Patrones de Diseño de Software

Ing. Román Arturo Pineda Soto
roman.pineda@unitec.edu
@romanpineda

Patrones de Diseño de Software

- Incrementan la eficiencia de los programadores.
- Proveen una guía de desarrollo de Software.
- No son una única fuente de verdad, pero si asemejan a mejores prácticas.

Principales patrones de Diseño

Singleton:

Limita la creación de una clase a únicamente un objeto.

Es útil para objetos de control para coordinar acciones a lo largo de un sistema.

Se estila que la definición sea de tipo private

```
public class EagerSingleton {
    private static volatile EagerSingleton instance = new
EagerSingleton();

    // private constructor
    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

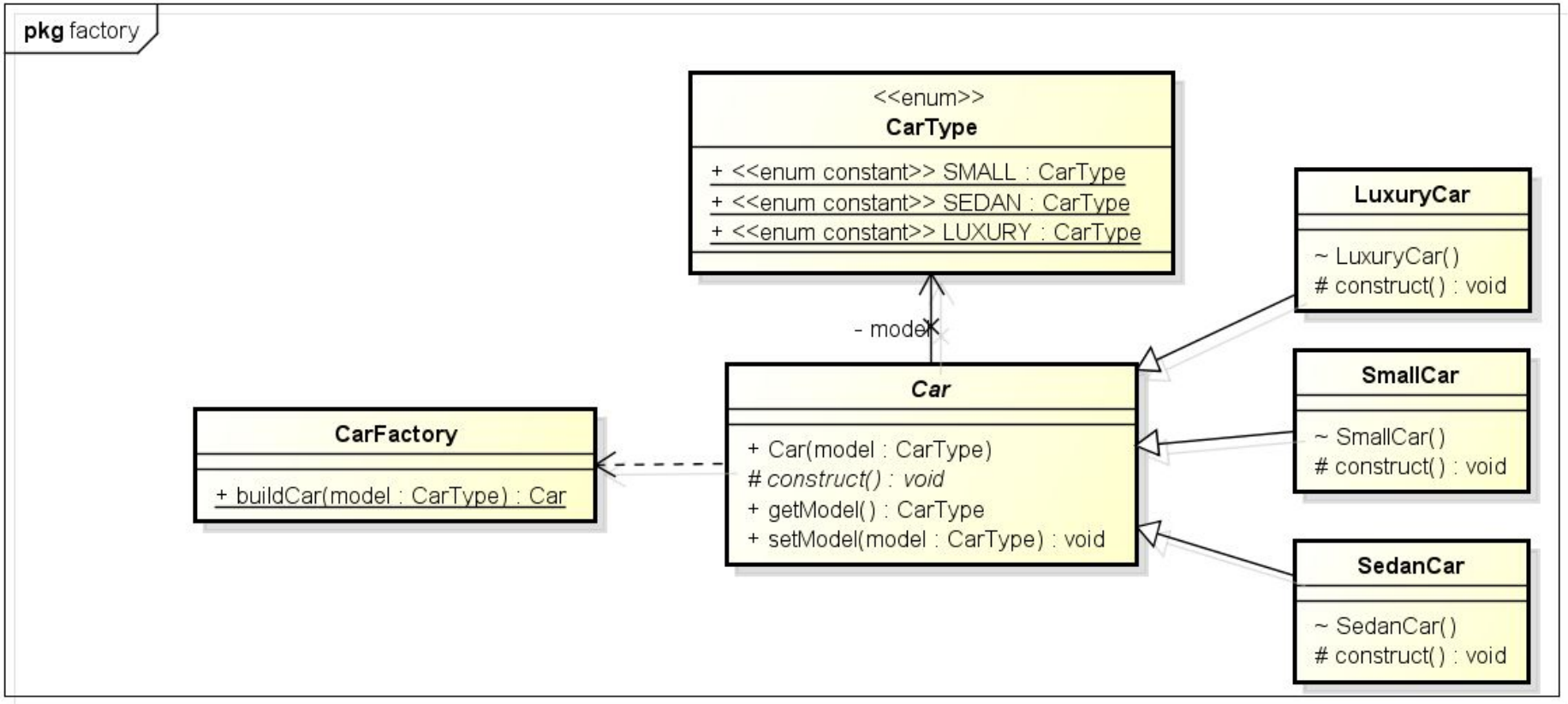
Principales patrones de Diseño

Factory:

Así como una fábrica produce bienes, una fábrica de desarrollo produce objetos.

