

­­­



과제 04 보고서

모바일 앱 개발 기초반

10413 박희찬

# Observer 패턴

Observer는 관찰자라는 뜻을 가지고 있다. 이름에 나와 있듯이 observer패턴은 관찰을 해서 결과를 나타내는 패턴이다.

어떤 객체의 상태(멤버변수)가 바뀌었음을 관찰하고 바뀌었다면 결과에 영향을 받는 다른 객체들에게 코드를 실행하는 패턴이다.

Observer 패턴은 트리구조와 비슷한 구조를 가지고 있고 이때 부모 노드는 관찰받는 객체, 자식 나드는 관찰에 대한 행위가 실행되는 객체이다.

open class Observable{  
 val observers = *mutableListOf*<Observer>()  
}  
  
abstract class Observer(val parent : Observable){  
 abstract fun upData()  
 var rm = false  
  
 fun addObserver(){  
 parent.observers.add(this)  
 }  
  
 fun rmObserver(){  
 parent.observers.remove(this)  
 }  
}  
  
class Elevator() : Observable() {  
 var now = 1  
  
  
 init {  
 this.now = 1  
 }  
  
 fun up(){  
 now += 1  
  
 *println*("${now}층")  
 for(i in observers){  
 if(!i.rm){  
 i.upData()  
 }  
 }  
 }  
  
 fun down(){  
 if(now < 1) now -1  
  
 *println*("${now}층")  
  
 for(i in observers){  
 if(!i.rm){  
 i.upData()  
 }  
 }  
 }  
}  
  
  
data class Passenger(val destination : Int, val p : Observable) : Observer(p){  
 init {  
 addObserver()  
 }  
  
 override fun upData() {  
 if((parent as Elevator).now == destination){  
 *println*("내립니다.")  
 rm = true  
 } else{  
 *println*("저는 ${destination}층에서 내립니다.")  
 }  
 }  
}  
  
  
fun main(){  
 val elebvator = Elevator()  
 val p1 = Passenger(10, elebvator)  
 val p2 = Passenger(3, elebvator)  
 val p3 = Passenger(5, elebvator)  
 val p4 = Passenger(4, elebvator)  
  
  
// println(elebvator.observers.toString())  
  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
 elebvator.up()  
}

Observer 패턴을 이용해서 엘리베이터 만들기

코드의 문제점은 Observer이 지워지지 않는다. 그래서 rm이라는 Boolean 값을 가져서 지워짐을 표시하였습니다.

# Builder 패턴

Builder 패턴은 객체를 생성할 때 생성자를 사용하지 않고 만들어둔 setter와 getter를 사용해서 객체의 상태를 지정하는 패턴이고, 이때 Builder이라는 클래스에 setter와 getter 만들어주어 상속시켜준다.

open class Car(){  
 protected var kind = ""  
 protected var color = ""  
 protected var name = ""  
 protected var owner = ""  
 protected var carId = 0  
}  
  
abstract class CarBuilder : Car(){  
 abstract fun setKinds()  
 fun setColors(color : String) {  
 this.color = color  
 }  
 fun setNames(name: String) {  
 this.name = name  
 }  
  
 fun setOwners(owner: String) {  
 this.owner = owner  
 }  
  
 fun setCarIDs(carId: Int) {  
 this.carId = carId  
 }  
  
  
 fun getKinds() = kind  
 fun getColors() = color  
 fun getNames() = name  
 fun getOwners() = owner  
 fun getCarIDs() = carId  
}  
  
class Suv() : CarBuilder(){  
 override fun setKinds() {  
 kind = "SUB"  
 }  
}  
  
class Bus() : CarBuilder(){  
 override fun setKinds() {  
 kind = "버스"  
 }  
}  
  
  
fun main(){  
 val car1 = Suv()  
 car1.setKinds()  
 car1.setColors("검정색")  
 car1.setNames("부릉이")  
 car1.setOwners("박희찬")  
 car1.setCarIDs(1189)  
   
 val car2 = Bus()  
 car2.setKinds()  
 car2.setColors("초록색")  
 car2.setNames("타요")  
 car2.setOwners("바키찬")  
 car2.setCarIDs(9999)  
}

# Adapter 패턴

먼저 현실에서 Adapter란 서로 맞지 않는 두 물건을 연결해줄 때 사용한다. C타입 핸드폰에 A타입 충전기로 충전을 하려고할 때 어댑터를 사용하면 충전할 수 있다.

