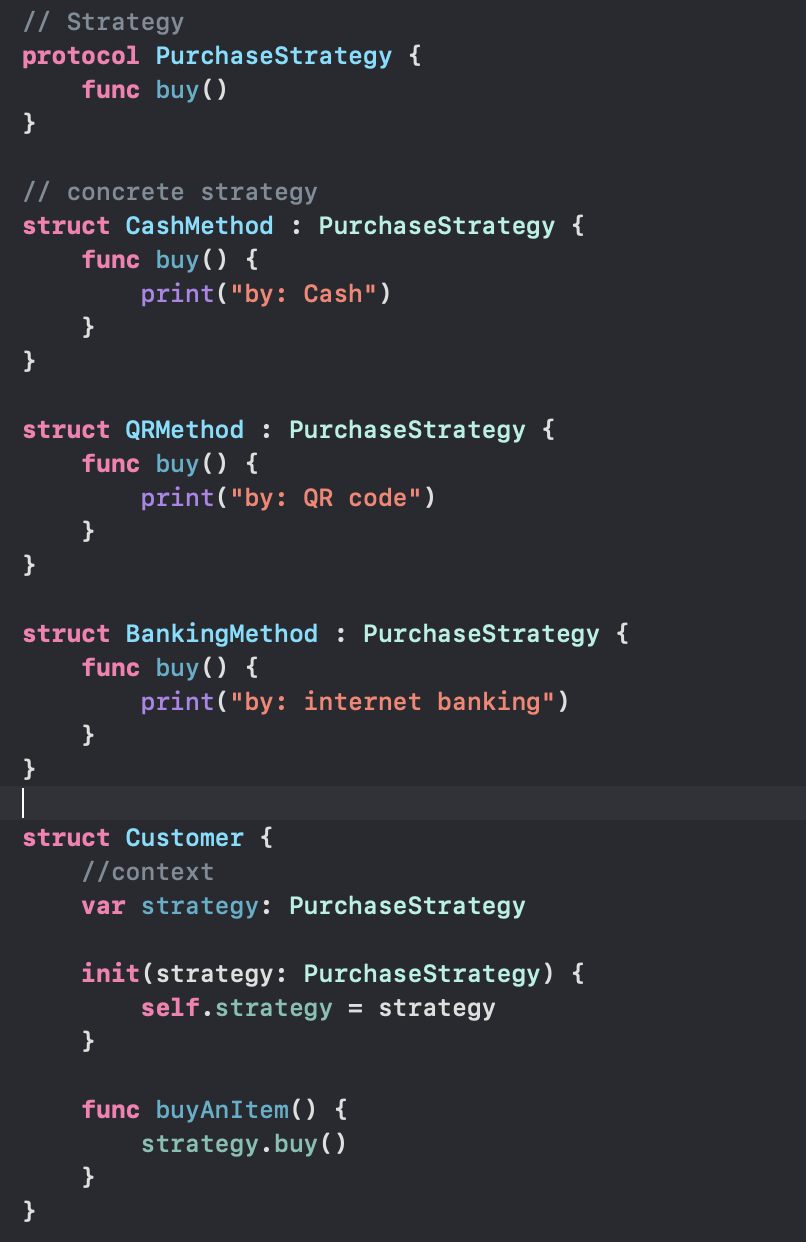
+ Strategy: define the abstract method of the target behavior

+ ConcreteStrategy: set up the concrete implementation of that strategy

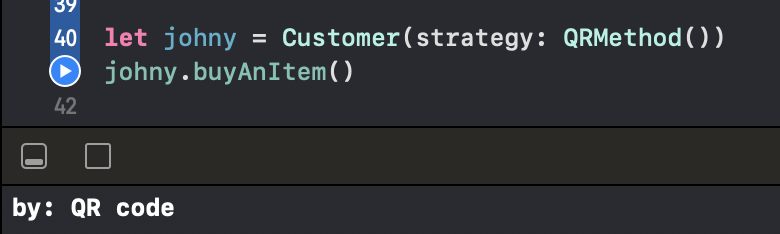
+ Context: contains a reference to the object. Taking request from the user, the strategy gets delegate to choose the suitable implementation