Para instanciar un objeto, se llama a la fábrica en lugar de a su constructor.

SomeClass someClassObject = new SomeClass();

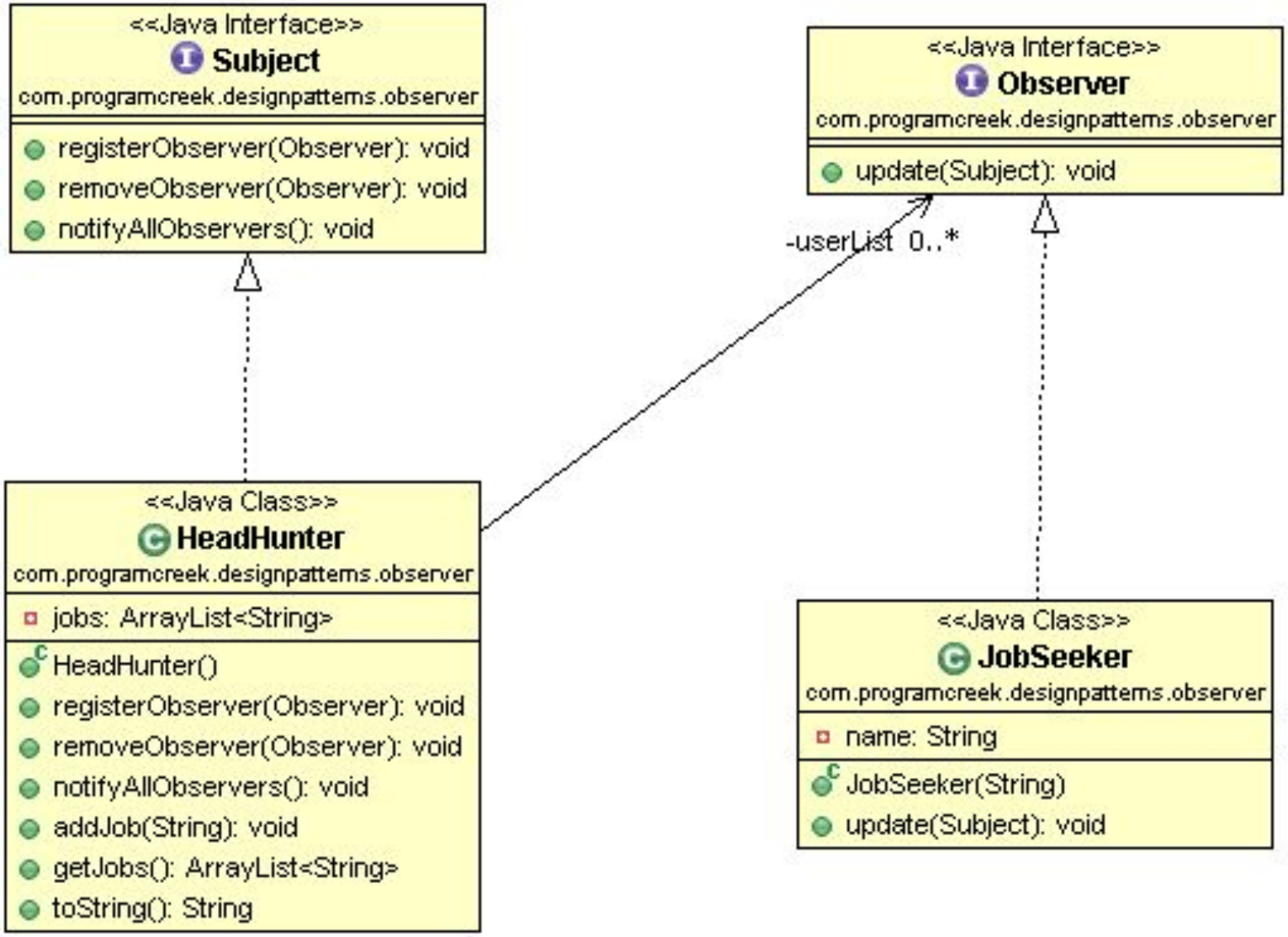

pkg factory
<<enum>>
CarType
+ <<enum constant>> SMALL : CarType
+ <<enum constant>> SEDAN : CarType
+ <<enum constant>> LUXURY : CarType
- model
Car
+ Car(model : CarType)
# construct() : void
+ getModel() : CarType
+ setModel(model : CarType) : void
CarFactory
+ buildCar(model : CarType) : Car
LuxuryCar
~ LuxuryCar()
# construct() : void
SmallCar
~ SmallCar()
# construct() : void
SedanCar
~ SedanCar()
# construct() : void

Principales patrones de Diseño

Observer:

Es una dependencia uno a muchos, cuando un objeto cambia de estado, se notifica a los demás usualmente llamando a uno de sus métodos.

