Laboratorul 8: Design patterns în Kotlin

Introducere

Se recomandă parcurgerea următoarelor resurse:

- **„Design Patterns Elements of Reusable Object-Oriented Software”** scrisă de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides
- **„Hands-on design patterns with Kotlin”** - Alexey Soshin
- „Kotlin Programming: The big nerd ranch guide” - Josh Skeen și David Greenhalgh
- **Exemplele din cursul 8 de la disciplina *Paradigme de programare***

Modelul Stare (State)

Scopul modelului: Permite unui obiect să își modifice comportamentul atunci când starea sa internă se schimbă. Obiectul va părea că își schimbă clasa.

Aplicabilitate: Modelul *stare* se utilizează în următoarele cazuri:

- Comportamentul unui obiect depinde de starea sa și trebuie să-și schimbe comportamentul în timpul execuției, în funcție de starea respectivă;
- Operațiile au declarații condiționale compuse, mari, care depind de starea obiectului.

Consecințe:

- Localizează un comportament specific stării și comportamentul partițiilor pentru diferite stări;
- Face tranzițiile între stări explicite;
- Obiectele stării pot fi partajate.

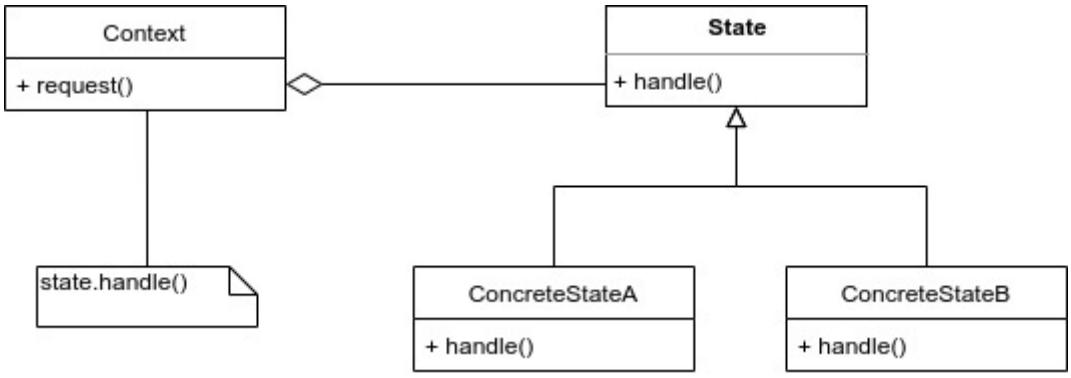


Diagrama de clase pentru modelul *Stare*

Vezi și exemplul cu automatul finit de stări din cursul 8.

Implementarea modelului Stare

Pentru a implementa modelul *Stare*, există două abordări:

