Paradigma Modelelor de Proiectare

Cursul nr. 8

Mihai Zaharia

Design Pattern - Definiții

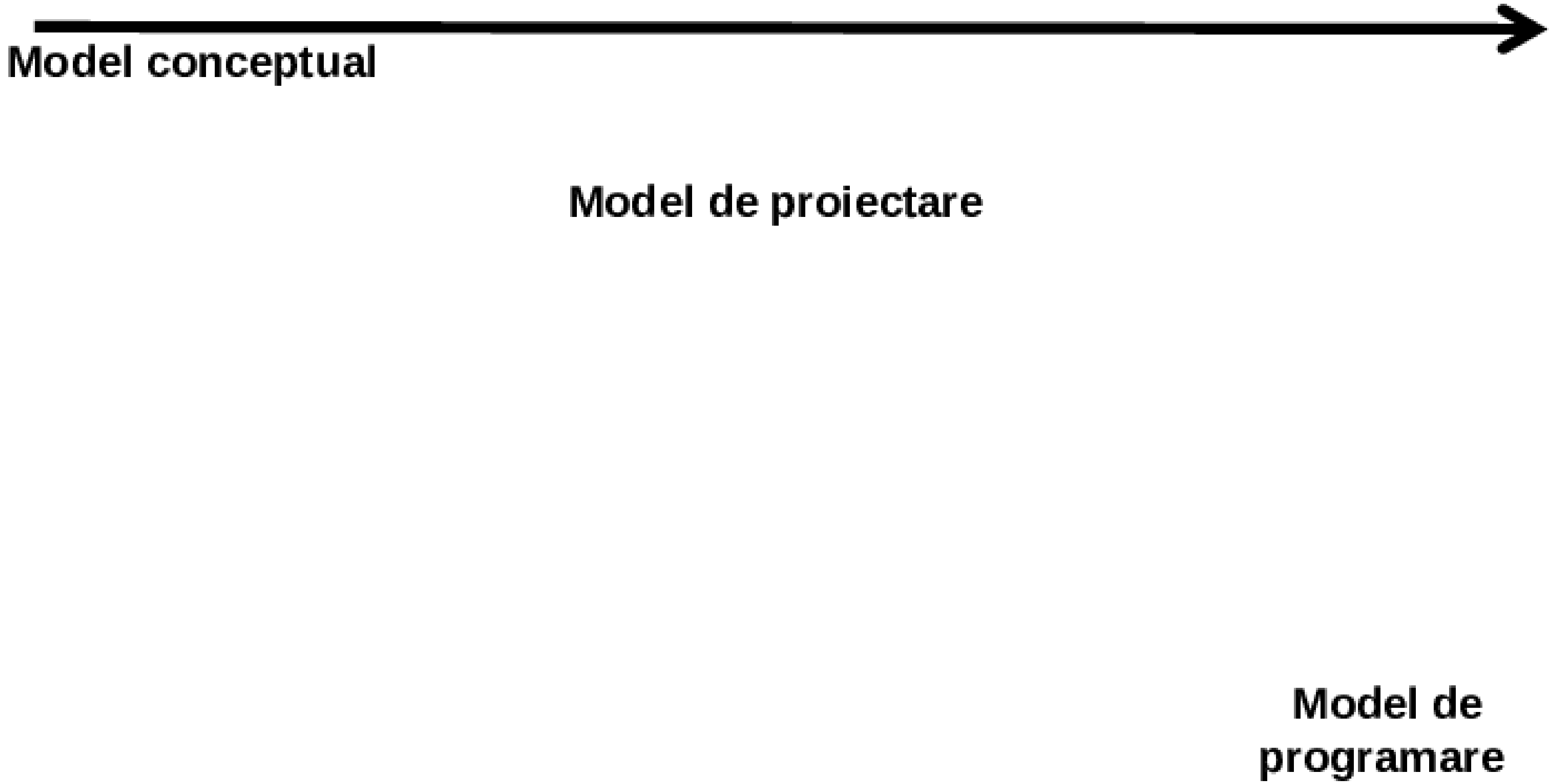
- **Conform dicţionarului** Merriam-Webster termenul de pattern înseamnă:
  - 1. o formă sau model propus pentru imitare
  - 2. ceva proiectat sau folosit ca model pentru a face lucruri (calapodul croitorului)
  - 3. o formă sau un proiect
  - 4. o configuraţie de evenimente
  - 5. ruta prestabilită a unui avion
  - 6. model comportamental
- Are ca sinonim indicat termenul de **model**

Istoria evoluției conceptului în IT

- 1987 Cunningham şi Beck - limbaj
- 1990 “Gaşca celor patru” – G4 - catalog
- 1995 GoF - carte

Apoi...

• Riehle şi Zullighoven menţionează trei tipuri de modele software



Elementele unui model

Numele:

Consecinţele!

