Realizat de: Adam Orban
Coordonator: Lect. dr. Sabo Cosmin Nicolae

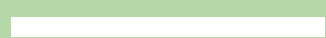
Rezultate obtinute

Eficientizarea căutărilor in bibliotecile online

Posibilitate de interconectare a codurilor scrise

Găsirea unor relații de interdependență intre bibliotecile online independente

Kotlin



Motivație

Este un limbaj nou, apărut in 2011, creat de JetBrains

In 7 mai, 2019, a devenit limbajul preferat pentru dezvoltarea de aplicații Android

Numele de Kotlin vine de la insula Kotlin, lângă St. Petersburg.

Comparatie cu Java

```
var nume = “John”

fun main(args: Array<String>) {
        val variabila = “World”
        println(“Hello, $scope!”)
}
```

```
String nume = “John”;

public static void main(String[] args) {
        String variabila = “World”;
        System.out.println(“Hello” +
variabila);
}
```

Extension functions

Similar cu C#, se pot defini metode suplimentare pentru o clasă, fără a se extinde clasa respectivă.

Este folositoare mai ales in situații când nu se poate extinde clasa.

```
fun String.lastChar() : Char = get(length - 1)
```

Alte facilități

Clasele sunt implicit publice și finale

Data class - clasă specifică pentru stocarea datelor

Type inference - compilatorul deduce tipul variabilei declarate

Null pointer safety

+

Build Gradle KTS

Gradle

O unealtă open-source pentru automatizarea build-urilor

In Android, este folosit mai ales pentru declararea dependințelor din proiect

Fișierele gradle folosesc limbajul Groovy

Avantaje Build Gradle KTS

Folosește limbajul Kotlin in locul limbajului Groovy

Se stochează dependințele proiectului intr-un fișier dedicat

Completarea codului

Fiind o metodă nouă pentru interacționarea cu Gradle, sunt unele buguri, probleme de performanță

Arhitectura

Descriere

O arhitectură este importantă in oricare proiect, pentru a fi mai ușor de intreținut și extins

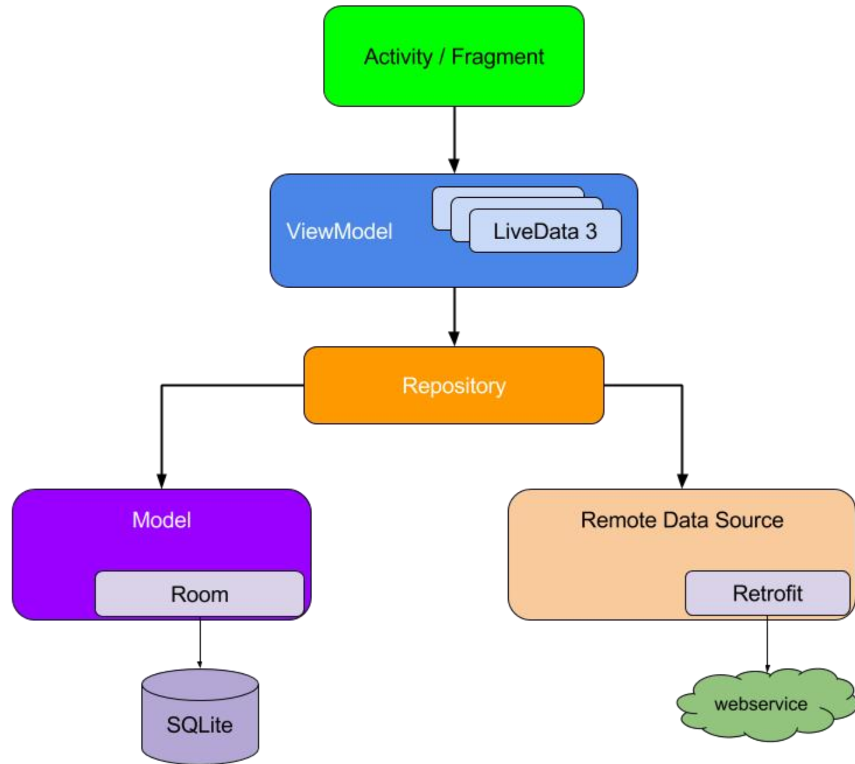
Dintre arhitecturile care sunt pe Android am ales MVVM

MVVM - Model - View - ViewModel

Este arhitectura recomandată de către Google pentru dezvoltarea aplicațiilor

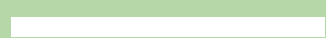
Presupune crearea de cereri unidirecționale: componenta de view cere date, care sunt procesate de ViewModel, care apelează modelul. Rezultatul modelului este trimis la ViewModel, in final la view.