1. **Abordarea clasică**

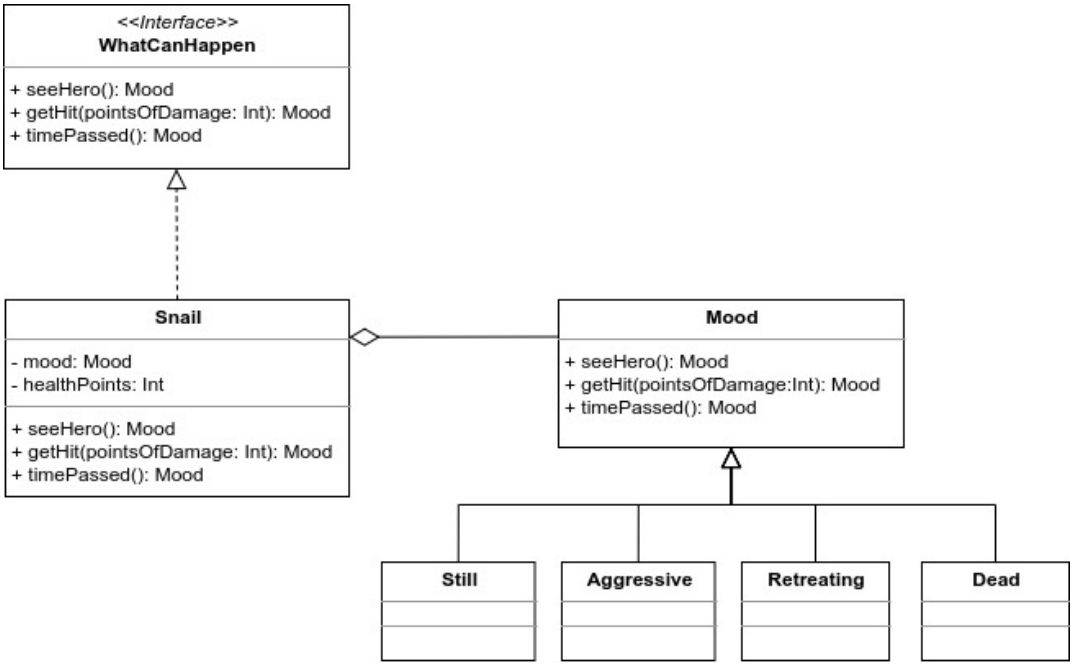


Diagrama de clase pentru implementarea clasică a modelului Stare

Observație: Această abordare necesită gestionarea fiecărei stări într-un switch din cadrul fiecărei metode din clasa *Snail* (seeHero, getHit, timePassed).

