Control 2

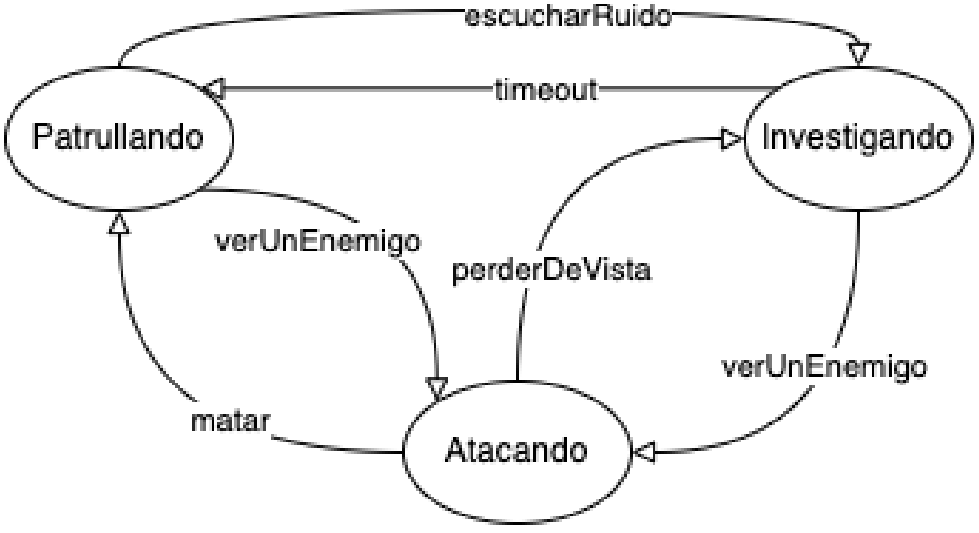
Profesores: Nancy Hitshfeld y Matías Toro
Auxiliares: Beatriz Graboloza, Daniel Ramírez e Ignacio Slater
Con apuntes. Respuestas en hojas separadas.

P1. Un guardia de seguridad en un videojuego puede encontrarse en tres estados: **Patrullando**, **Investigando** y **Atacando**. Inicialmente, el guardia se encuentra en el estado de Patrullando. Cuando ocurre un evento de **escucharUnRuido**, el guardia cambia al estado de Investigando. Si se produce un evento de **tiempoAgotado** mientras está en el estado de Investigando, el guardia regresa al estado de Patrullando. Tanto desde el estado de Patrullando como desde el estado de Investigando, si se presenta un evento de **verUnEnemigo**, el guardia cambia al estado de Atacando. Por otro lado, desde el estado de Atacando, si el guardia presenta el evento **perderDeVista**, vuelve al estado de Investigando, mientras que si logra **neutralizar** a un enemigo, retorna al estado de Patrullando.

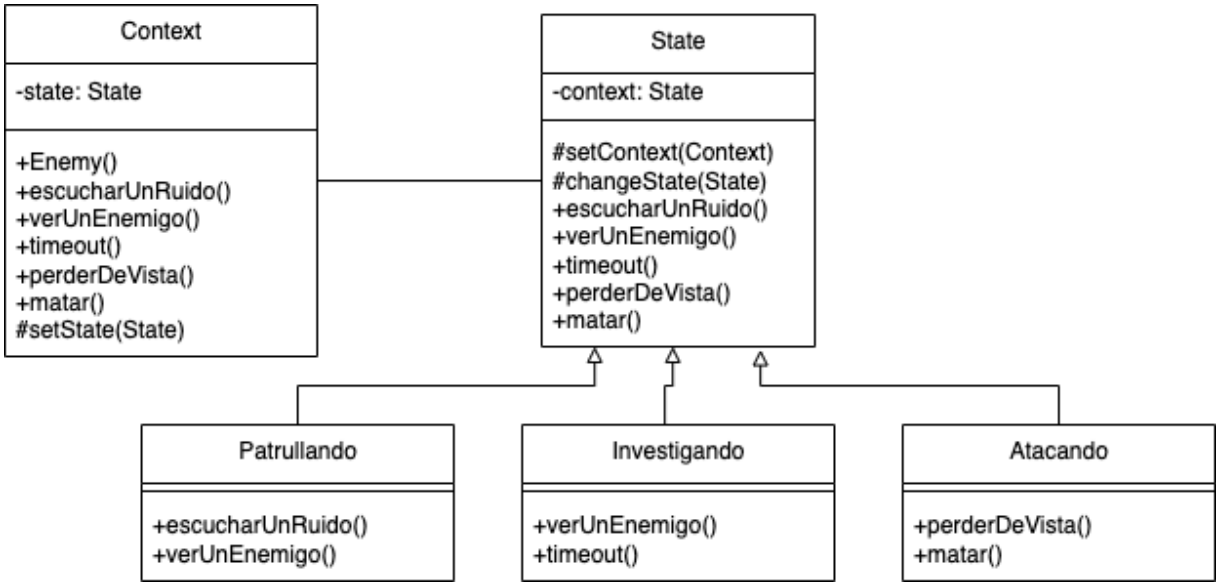
a) (0.3 pts) Dibuje el diagrama de estados.
b) (0,9 pts) Use el patrón de diseño State para modelar el problema. Para esto haga el diagrama de clases completo e incluya en cada clase los métodos que correspondan.
c) (0,8 pts) Implemente los métodos del estado Investigando y de la clase que representa Contexto.