코딩에서 Adapter이란 서로 맞지않는 즉 관계가 없는 인터페이스들을 같이 사용할 수 있다.

두 인터페이스를 같이 상속받는 클래스에서 같은 메서드가 있다면 오버라이딩 하는 과정에서 에러가 발생할것이다. 이때는 하나의 인터페이스만 상속시켜주고, 다른 인터페이스는 생성자로 받아서 해결한다. 그리고 두 인터페이스의 메서드중 하나는 예외로 처리한다.

interface EDCAN{  
 fun coding(language : String)  
 fun design()  
 fun android()  
 fun web()  
 fun IoT()  
 fun game()  
  
 fun noSleep(time : Int)  
 fun otaku()  
}  
  
class EDCANPeople : EDCAN{  
 override fun coding(language: String) {  
 *println*("$language 언어 코딩중")  
 }  
  
 override fun design() {  
 *println*("에드캔 디자인은 세계 제일")  
 }  
  
 override fun android() {  
 *println*("EDCAN은 안드로이드를 하지")  
 }  
  
 override fun web() {  
 *println*("EDCAN은 웹도 하지")  
 }  
  
 override fun IoT() {  
 *println*("EDCAN은 IoT도 했지")  
 }  
  
 override fun game() {  
 *println*("EDCAN은 게임도 했지")  
 }  
  
 override fun noSleep(time: Int) {  
 *println*("EDCAN이 잠을 잔다?")  
 }  
  
 override fun otaku() {  
 *println*("사실 에드캔은 씹덕 동아리 였습니다")  
 }  
}  
  
interface AppplePi{  
 fun coding(language : String)  
 fun design()  
 fun android()  
  
 fun firebase()  
 fun baking()  
}  
  
class AppplePiPeople : AppplePi{  
 override fun coding(language: String) {  
 *println*("$language 언어 코딩중")  
 }  
  
 override fun design() {  
 *println*("애플파이 디자인은 세계 제일")  
 }  
  
 override fun android() {  
 *println*("애플파이는 안드로이드 개발 동아리 입니다.")  
 }  
  
 override fun firebase() {  
 *println*("애플파이에서는 파이어베이스도 배워요")  
 }  
  
 override fun baking() {  
 *println*("맛있는 애플파이를 굽고있어요!")  
 }  
}  
  
class EdcanplePi(val appPlePiPeople : AppplePiPeople) : EDCAN{  
  
 override fun coding(language: String) {  
 *println*("$language 언어 코딩중")  
 }  
  
 override fun design() {  
 throw Exception()  
 }  
  
 override fun android() {  
 throw Exception()  
 }  
  
 override fun web() {  
 *println*("EDCAN:ple Pi는 웹을 하지")  
 }  
  
 override fun IoT() {  
 *println*("EDCAN:ple Pi는 IoT도 했지")  
 }  
  
 override fun game() {  
 *println*("EDCAN:ple Pi는 게임도 했지")  
 }  
  
 override fun noSleep(time: Int) {  
 println("EDCAN:ple Pi는 잠을 잔다?")  
 }  
  
 override fun otaku() {  
 *println*("사실 여기도 씹덕 동아리 였습니다")  
 }  
}  
  
fun main() {  
 val student1 = EDCANPeople()  
 val student2 = AppplePiPeople()  
 val student3 = EdcanplePi(AppplePiPeople())  
  
 student1.coding("Kotlin")  
 student1.design()  
 student1.android()  
 student1.web()  
 student1.IoT()  
 student1.game()  
 student1.otaku()  
  
 student2.coding("JAVA")  
 student2.design()  
 student2.android()  
 student2.firebase()  
 student2.baking()  
  
  
 student3.coding("C")  
 try{  
 student3.design()  
 } catch (e : Exception){  
 *println*("디자인은 애플파이에서 담당합니다.")  
 }  
  
 try{  
 student3.android()  
 } catch (e : Exception){  
 *println*("안드로이드는 애플파이에서 담당합니다.")  
 }  
  
   
 student3.web()  
 student3.IoT()  
 student3.game()  
 student3.otaku()  
 student3.appPlePiPeople.coding("C")  
 student3.appPlePiPeople.design()  
 student3.appPlePiPeople.android()  
 student3.appPlePiPeople.baking()  
}