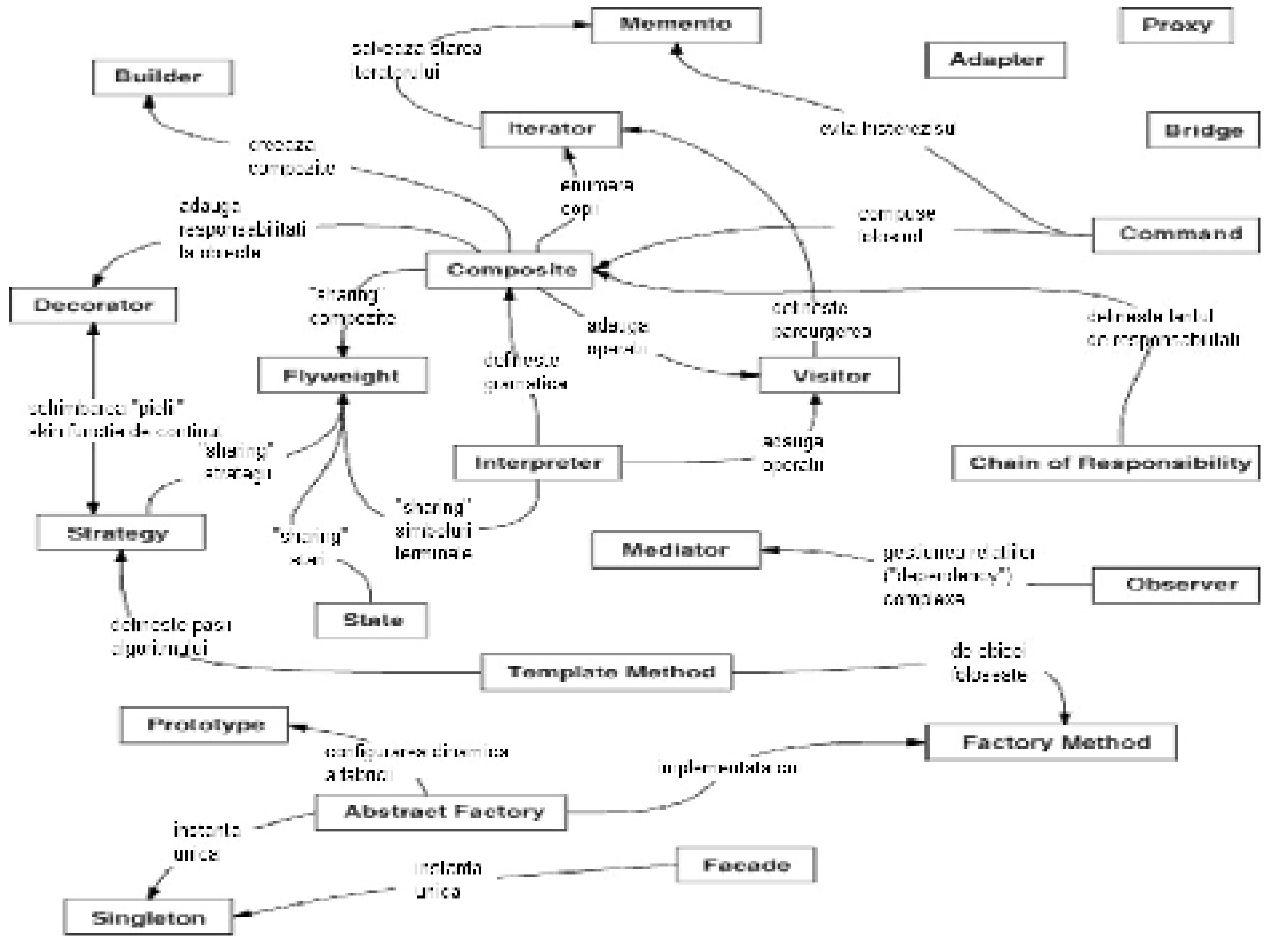
Problema:

Soluţia:

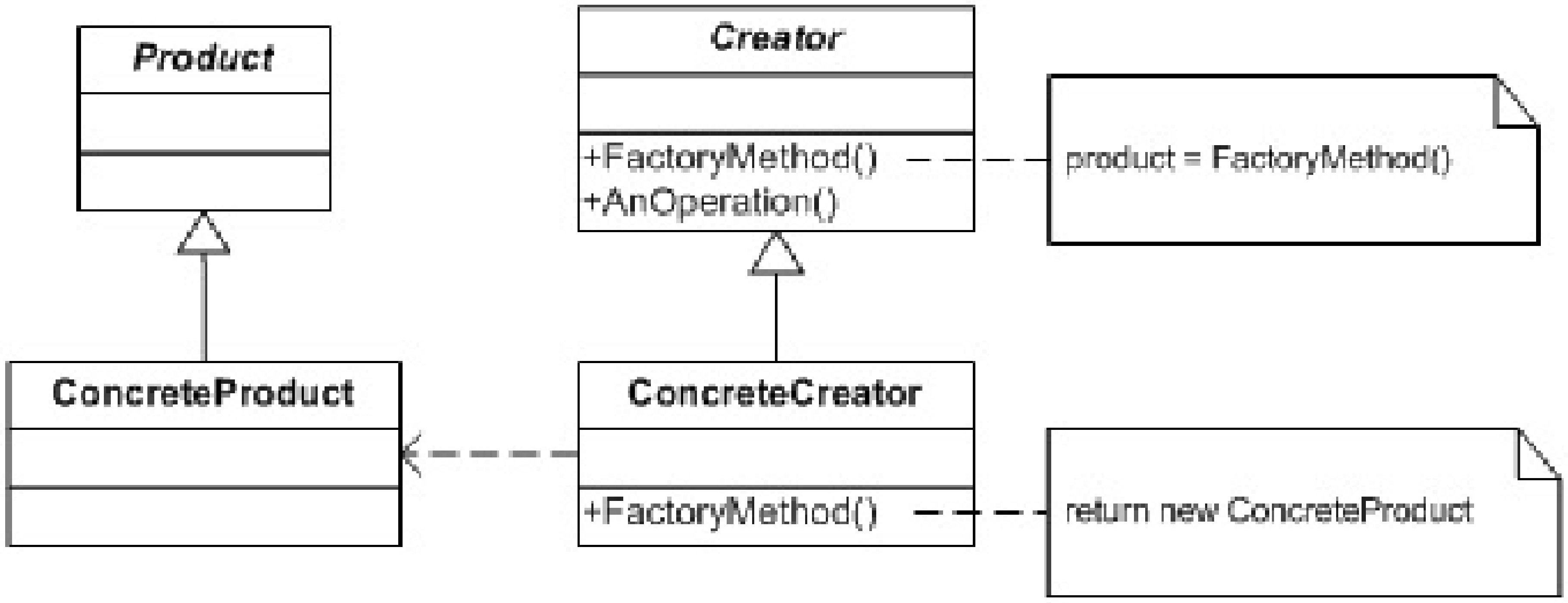
G4

		Scop		
		Creaţional	Structural	**Comportamental**
Domeniu	Clasă	Fabric Method	Adapter (clasă)	**Interpreter** **Template Method**
	Obiect	**Abstract Fabrica** **Builder** **Prototype** **Singleton**	**Adapter (obiect)** **Bridge** **Composite** **Decorator** **Facade** **Flyweight** **Proxy**	**Chain of Responsibility** **Command** **Iterator** **Mediator** **Memento** **Observer** **State** **Strategy** **Visitor**