Sol:

a)
- $-0.05 \times$ falta/sobra algún estado
- $-0.025 \times$ falta/sobra alguna transición



b)
- $-0.09 \times$ falta alguna clase
- $-0.011 \times$ falta algún método o variable
- $-0.011 \times$ modificador de acceso incorrecto



c)

```
// -0.4 Si no existe la clase Investigando
// -0.3 Si existe y no extiende de State
class Investigando extends State {
   // -0.2 Si no existe o está mal implementado
   override def verUnEnemigo() = {
       changeState(new Atacando())
   }
   // -0.2 Si no existe o está mal implementado
   override def timeout() = {
       changeState(new Patrullando())
   }
}

// -0.4 Si no existe la clase
class Guardia { //Context
  // -0.1 si no existe la variable
  // -0.05 si la variable no es privada
  // -0.05 si no empieza en estado Patrullando
   private val state: State = new Patrullando()

   // -0.1 si no existe o está mal implementado
   def setState(state: State) = {
       this.state = state
       this.state.setContext(this)
   }

   // -0.05 x método faltante o mal implementado
   def escucharUnRuido() = {state.escucharUnRuido()}
   def tiempoAgotado() = {state.tiempoAgotado()}
```

```
    def verUnEnemigo() = {state.verUnEnemigo()}
    def perderDeVista() = {state.perderDeVista()}
    def neutralizar() = {state.neutralizar()}
}
```

P2. Suponga que está trabajando con un lenguaje de programación en el cual las funciones no son valores y, por ende, no pueden ser pasadas como argumentos. Afortunadamente, se da cuenta de que para resolver esto puede modelar funciones utilizando clases. A continuación se presenta una especificación para funciones genéricas de 1 argumento: una función puede ser aplicada (**apply**) a valores del tipo del dominio y siempre devuelve valores del tipo del codominio.

```
trait MyFun[-D,+C]{
  def apply(x: D): C
}
```

a) (0,8 pts) Implemente las clases concretas **SumaUno** y **Reciproco** que extiendan de la especificación de funciones mencionada anteriormente. El recíproco de un número es el resultado de dividir 1 entre ese número.

b) (0,4 pts) Justifique la varianza escribiendo contra ejemplos.

c) (0,8 pts) Escriba una función genérica $compose(x, f, g)$, que aplica x a f, y el resultado a g. Muestre un ejemplo de uso con SumaUno y Reciproco.

Sol:

a)
```
// -0.4 Si no existe la clase
// -0.3 Si no se extiende de MyFun
// -0.2 Si los generics [Int, Int] están incorrectos
class SumaUno extends MyFun[Int, Int]{
  def apply(x: Int) = x + 1  // -0.2 si está mal implementado
}
// -0.4 Si no existe la clase
// -0.3 Si no se extiende de MyFun
// -0.2 Si los generics [Int, Double] están incorrectos
class Reciproco extends MyFun[Int, Double]{
  // -0.2 si está mal implementado
  def apply(x: Int): Double = {
    // -0.1 si no se considera este caso
    if(x==0) throw new Exception("Division by zero!")
    else 1.0/x
  }
}
```

b) El puntaje de esta pregunta es todo o nada uwu

Si permitimos covarianza en el argumento, poodemos definir una función de la siguiente forma que se va a quedar pegada o dará error:

```
def foo(f: MyFun[Object, Int]) = {
  f("hola")
}
foo(new Reciproco())
```

También pueden argumentar con palabras usando el mismo argumento visto en clases usando funciones: si permites covarianza en el argumento puedes aplicar el argumento con un valor subtipo del dominio pero que no tiene nada que ver con el de la función que se proveyó (A <: B y C < : B).

c)

```
// -0.6 si no existe la función
// -0.2 si están mal los generics
def compose[A,B,C](x: A, f: MyFun[A,B], g: MyFun[B,C]): C = {
g(f(x))  // -0.2 si estp está mal implementado
}
// -0.2 si falta el ejemplo
def main(args: Array[String]): Unit = {
val f = new SumaUno
val g = new Reciproco
println(compose(9, f,g))
}
```

P3. Una tienda cuenta con diferentes tipos de productos: teléfonos, televisores y computadoras. Un inventario está conformado por una lista de productos. La tienda necesita contar la cantidad de productos por tipo, y buscar productos por tipo y nombre. Su solución debe ser extensible de manera que agregar nuevas funcionalidades debería tener bajo costo (no deberían tener que modificar las clases que representan productos).

a) (0,9 pts) Dibuje un diagrama de clases UML de su solución.
b) (0,3 pts) Indique qué patrón de diseño utilizó. Justifique su elección.
c) (0,8 pts) Agregue la funcionalidad de imprimir el tipo de sus productos. Para ello extienda su diagrama e implemente las clases necesarias.