2. **Abordarea „smart”**

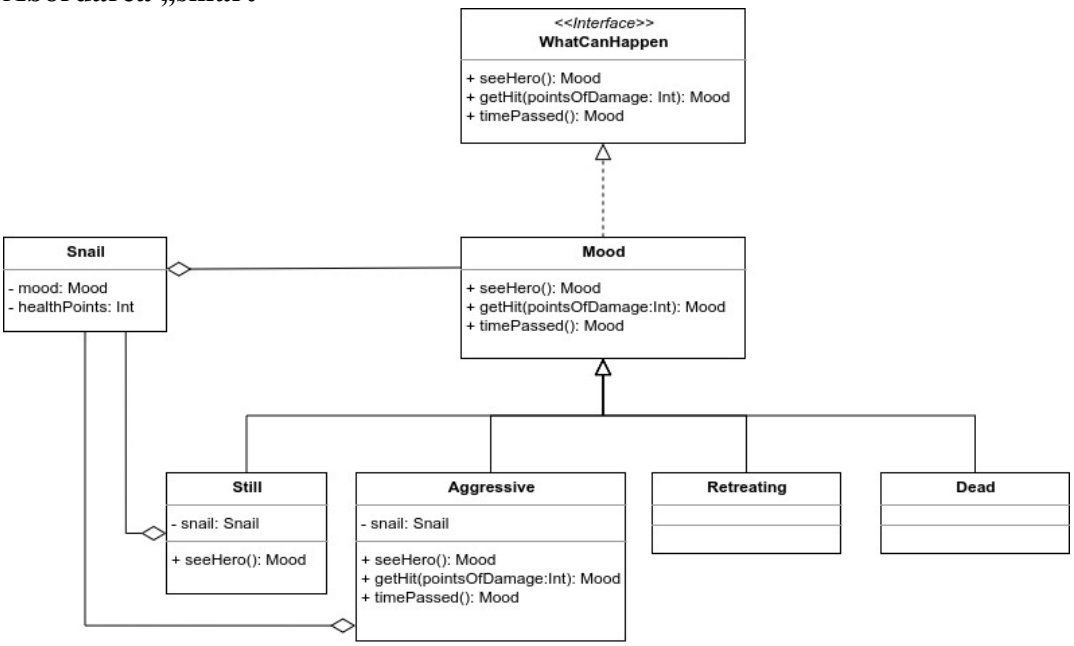


Diagrama de clase pentru implementarea „smart” a modelului Stare

Observație: Această abordare permite adăugarea unor stări adiționale fără a fi nevoie de modificarea codului deja scris. Fiecare stare își va defini comportamentul propriu.

```
package com.pp.laborator

fun main(args: Array<String>) {
    val snail = Snail()
    val still = Still(snail)

    val a = still.seeHero()
    println(a is Still)
    println(a is Aggressive)

    val b = still.timePassed()
    println(b is Still)
}

class Snail {
    internal var mood: Mood = Still(this)
    private var healthPoints = 10
}

interface WhatCanHappen {
    fun seeHero(): Mood
    fun getHit(pointsOfDamage: Int): Mood
    fun timePassed(): Mood
}

sealed class Mood : WhatCanHappen {
    override fun seeHero() = this
    override fun getHit(pointsOfDamage: Int) = this
    override fun timePassed() = this
}

class Still(private val snail: Snail) : Mood() {
    override fun seeHero(): Mood {
        return snail.mood.run {
            Aggressive(snail)
        }
    }
}

class Aggressive(snail: Snail) : Mood() {
    override fun seeHero(): Mood {
        TODO("not implemented")
    }

    override fun getHit(pointsOfDamage: Int): Mood {
        TODO("not implemented")
    }

    override fun timePassed(): Mood {
        TODO("not implemented")
    }
}

class Retreating : Mood()

class Dead : Mood()
```

Exemple

Exemplu de combinare a modelelor Composite și Command

Cerință: Să se proiecteze și să se implementeze un meniu de editare minimalist, utilizând modelul *Composite* pentru a obține structura arborescentă și modelul *Command* pentru a furniza o funcționalitate de afișare a clasei/funcției curente fiecărui element din meniu.

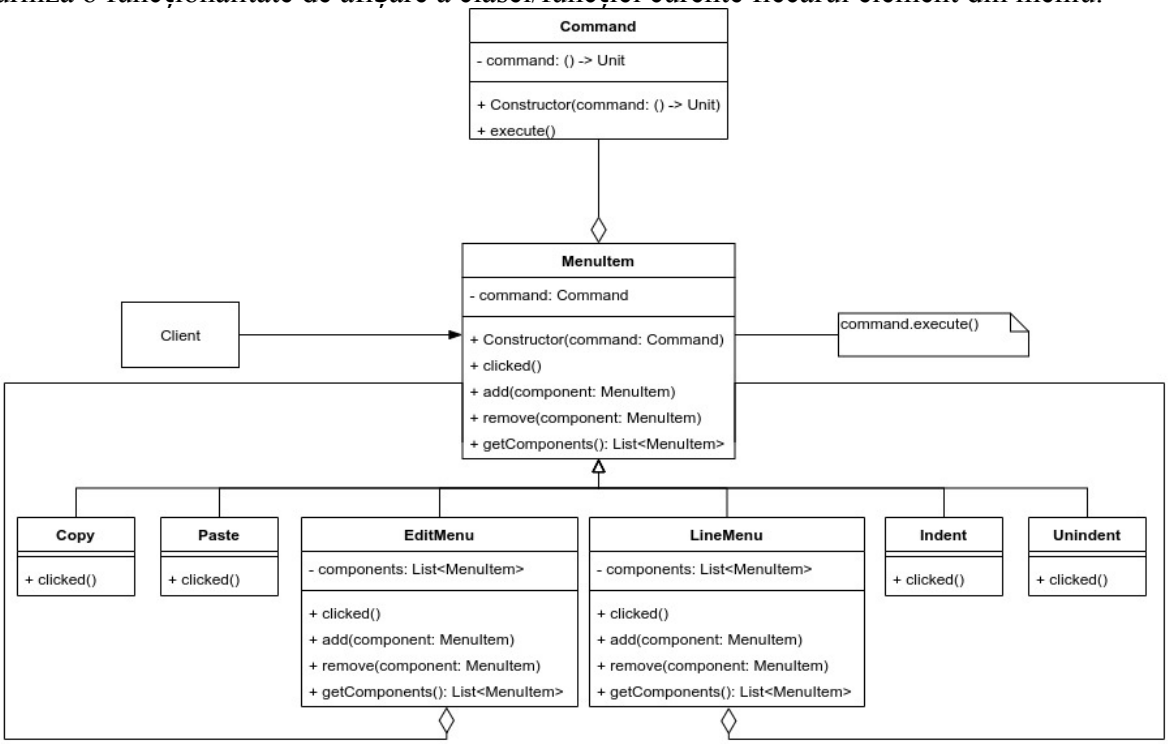
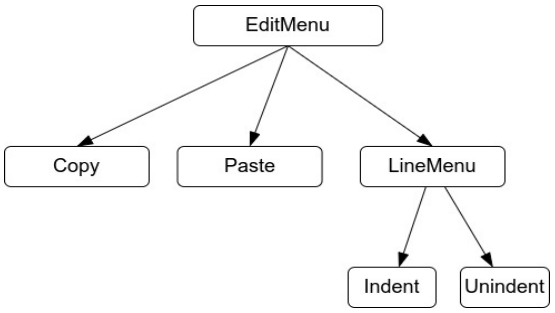


Diagrama de clase



Ierarhia meniului de editare

Observație: Nu este necesară crearea unui proiect Maven/Gradle. Se poate crea un proiect Kotlin/JVM, iar la *Project SDK* se va specifica Java 1.8.