**Code example:**

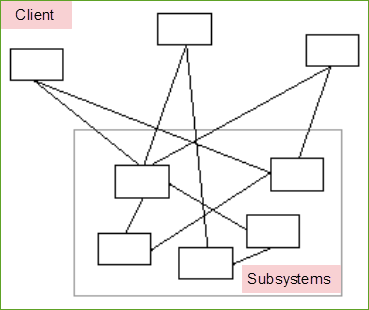
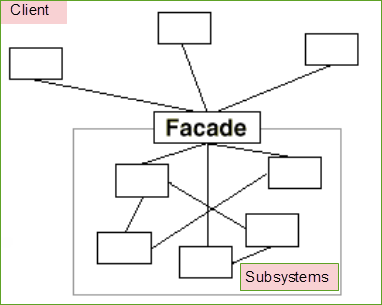
****

* Output

****

1. **Facade**

* Group: **Structural**
* Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use
* Defined a higher interface for a subsytem -> hide complex implementation, make the system easier to use
* Without Facade: With Facade:

** **

**Components:**

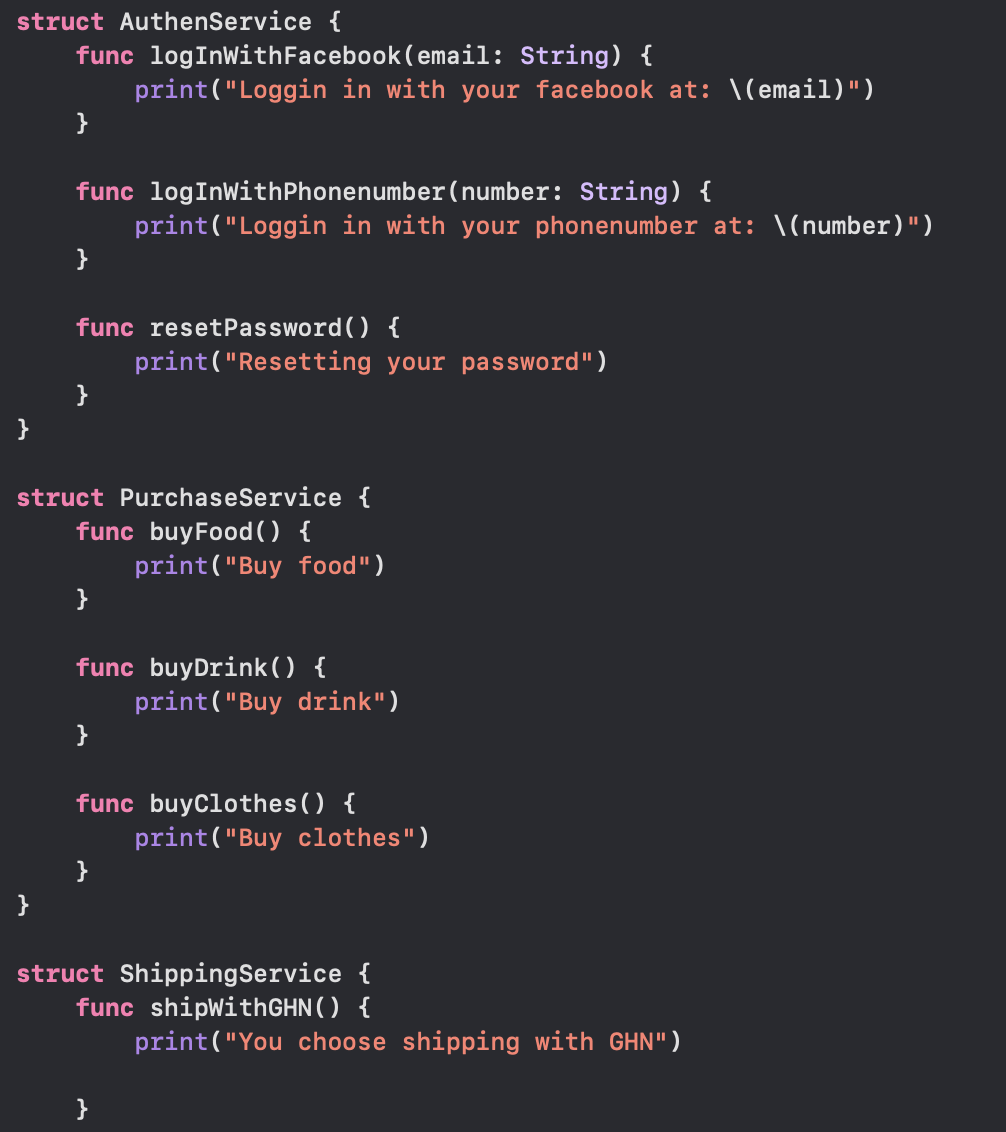
+ Facade: has a transparent view of the inside system