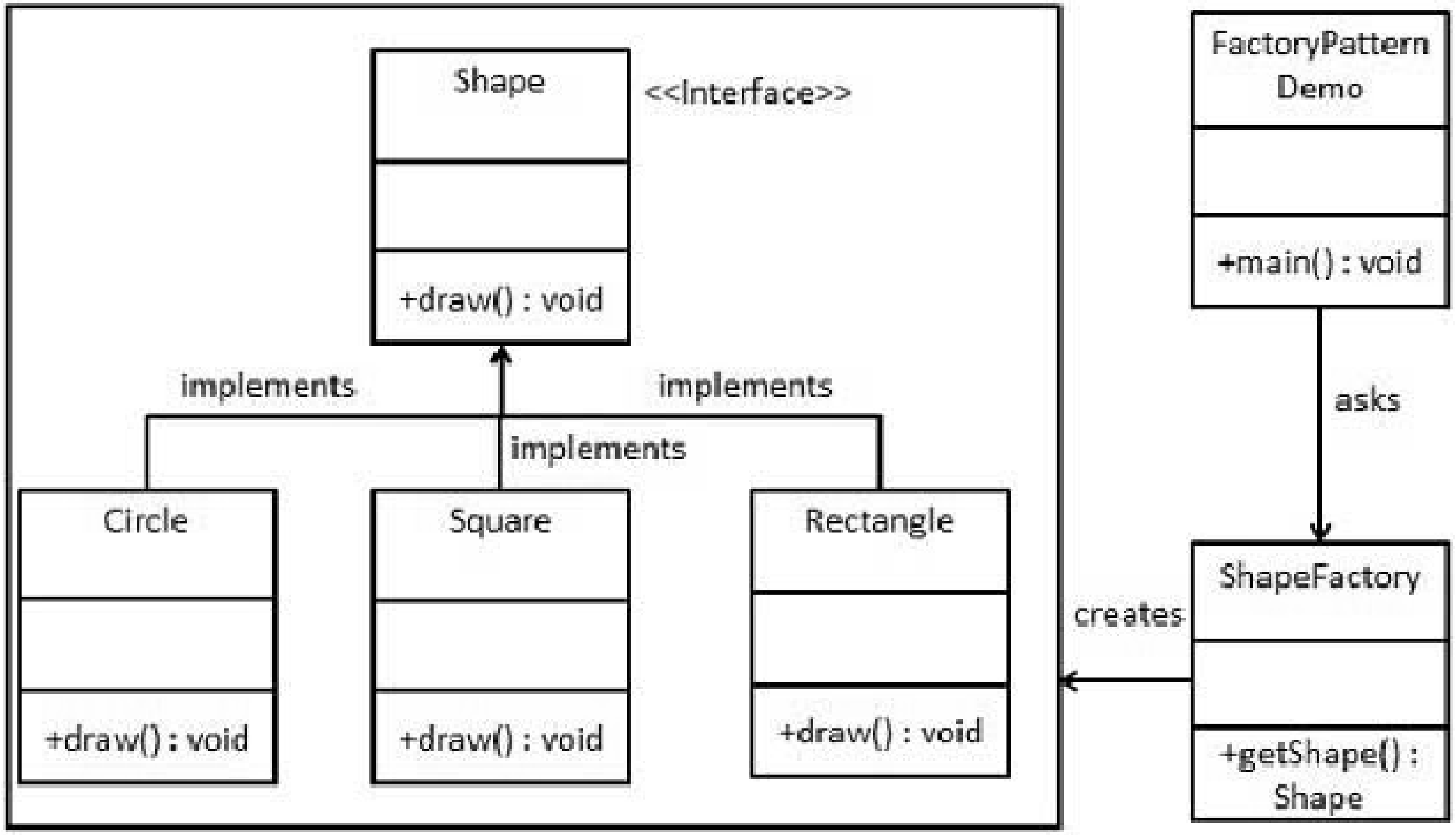
Relații între modelele GoF



Modelul Fabrică de obiecte



Modelul Fabrică de obiecte - caz de utilizare

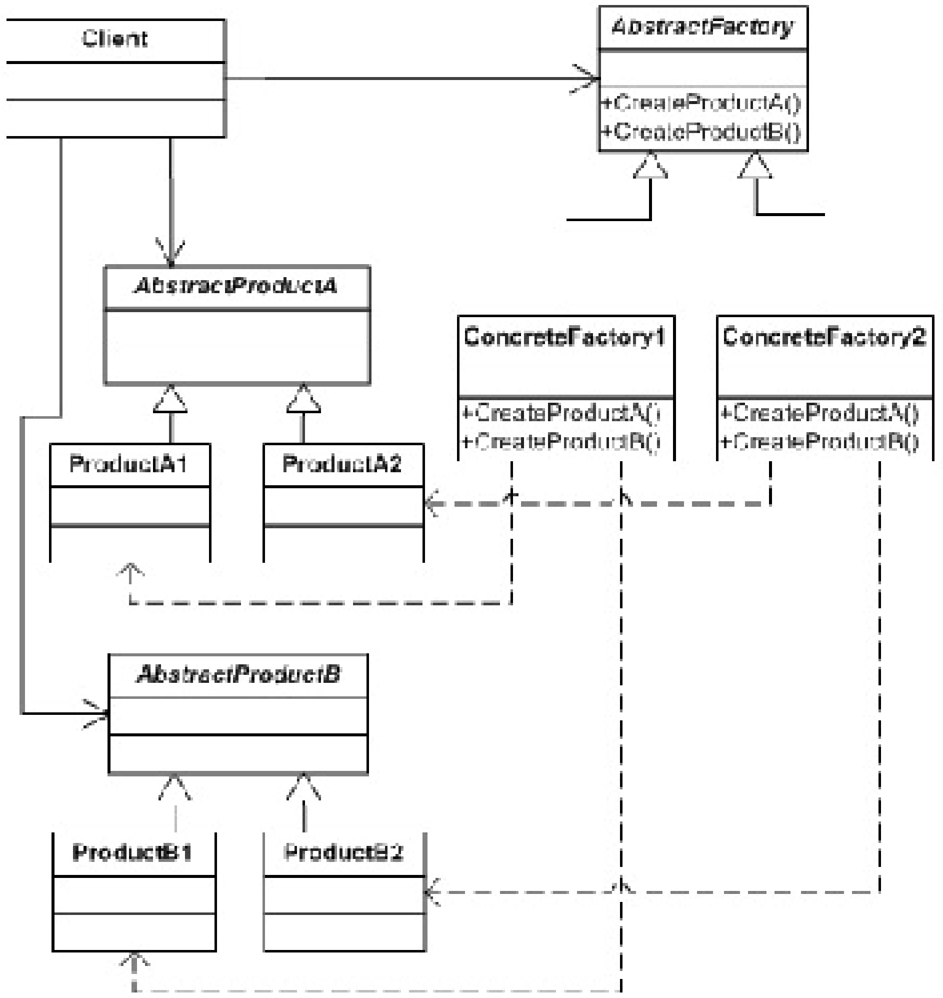


Modelul Fabrică de obiecte - caz de utilizare - implementare

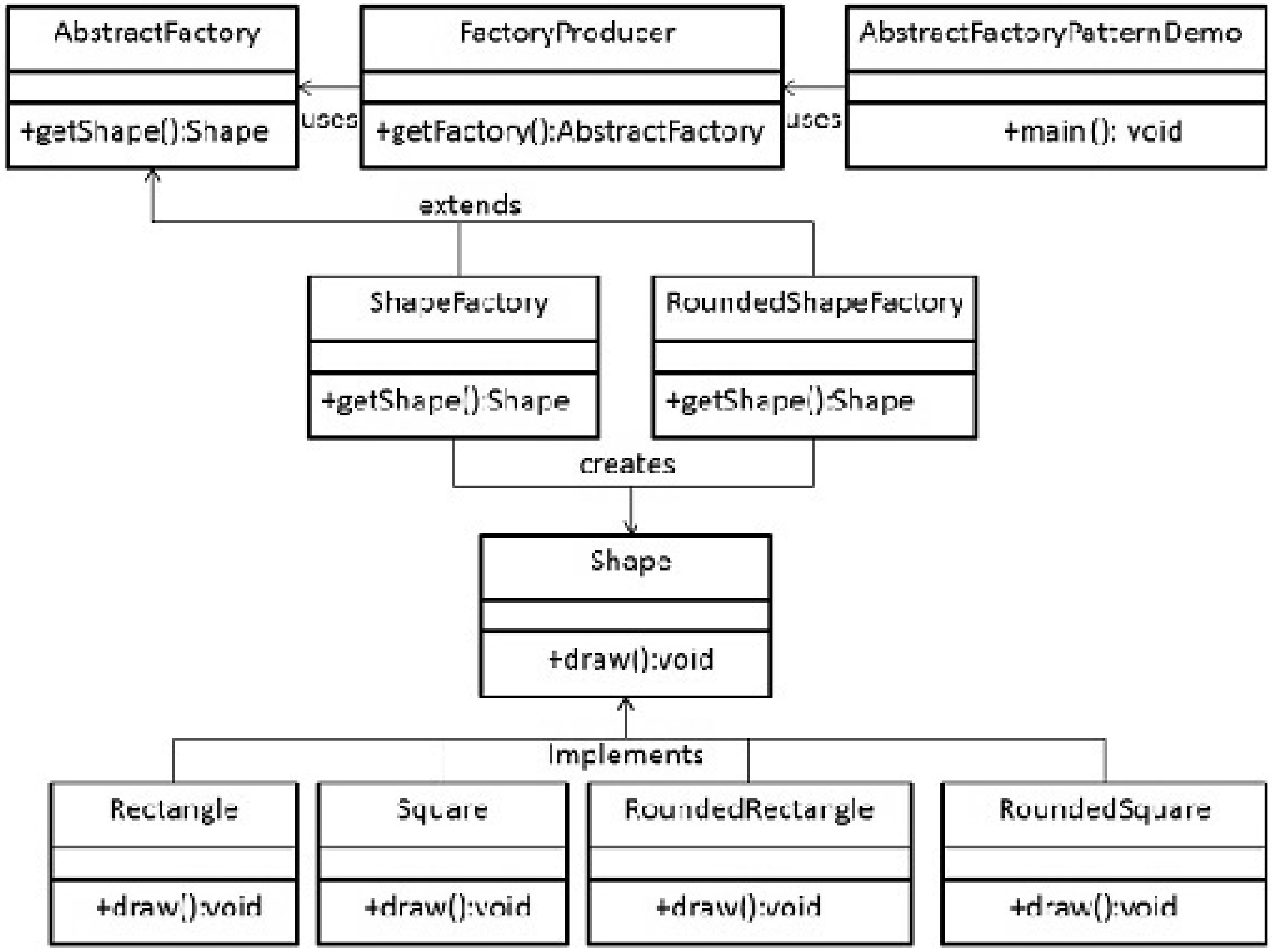
```
interface Shape
{ fun draw() }
class ShapeFactory {
 fun getShape(shapeType: String?): Shape?
  if (shapeType.equals("CIRCLE", true))
     return Circle()
  if (shapeType.equals("RECTANGLE", true))
     return Rectangle()
  if (shapeType.equals("SQUARE", true))
     return Square()
  return null  }
}

fun main(args: Array<String>)
{
   val shapeFactory = ShapeFactory()
   shapeFactory.getShape("CIRCLE")?.draw()
   shapeFactory.getShape("RECTANGLE")?.draw()
   shapeFactory.getShape("SQUARE")?.draw()
}
```

```
class Circle : Shape
{
   override fun draw()
   { println("Inside Circle::draw() method.") }
}

class Rectangle : Shape
{
   override fun draw()
   { println("Inside Rectangle::draw() method.") }
}
class Square : Shape
{
   override fun draw()
   { println("Inside Square::draw() method.") }
}
```