Sol:

a)
- -0.033 falta interfaz para los productos
- $-0.033 \times$ falta tipo de producto
- -0.133 productos no tienen nombre (debe estar en la interfaz)
- -0.1 no existe método `accept(Visitor)` (debe estar en la interfaz y sobrescribirse en las subclases)
- $-0.1 \times$ falta método `visitX(X)` para producto `X` (no debe hacerse con overloading)
- -0.133 falta interfaz para visitors
- $-0.133 \times$ falta tipo de visitor (uno para contar y otro para buscar)
- -0.05 no utiliza una clase abstracta para los tipos de productos

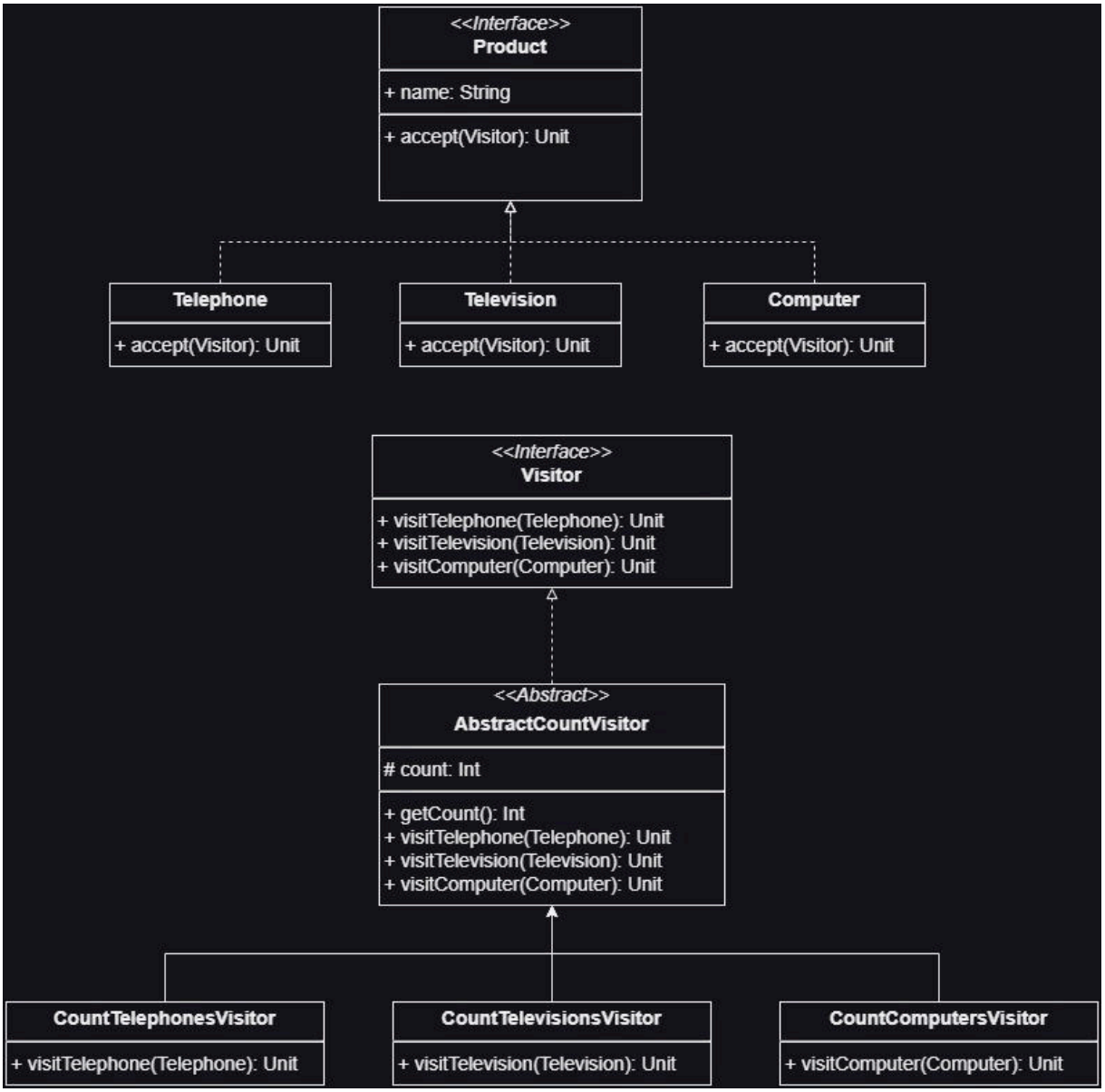


Figura 1: Faltan las clases para buscar, pero son equivalentes a las de contar

b) (Todo o nada) Visitor porque agregar operaciones nuevas tiene bajo costo

c)
- 0.4 por agregar el nuevo visitor al uml
- 0.4 por implementar

```
class PrintVisitor extends Visitor {
  override def visitTelephone(t: Telephone): Unit = {
    s"Telephone"
  }
  override def visitTelevision(t: Television): Unit = {
    s"Television"
  }
```


```
    override def visitComputer(t: Computer): Unit = {
      s"Computer"
    }
  }
```