+ Subsystem: where feature of small systems are set up

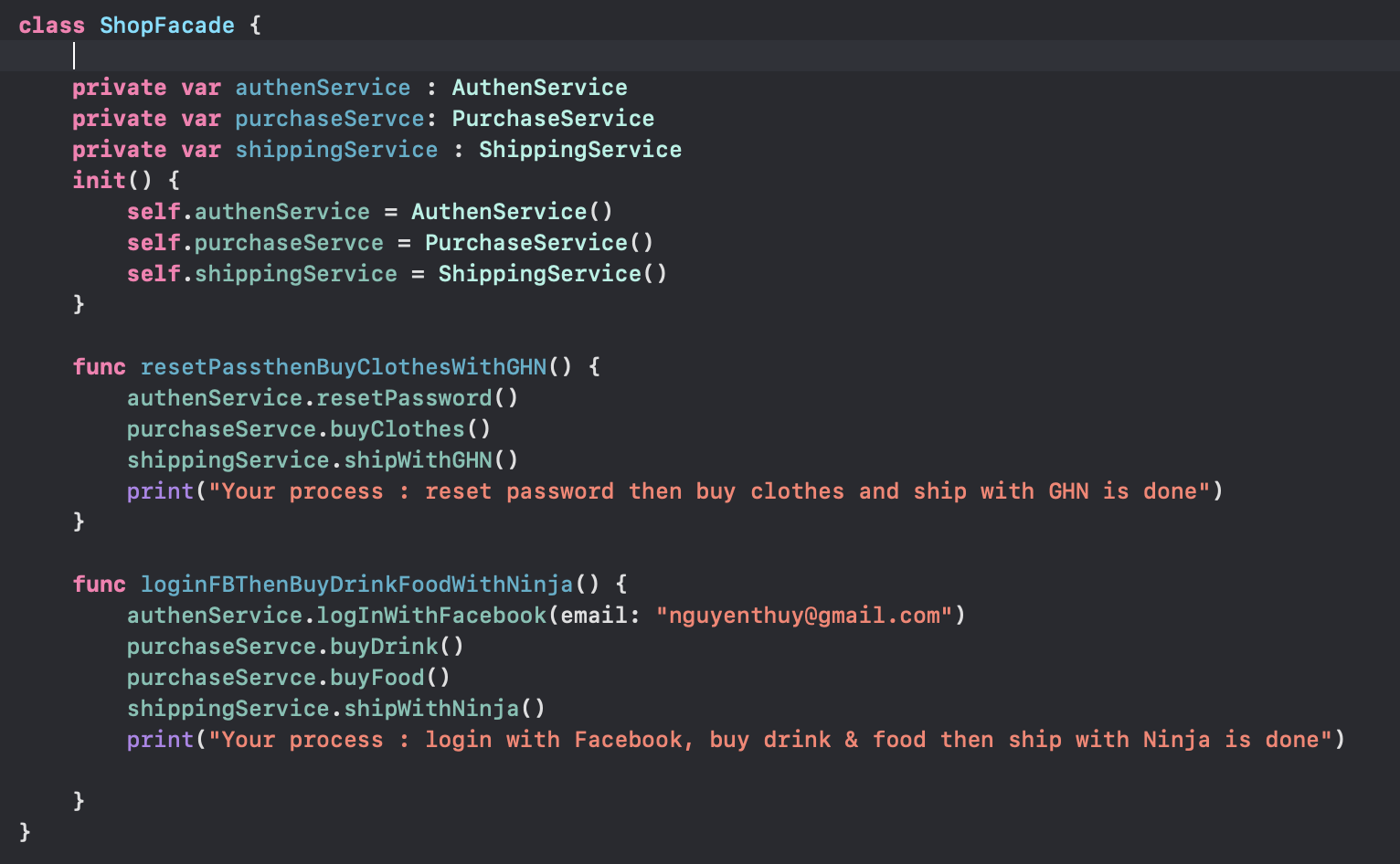
+ Client: that uses Façade

**Code Example:**

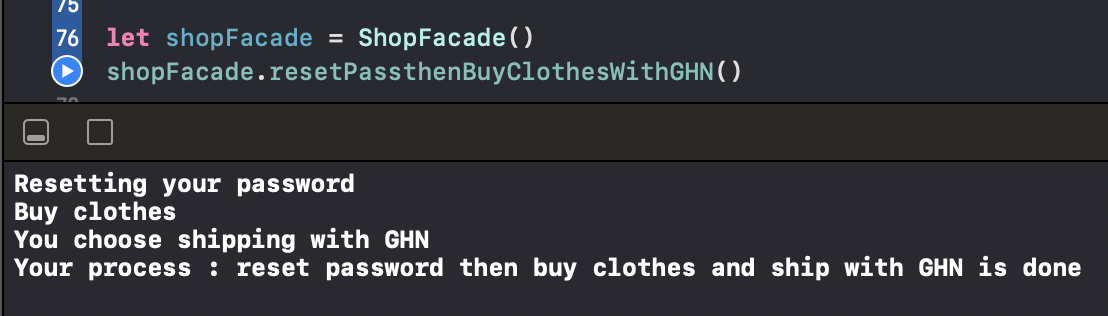
* **Subsystem**

****

* **Facade**

****

* Output

****

1. **Delegation**

* Delegation is a design pattern that enables a class to hand off (or “delegate”) some of its responsibilities to an instance of another class
* Think about *delegation* in the real world. Image your family are making dinner together. Your parents are the key members. Your mothers delegate making soups task to you, her child. Once you’re done making soups, you tell mother and then your family starts the dinner.