Model Fabrica abstractă



Modelul Fabrica abstractă - caz de utilizare



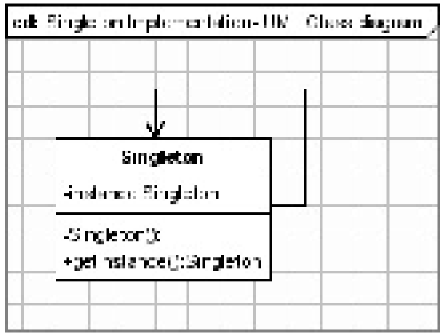
Modelul Fabrica abstractă - implementare

```
interface Shape
{    fun draw()  }
interface Color
{  fun fill()  }
abstract class AbstractFactory {
    abstract fun getColor(color: String): Color?
    abstract fun getShape(shape: String): Shape?  }
class ShapeFactory : AbstractFactory() {
    override fun getShape(shape: String): Shape?
       { if (shape.equals("CIRCLE", true)) return Circle()
        if (shape.equals("RECTANGLE", true)) return Rectangle()
        if (shape.equals("SQUARE", true)) return Square()
        return null  }
    override fun getColor(color: String): Color? = null  }
class ColorFactory : AbstractFactory() {
    override fun getShape(shape: String): Shape? = null
    override fun getColor(color: String): Color?
       { if (color.equals("RED", true)) return Red()
        if (color.equals("GREEN", true)) return Green()
        if (color.equals("BLUE", true)) return Blue()
        return null  }  }
object FactoryProducer {
    fun getFactory(choice: String): AbstractFactory?
        { if (choice.equals("SHAPE", true)) return ShapeFactory()
        if (choice.equals("COLOR", true)) return ColorFactory()
        return null  } }
```

```
class Circle : Shape {
    override fun draw()
   {  println("Inside Circle::draw() method.")  }   }
class Square : Shape {
    override fun draw()
    {  println("Inside Square::draw() method.")  } }
class Rectangle : Shape {
    override fun draw()
    {  println("Inside Rectangle::draw() method.")  }  }
class Red : Color {
    override fun fill()
    {  println("Inside Red::fill() method.")  }  }
class Green : Color {
    override fun fill()
   {  println("Inside Green::fill() method.") } }
class Blue : Color {
    override fun fill()
   {  println("Inside Blue::fill() method.")  } }
fun main(args: Array<String>)
{  val shapeFactory = FactoryProducer.getFactory("SHAPE")
    shapeFactory?.getShape("CIRCLE")?.draw()
    shapeFactory?.getShape("RECTANGLE")?.draw()
    shapeFactory?.getShape("SQUARE")?.draw()

    val colorFactory = FactoryProducer.getFactory("COLOR")
    colorFactory?.getColor("RED")?.fill()
    colorFactory?.getColor("GREEN")?.fill()
    colorFactory?.getColor("BLUE")?.fill()   }
```

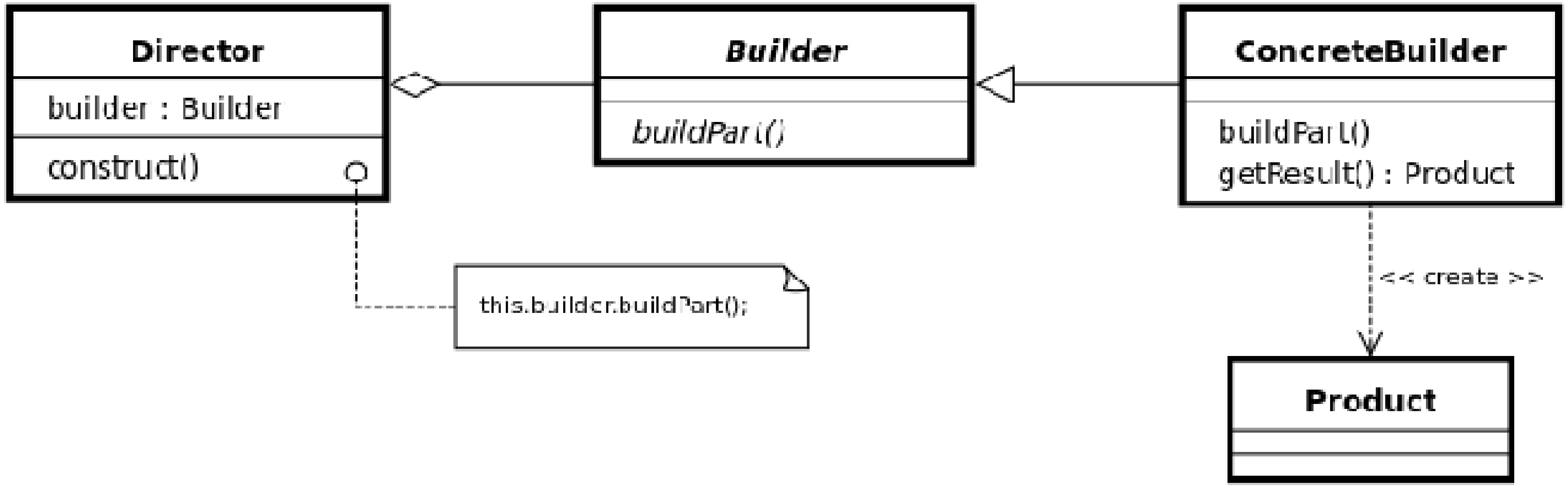
Modelul burlacului

object burlac



```
object Payroll
{
val allEmployees = arrayListOf<Person>()
fun calculateSalary()
  {
  for (person in allEmployees)
      {
        ...
      }
  }
}
```