Codul sursă

- **Command.kt**

```
class Command(private val command: ()->Unit) {
    fun execute() {
        command()
    }
}
```

- **MenuItem.kt**

```
open class MenuItem(val command: Command) {
    open fun clicked() {
        command.execute()
    }
    open fun add(component: MenuItem) {}
    open fun remove(component: MenuItem) {}
    open fun getComponents(): List<MenuItem> = TODO()
}
```

- **Copy.kt**

```
class Copy: MenuItem(Command{println("Sunt in clasa Copy")}) {
    override fun clicked() {
        this.command.execute()
        println("Copy: clicked()")
    }
}
```

- **Paste.kt**

```
class Paste: MenuItem(Command{println("Sunt in clasa Paste")}) {
    override fun clicked() {
        this.command.execute()
        println("Paste: clicked()")
    }
}
```

- **Indent.kt**

```
class Indent: MenuItem(Command{println("Sunt in clasa Indent")}) {
    override fun clicked() {
        this.command.execute()
        println("Indent: clicked()")
    }
}
```

- **Unindent.kt**

```
class Unindent: MenuItem(Command{println("Sunt in clasa Unindent")}) {
    override fun clicked() {
        this.command.execute()
        println("Unindent: clicked()")
    }
}
```

- **EditMenu.kt**

```
class EditMenu: MenuItem(Command{println("Sunt in clasa EditMenu")}) {
    private var components: MutableList<MenuItem> = mutableListOf()

    override fun clicked() {
        this.command.execute()
        println("EditMenu: clicked()")
    }

    override fun add(component: MenuItem) {
```

```
        components.add(component)
    }

    override fun remove(component: MenuItem) {
        components.remove(component)
    }

    override fun getComponents(): MutableList<MenuItem> {
        return components
    }
}
```

- **LineMenu.kt**

```
class LineMenu: MenuItem(Command{println("Sunt in clasa LineMenu")}) {
    private var components: MutableList<MenuItem> = mutableListOf()

    override fun clicked() {
        this.command.execute()
        println("LineMenu: clicked()")
    }

    override fun add(component: MenuItem) {
        components.add(component)
    }

    override fun remove(component: MenuItem) {
        components.remove(component)
    }

    override fun getComponents(): MutableList<MenuItem> {
        return components
    }
}
```

- **Main.kt**

```
fun main(args: Array<String>) {
    var editMenu = EditMenu()
    var copy = Copy()
    var paste = Paste()

    var lineMenu = LineMenu()
    var indent = Indent()
    var unindent = Unindent()

    lineMenu.add(indent)
    lineMenu.add((unindent))

    editMenu.add(copy)
    editMenu.add(paste)
    editMenu.add(lineMenu)

    editMenu.clicked()
    editMenu.getComponents().forEach{ it.clicked() }
    lineMenu.getComponents().forEach{ it.clicked() }
}
```

Aplicații și teme

Aplicații de laborator:

1) Să se proiecteze și să se implementeze un *lanț de responsabilități* **dublu** (similar unei liste dublu înlănțuite). Scopul duplicării lanțului este de a trimite un răspuns către handler-ul ierarhic superior la finalul procesării unei cereri, pentru a-l anunța că sarcina a fost încheiată cu succes (sau nu). Pentru instanțierea *Handler-elor*, se va utiliza modelul *fabrică abstractă*, care va crea două fabrici:

- *EliteFactory* ce permite crearea *CEOHandler, ExecutiveHandler* și *ManagerHandler*;
- *HappyWorkerFactory* ce permite crearea *HappyWorkerHandler;*