Un sujeto puede tener varios observadores, imagine el caso de Twitter.

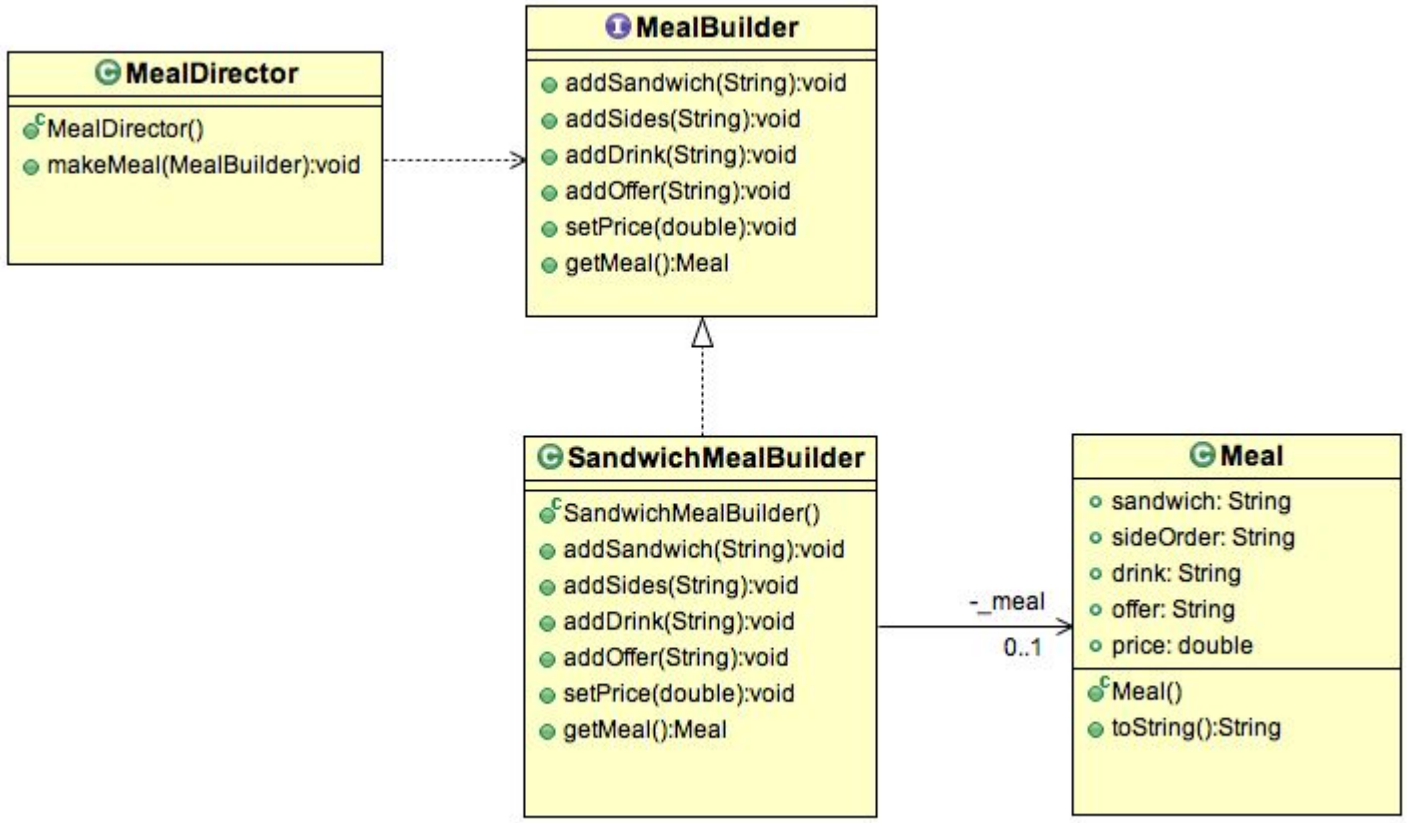

<<Java Interface>>
Subject
com.programcreek.designpatterns.observer
registerObserver(Observer): void
removeObserver(Observer): void
notifyAllObservers(): void
<<Java Interface>>
Observer
com.programcreek.designpatterns.observer
update(Subject): void
-userList 0..*
<<Java Class>>
HeadHunter
com.programcreek.designpatterns.observer
jobs: ArrayList<String>
HeadHunter()
registerObserver(Observer): void
removeObserver(Observer): void
notifyAllObservers(): void
addJob(String): void
getJobs(): ArrayList<String>
toString(): String
<<Java Class>>
JobSeeker
com.programcreek.designpatterns.observer
name: String
JobSeeker(String)
update(Subject): void

Principales patrones de Diseño

Builder:

Este patrón de diseño es usado para construir objetos se construyen objetos complejos con relación de agregación o composición.