* Some key points that stand out:

+ Your parents are in charge of making dinner. You mom delegates the making soups task to you

+ Making soups are your parents’ responsibility. They’re handing off some of that responsibility to you

+ And it goes two ways: you notify your parents when you’re done making the soup

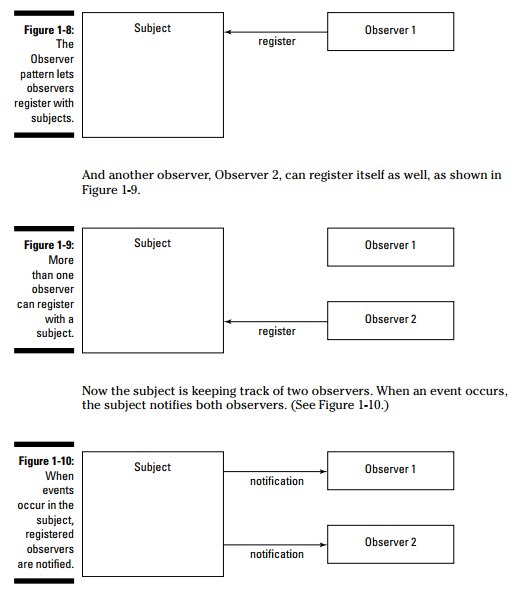
**Code example:**

****

1. **Observer**

* Group: Behavior
* When to use: to notify other objects about the changes of an object -> make corresponding changes
* In Swift, use particularly in updating GUI
* Observer Pattern is also known as Dependents, Publish / Subscribe, Source / Listener