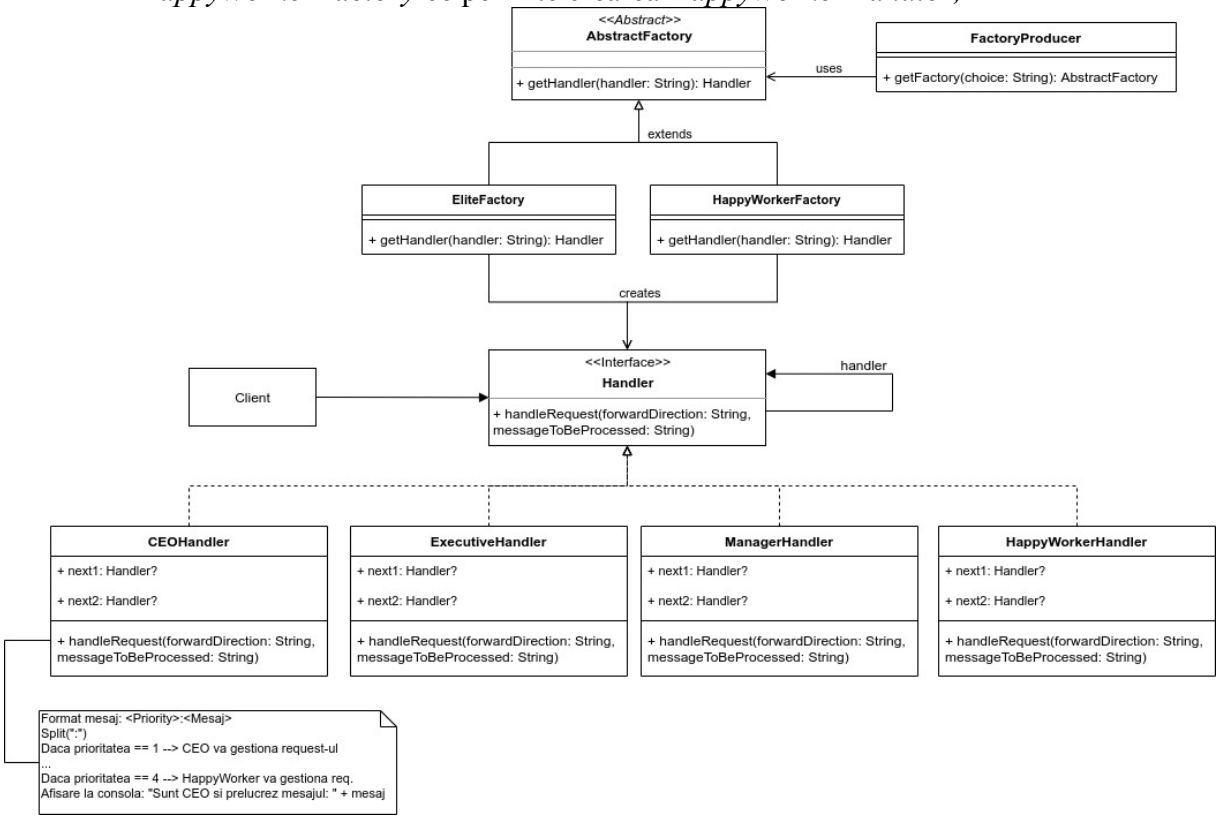


Diagrama de clase

Se remarcă faptul că diagrama de clase, deși oferă o perspectivă de ansamblu asupra structurii aplicației, nu specifică cum vor fi instanțiate obiectele, legăturile dintre ele sau câte obiecte vor fi create. Așadar, este necesară o diagramă de obiecte.