Se crea el objeto “paso a paso” a diferencia del patrón de Factory que se crea de una vez.

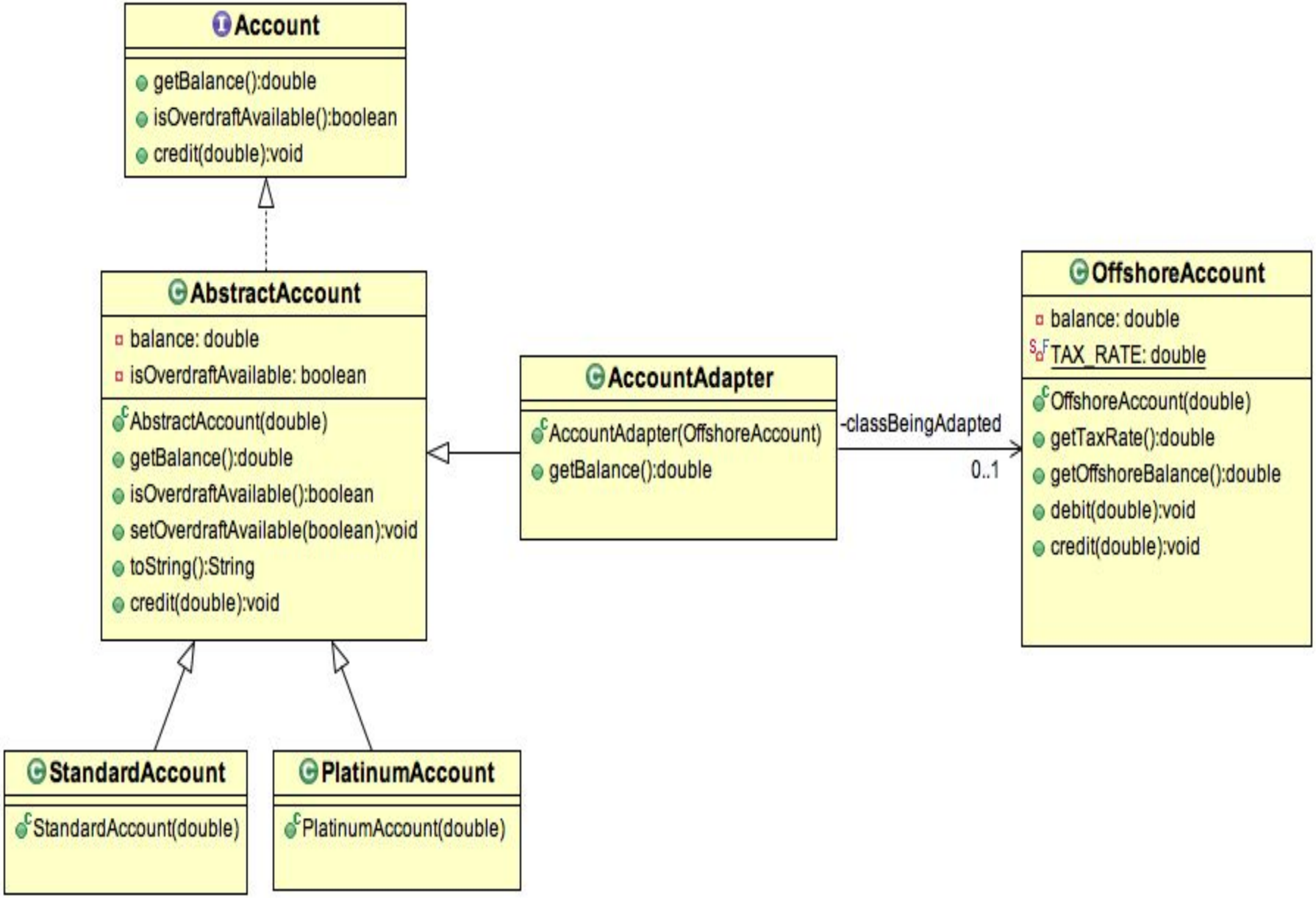

MealDirector
MealDirector()
makeMeal(MealBuilder):void
MealBuilder
addSandwich(String):void
addSides(String):void
addDrink(String):void
addOffer(String):void
setPrice(double):void
getMeal():Meal
SandwichMealBuilder
SandwichMealBuilder()
addSandwich(String):void
addSides(String):void
addDrink(String):void
addOffer(String):void
setPrice(double):void
getMeal():Meal
-_meal
0..1
Meal
sandwich: String
sideOrder: String
drink: String
offer: String
price: double
Meal()
toString():String

Principales patrones de Diseño

Adapter:

Permite a clases que son incompatibles, interactuar unas con otras realizando una convención de interface de una clase en la otra.