Modelul constructor



Model constructor - implementare concretă

```
data class Mail(val to: String,
    val title: String = "",
    val message: String = "",
    val cc: List<String> = listOf(),
    val bcc: List<String> = listOf(),
    val attachments: List<java.io.File> = listOf())

class MailBuilder(val to: String)
{
    private var mail: Mail = Mail(to)
    fun title(title: String): MailBuilder
    {
        mail.title = title
        return this
    }
    // acesta se repeta pentru alte variatii
    fun build(): Mail
    {  return mail  }
}
```

- și utilizare imediată:

```
val mail = Mail("one@recepient.org",
"Hi", "How are you")
```

- sau utilizare de obiect construit particularizat:

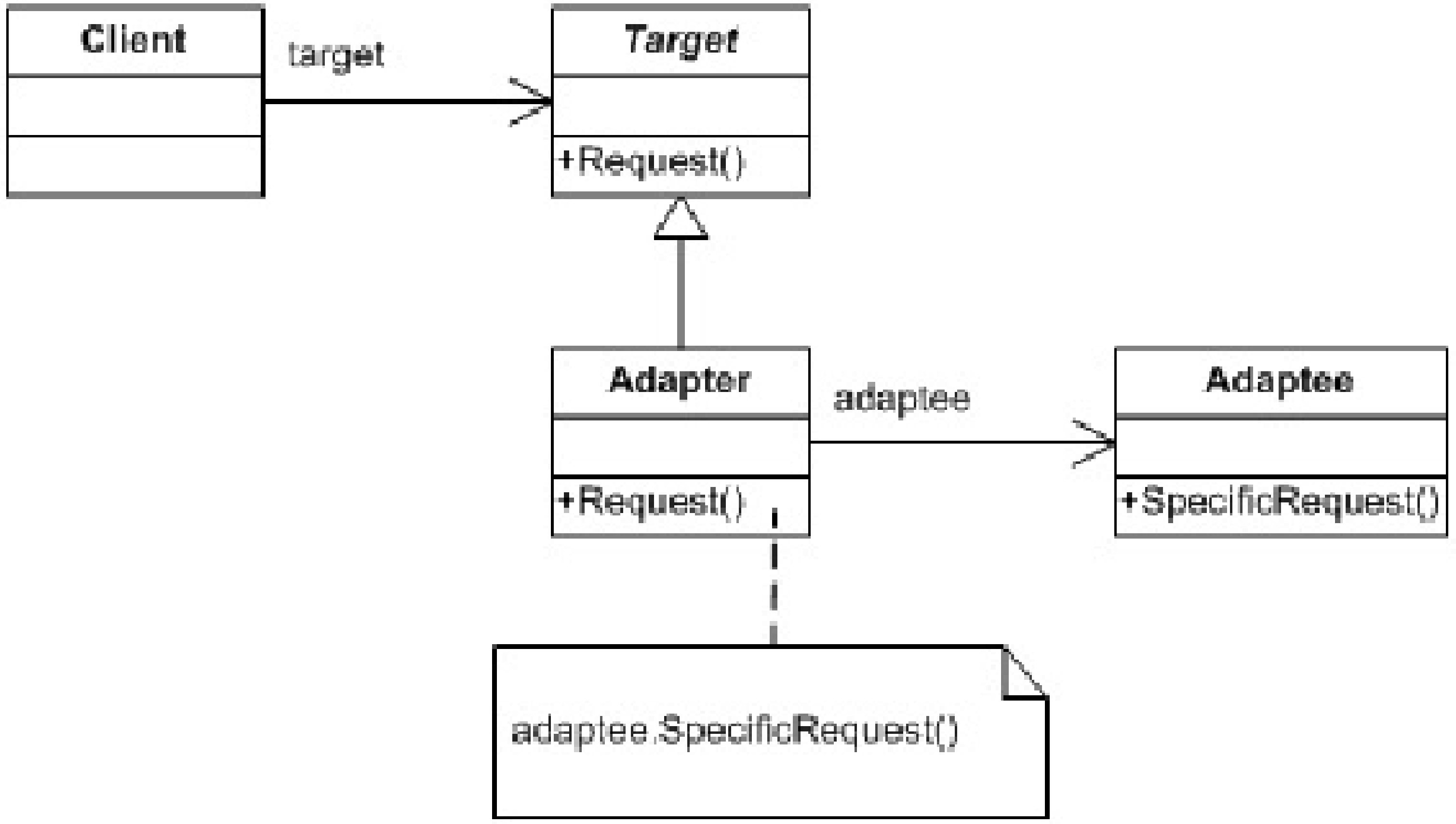
```
val email = MailBuilder("hello@hello.com").title
("What's up?").build()
```