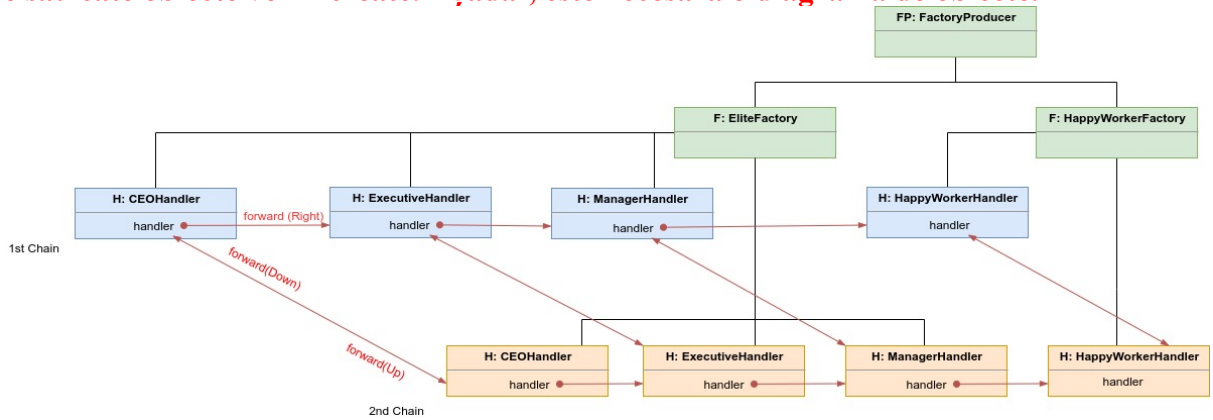


Diagrama de obiecte

Se dă următorul schelet al aplicației:

- **Handler.kt**

```
package chain

interface Handler {
    fun handleRequest(forwardDirection: String, messageToBeProcessed:
String)
}
```

- **CEOHandler.kt**

```
package chain

class CEOHandler(var next1: Handler? = null, var next2: Handler? =
null): Handler {
    override fun handleRequest(forwardDirection: String,
messageToBeProcessed: String) {
        TODO()
    }
}
```

- **ExecutiveHandler.kt**

```
package chain

class ExecutiveHandler(var next1: Handler? = null, var next2: Handler?
= null): Handler {
    override fun handleRequest(forwardDirection: String,
messageToBeProcessed: String) {
        TODO()
    }
}
```

- **ManagerHandler.kt**

```
package chain

class ManagerHandler(var next1: Handler? = null, var next2: Handler? =
null): Handler {
    override fun handleRequest(forwardDirection: String,
messageToBeProcessed: String) {
        TODO()
    }
}
```

- **HappyWorkerHandler.kt**

```
package chain

class HappyWorkerHandler(var next1: Handler? = null, var next2:
Handler? = null): Handler {
    override fun handleRequest(forwardDirection: String,
messageToBeProcessed: String) {
        TODO()
```

```
    }
}
```

- **AbstractFactory.kt**

```
package factory

import chain.Handler

abstract class AbstractFactory {
    abstract fun getHandler(handler: String): Handler
}
```

- **EliteFactory.kt**

```
package factory

import chain.Handler

class EliteFactory: AbstractFactory() {
    override fun getHandler(handler: String): Handler {
        TODO()
    }
}
```

- **HappyWorkerFactory.kt**

```
package factory

import chain.Handler

class HappyWorkerFactory: AbstractFactory() {
    override fun getHandler(handler: String): Handler {
        TODO()
    }
}
```

- **FactoryProducer.kt**

```
package factory

class FactoryProducer {
    fun getFactory(choice: String): AbstractFactory {
        TODO()
    }
}
```

- **Main.kt**

```
fun main(args: Array<String>) {
    TODO()
    // se creeaza 1xFactoryProducer, 1xEliteFactory,
1xHappyWorkerFactory
    //...

    // crearea instantelor (prin intermediul celor 2 fabrici):
    // 2xCEOHandler, 2xExecutiveHandler, 2xManagerHandler,
2xHappyWorkerHandler
```

```
    //...

    // se construieste lantul (se verifica intai diagrama de obiecte
si se realizeaza legaturile)
    //...

    // se executa lantul utilizand atat mesaje de prioritate diferita,
cat si directii diferite in lant
    //...
}
```

2) Să se proiecteze și să se implementeze o aplicație care citește dintr-un fișier un text de tipul *lorem ipsum*, „sparge" textul în cuvinte, pentru fiecare cuvânt fiind necesară o actualizare de stare în modelul *Memento*. În plus, se va utiliza modelul *Observer* pentru a urmări modificările efectuate asupra stării. Există două tipuri de observatori:

- *SmallWordObserver* - va urmări cuvinte de lungime mai mică sau egală cu 7
- *LargeWordObserver* - va urmări cuvinte de lungime > 7

Prin intermediul modelului *Memento,* fiecare observer poate restaura o stare anterioară. La fiecare 10 cuvinte mici afișate de metoda update din *SmallWordConsumer*, se va restaura starea *savedStates[savedStates.size % 10]*. La fiecare 7 cuvinte mari afișate de metoda update din *LargeWordConsumer*, se va restaura starea *savedStates[savedStates.size % 7]*

Un exemplu practic de utilizare a acestei aplicații ar fi restaurarea unei stări, după ce a fost consumată dintr-o coadă de mesaje.