Usualmente es necesario un interpretador para realizar la conexión.


Account
getBalance():double
isOverdraftAvailable():boolean
credit(double):void
AbstractAccount
balance: double
isOverdraftAvailable: boolean
AbstractAccount(double)
getBalance():double
isOverdraftAvailable():boolean
setOverdraftAvailable(boolean):void
toString():String
credit(double):void
AccountAdapter
AccountAdapter(OffshoreAccount)
getBalance():double
-classBeingAdapted
0..1
OffshoreAccount
balance: double
TAX_RATE: double
OffshoreAccount(double)
getTaxRate():double
getOffshoreBalance():double
debit(double):void
credit(double):void
StandardAccount
StandardAccount(double)
PlatinumAccount
PlatinumAccount(double)

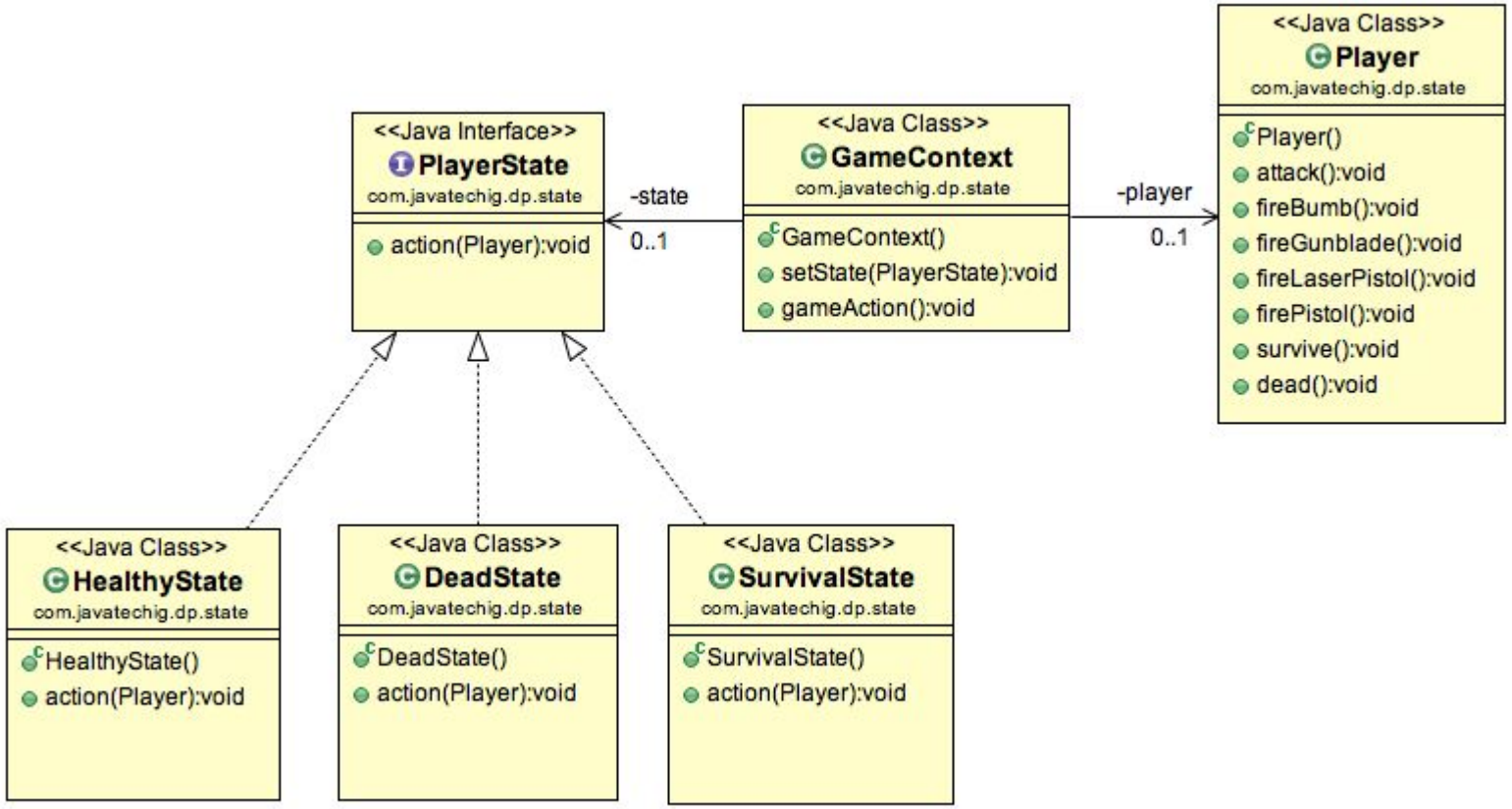
Principales patrones de Diseño

State:

Encapsula varios estados en los que una máquina se puede encontrar y permite al objeto alterar su comportamiento en estados intermedios.

La máquina o el contexto es llamado pattern-speak y puede contar con acciones tomadas en cada estado.

Sin el uso de este patrón, el código puede llenarse de basura y de muchos if/else


<<Java Interface>>
PlayerState
com.javatechig.dp.state
action(Player):void
-state
0..1
<<Java Class>>
GameContext
com.javatechig.dp.state
GameContext()
setState(PlayerState):void
gameAction():void
-player
0..1
<<Java Class>>
Player
com.javatechig.dp.state
Player()
attack():void
fireBumb():void
fireGunblade():void
fireLaserPistol():void
firePistol():void
survive():void
dead():void
<<Java Class>>
HealthyState
com.javatechig.dp.state
HealthyState()
action(Player):void
<<Java Class>>
DeadState
com.javatechig.dp.state
DeadState()
action(Player):void
<<Java Class>>
SurvivalState
com.javatechig.dp.state
SurvivalState()
action(Player):void


THANK YOU FOR YOUR TIME
quickmeme.com