Model protitip - implemntare de caz

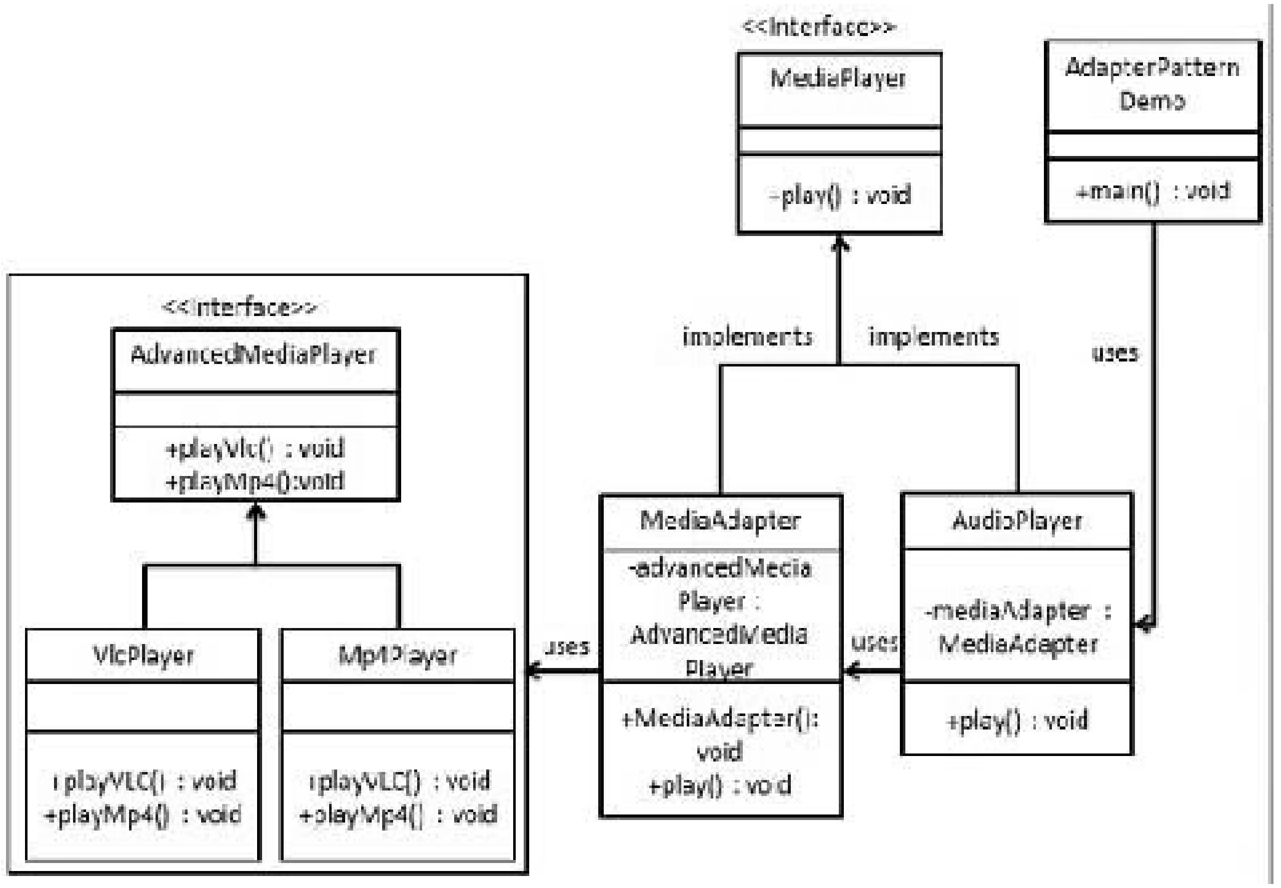
```
open class Bike : Cloneable
{
    private var gears: Int = 0
    private var bikeType: String? = null
    var model: String? = null
        private set

    init
    {
        bikeType = "Standard"
        model = "Carpati"
        gears = 4
    }

    public override fun clone(): Bike {
        return Bike()
    }
```

```
    fun makeAdvanced()
{
        bikeType = "Advanced"
        model = "Jaguar"
        gears = 6
    }
}

fun makeJaguar(basicBike: Bike): Bike
{
    basicBike.makeAdvanced()
    return basicBike
}

fun main(args: Array<String>)
{
    val bike = Bike()
    val basicBike = bike.clone()
    val advancedBike = makeJaguar(basicBike)
    println("Bicicleta mai buna: " + advancedBike.model!!)
}
```