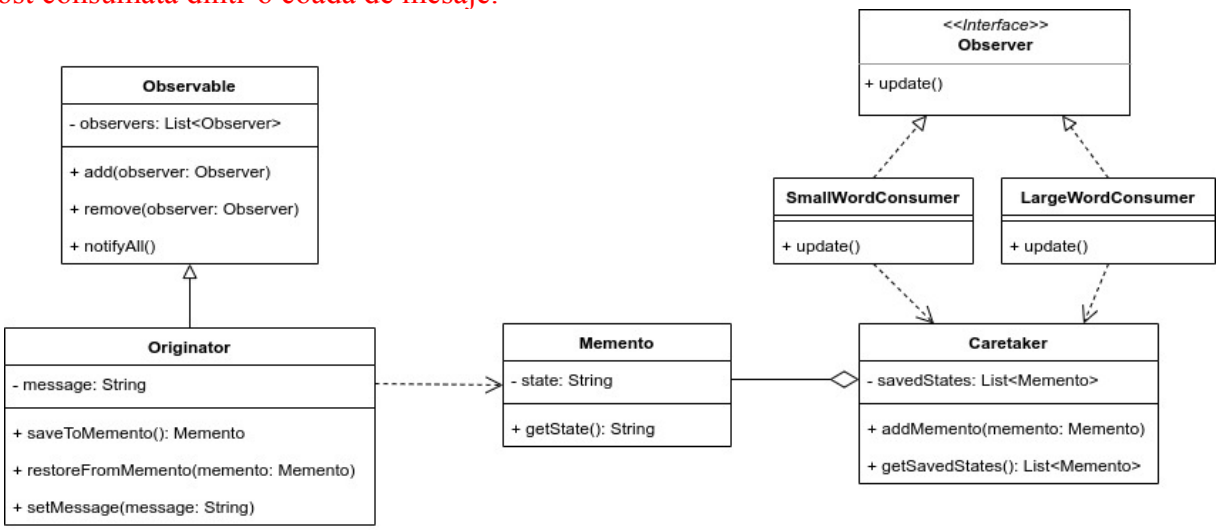


Diagrama de clase

Teme pe acasă:

1) Să se proiecteze și să se implementeze o aplicație care calculează ieșirile unor porți AND cu 2, 3, 4 și 8 intrări.

- Aplicația va decupla abstractizarea porții logice de implementarea acesteia prin utilizarea modelul *Bridge*.
- Fiecare poartă AND (cu 2, 3, 4, 8 intrări) va fi construită prin intermediul modelului *Builder* (se va trimite fiecare intrare în parte prin builder).
- Ieșirile porților logice vor fi calculate cu ajutorul unui *automat finit de stări* (vezi exemplul din curs).

2) Să se proiecteze și să se implementeze un browser pentru copii. Acesta va utiliza modelul *Prototype* pentru a crea o cerere generică, modelul *Proxy* pentru a adăuga control parental asupra unor cereri HTTP de tip GET, precum și modelul *Facade* pentru a oferi o interfață cât mai simplă.

Mai jos se regăsește un exemplu de proiectare. Acesta poate fi reproiectat dacă se dorește sau este necesar.