Observer registers to Observable -> get update whenever Observable has a change



**Components:**

+ Subject: contains a list of observers, with methods to add / remove a particular observer

+ Observer: define a method update() for observer when observable has any change

+ ConcreteSubject: concrete implementations for Subject, store states of ConcreteObserver, notify ConcreteObserver when the Observable has any change

+ ConcreteObserver: concrete implementations for Observer, store state of subject, execute the update process to synchronize its state with the subject

**Code example:**

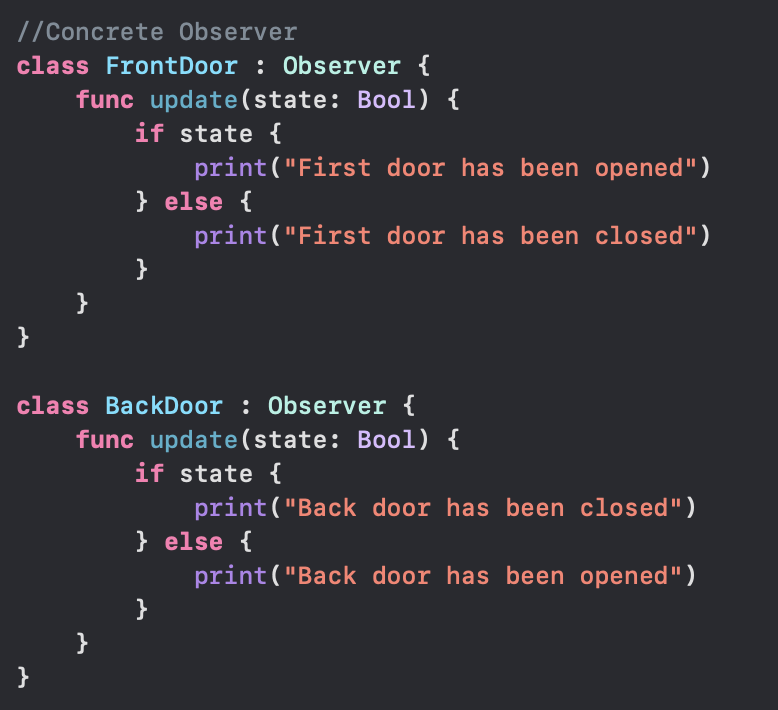
* Subject:



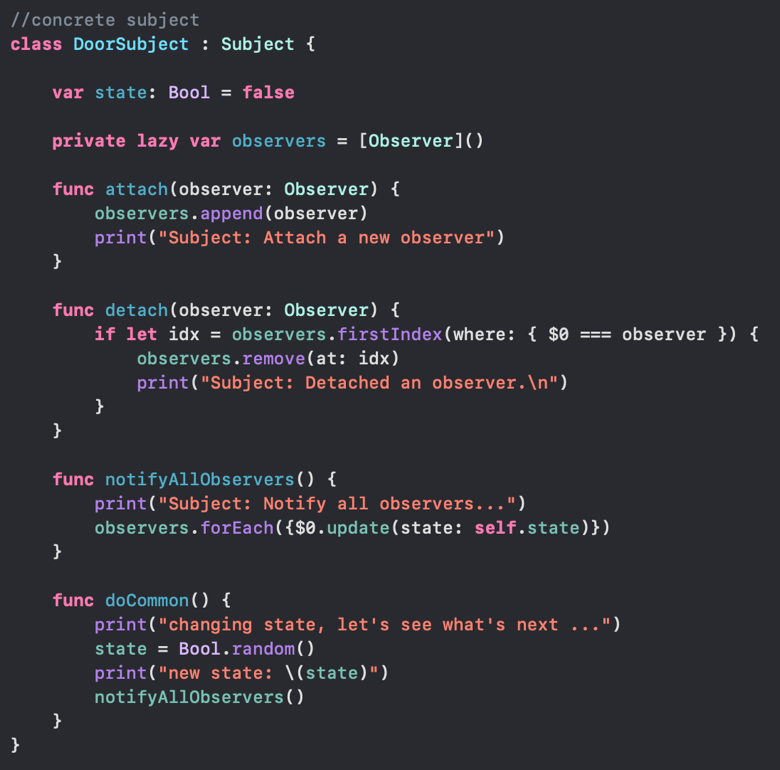
* Observer



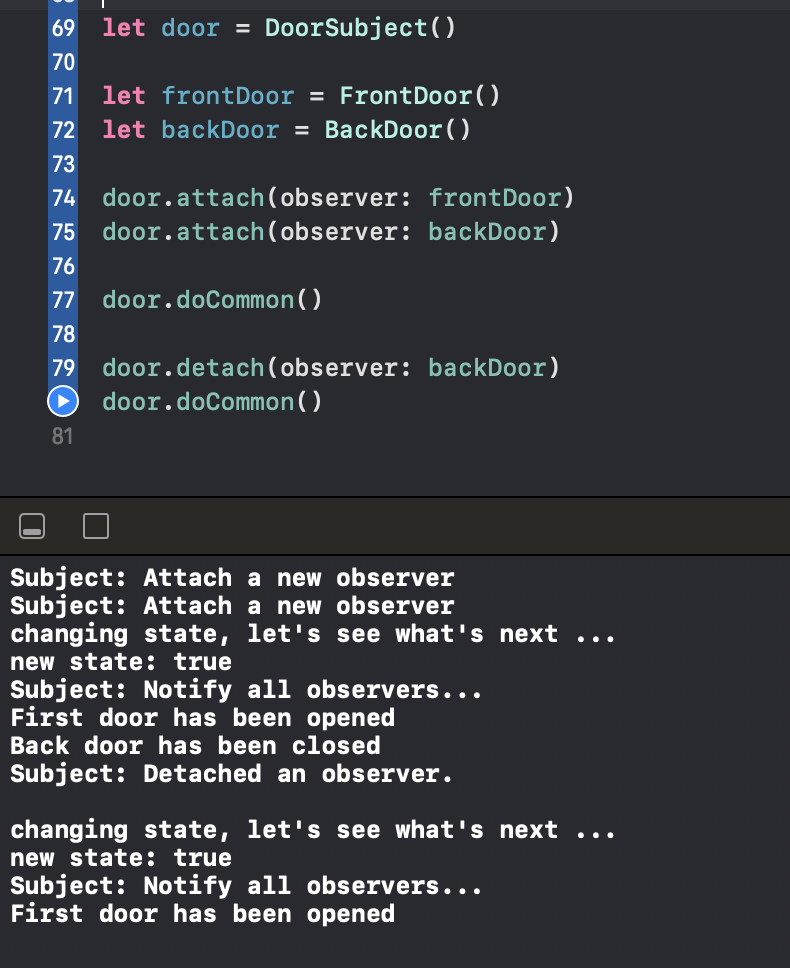
* Concrete Observer



* Concrete Subject



* Output:



1. **NSNotication**

* NSNotificationCenter allows us to send and receive information between classes and/or structs based on an action that has occurred in our app.
* To put it simply, NSNotificationCenter can be thought of as a broadcaster and we can tune into different stations, or channels to listen for any changes.
* To register a class instance as an observer for a notification, you invoke addObserver(\_:selector:name:object:), a method of the NotificationCenter class

**Code example:** in project: NSNotificationCenterDemo