Modelul Adaptor



Model Adaptor - caz de utilizare

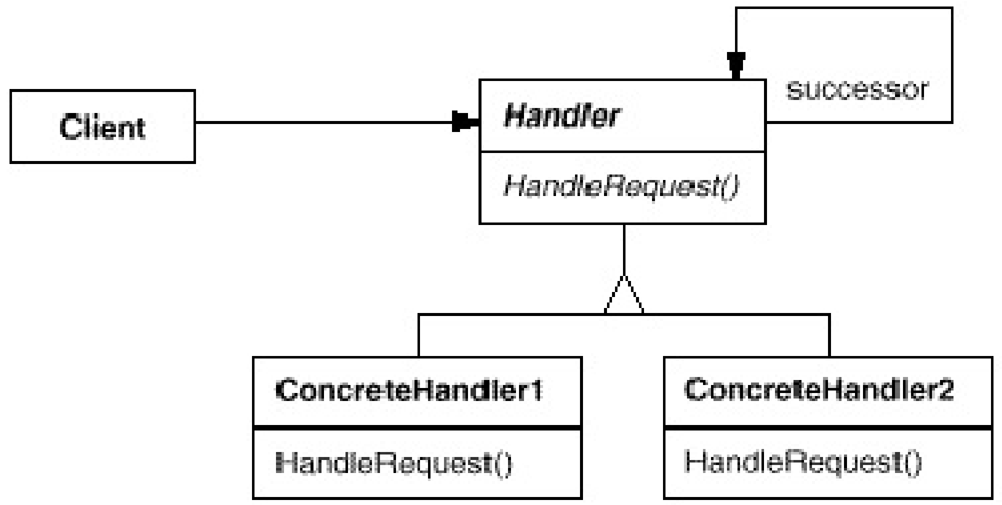


Model Adaptor - implementare

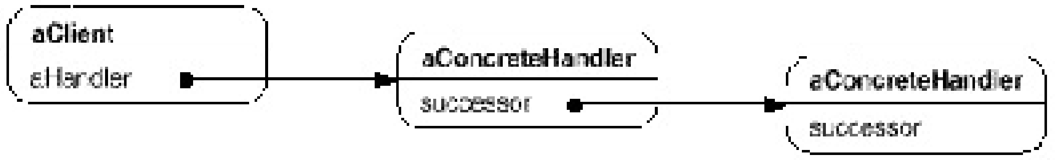
```
interface AdvanceMediaPlayer
  {fun playVlc(fileName: String)
   fun playMp4(fileName: String) }
interface MediaPlayer
  { fun play(audioType: String, fileName: String) }
open class MediaAdapter : MediaPlayer
  { private var advancedMusicPlayer: AdvanceMediaPlayer? = null
    override fun play(audioType: String, fileName: String)
    { if (audioType.equals("vlc", true))
         {if (advancedMusicPlayer == null)
              { advancedMusicPlayer = VlcPlayer() }
              advancedMusicPlayer?.playVlc(fileName) }
       else if (audioType.equals("mp4", true))
        {if (advancedMusicPlayer == null)
             { advancedMusicPlayer = Mp4Player()  }
             advancedMusicPlayer?.playMp4(fileName) }   }
}
class AudioPlayer : MediaAdapter()
{    override fun play(audioType: String, fileName: String)
    {   if (audioType.equals("mp3", true))
                { println("Playing mp3 file. Name: $fileName ") }
       else if (audioType.equals("vlc", true) || audioType.equals("mp4", true))
                { MediaAdapter().play(audioType, fileName) }
             else { println("Invalid media. $audioType format not supported") }  }
    }
```

```
class Mp4Player : AdvanceMediaPlayer {
   override fun playMp4(fileName: String) {
      println("Playing mp4 file. Name: $fileName")
   }
   override fun playVlc(fileName: String) {
      println("Only support mp4 type")
   }
}
class VlcPlayer : AdvanceMediaPlayer {
   override fun playMp4(fileName: String) {
      println("Only support vlc type")
   }
   override fun playVlc(fileName: String) {
      println("Playing vlc file. Name: $fileName")
   }
}
fun main(args: Array<String>) {
   val audioPlayer = AudioPlayer()
   audioPlayer.play("mp3", "beyond the horizon.mp3")
   audioPlayer.play("mp4", "alone.mp4")
   audioPlayer.play("vlc", "far far away.vlc")
   audioPlayer.play("avi", "mind me.avi")
}
```

Modelul lanț de responsabilități



Unde o structură tipică de înlănțuire de obiecte ar fi



Modelul lanț de responsabilități - implementare

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
interface HeadersChain
{    fun addHeader(inputHeader: String): String   }
class AuthenticationHeader(val token: String?, var next: HeadersChain? =
null) : HeadersChain
{    override fun addHeader(inputHeader: String): String
     { token ?: throw IllegalStateException("Token should be not null")
       return inputHeader + "Authorization: Bearer $token\n"
           .let { next?.addHeader(it) ?: it }  }     }
class ContentTypeHeader(val contentType: String, var next: HeadersChain?
= null) : HeadersChain
{  override fun addHeader(inputHeader: String): String =
       inputHeader + "ContentType: $contentType\n"
           .let { next?.addHeader(it) ?: it }  }
class BodyPayload(val body: String, var next: HeadersChain? = null) :
HeadersChain
{    override fun addHeader(inputHeader: String): String =
       inputHeader + "$body"
           .let { next?.addHeader(it) ?: it }   }
class ChainOfResponsibilityTest
{    @Test
    fun `Chain Of Responsibility`()
     { //crearea elemnteloru lantuli
       val authenticationHeader = AuthenticationHeader("123456")
       val contentTypeHeader = ContentTypeHeader("json")
       val messageBody =
BodyPayload("Body:\n{\n\"username\"=\"dbacinski\"\n}")
//se construieste lantul
authenticationHeader.next = contentTypeHeader
contentTypeHeader.next = messageBody
//se executa lantul
val messageWithAuthentication =
    authenticationHeader.addHeader("Headers with Authentication:\n")
println(messageWithAuthentication)
val messageWithoutAuth =
    contentTypeHeader.addHeader("Headers:\n")
println(messageWithoutAuth)
assertThat(messageWithAuthentication).isEqualTo
(       """
        Headers with Authentication:
        Authorization: Bearer 123456
        ContentType: json
        Body:
        { "username"="bonjovi2987" }
        """.trimIndent() )
assertThat(messageWithoutAuth).isEqualTo
(       """
        Headers:
        ContentType: json
        Body:
        { "username"="dbacinski" }
    """.trimIndent() )
 }
}
```

Model Mediator - caz de utilizare

```
interface Command
  {  fun land()  }
class Flight(private val atcMediator: IATCMediator) : Command
{    override fun land()
       { if (atcMediator.isLandingOk)
              {  println("Landing done....")
           atcMediator.setLandingStatus(true)
        } else
           println("Will wait to land....")   }
   fun getReady()
     {  println("Getting ready...")  }   }
class Runway(private val atcMediator: IATCMediator) :
Command
{    init  { atcMediator.setLandingStatus(true)  }
     override fun land()
       {  println("Landing permission granted...")
          atcMediator.setLandingStatus(true)  }   }
interface IATCMediator
  { val isLandingOk: Boolean
     fun registerRunway(runway: Runway)
     fun registerFlight(flight: Flight)
     fun setLandingStatus(status: Boolean)  }
```

```
class ATCMediator : IATCMediator
{   private var flight: Flight? = null
    private var runway: Runway? = null
    override var isLandingOk: Boolean = false
    override fun registerRunway(runway: Runway)
       {   this.runway = runway  }
    override fun registerFlight(flight: Flight)
       {   this.flight = flight  }
    override fun setLandingStatus(status: Boolean)
       {   isLandingOk = status  }
}

fun main(args: Array<String>)
{
   val atcMediator = ATCMediator()
    val sparrow101 = Flight(atcMediator)
    val mainRunway = Runway(atcMediator)
    atcMediator.registerFlight(sparrow101)
    atcMediator.registerRunway(mainRunway)
    sparrow101.getReady()
    mainRunway.land()
    sparrow101.land()
}
```