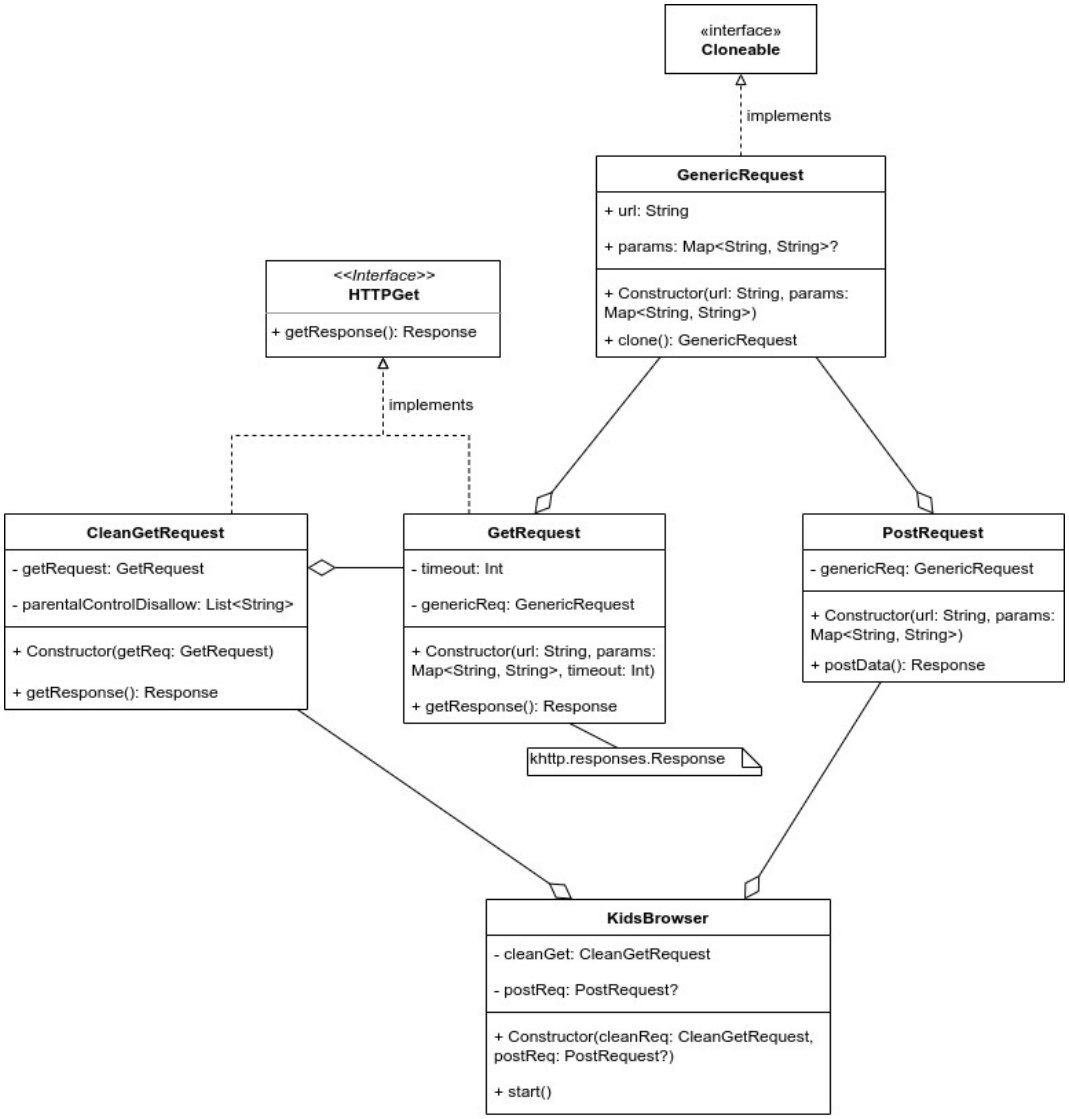


Diagrama de clase

Hint: Vezi documentația oficială a modulului khttp: https://khttp.readthedocs.io/en/latest/

Se va crea un proiect Maven, se va bifa opțiunea „Create from archetype”, apoi se va selecta *org.jetbrains.kotlin:kotlin-archetype-jvm --> kotlin-archetype-jvm:1.3.71*. În ***pom.xml*** se va adăuga:

```
<repositories>
    <repository>
        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>

<dependencies>
...
    <dependency>
        <groupId>com.github.jkcclemens</groupId>
        <artifactId>khttp</artifactId>
        <version>0.1.0</version>
    </dependency>
    ...
</dependencies
```

3) Temă de studiu: Mediator vs Proxy vs Adapter