Componenta de view nu conține logică de business, este numai o componentă care afișează datele fără a știi de unde vin datele



Model arhitectura MVVM

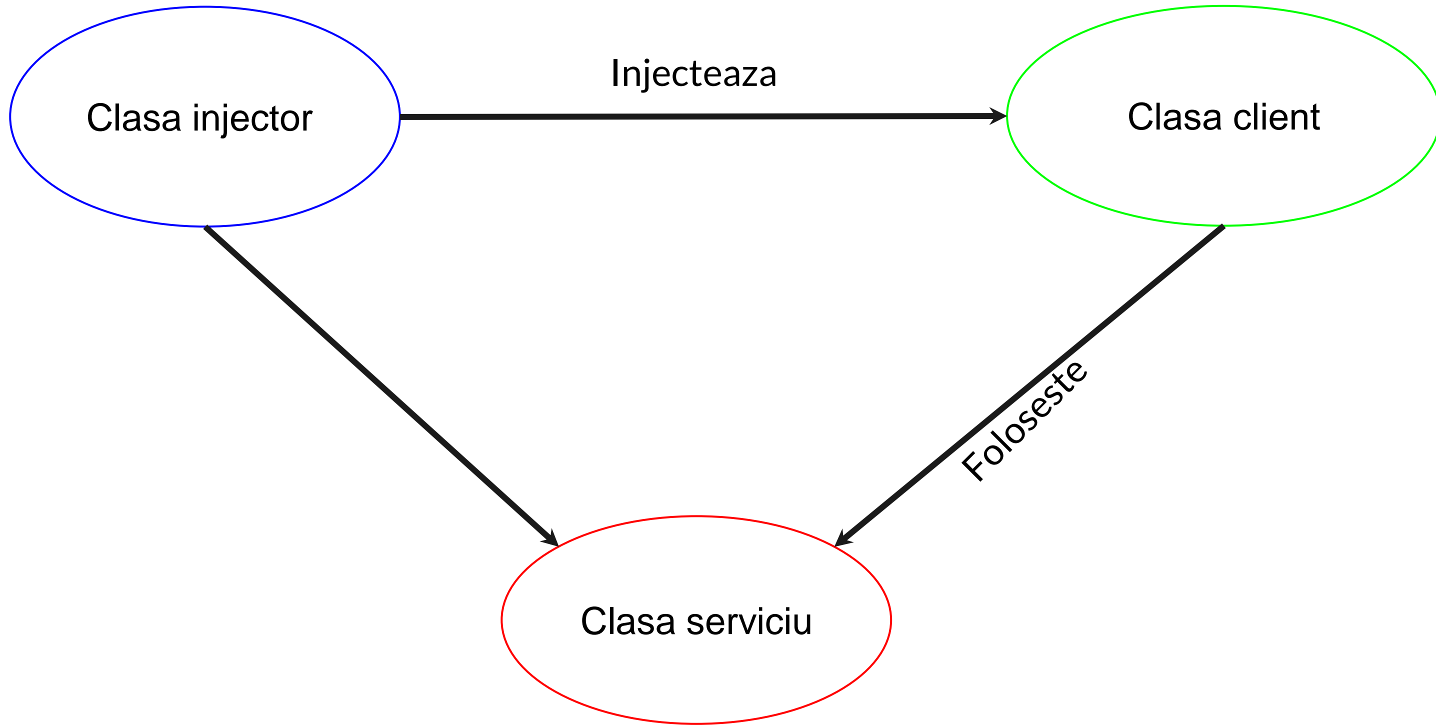
Dependency Injection



Un obiect oferă dependințele unei alt obiect

Injecție inseamnă pasarea de dependință intr-un obiect care o va folosi

Se obține Separation of Concerns (separarea de preocupări) de creare și folosire de obiecte


Injecteaza
Clasa injector
Clasa client
Foloseste
Clasa serviciu

Avantaje

Un client poate fi configurabil

Se externalizează detaliile de configurație ale sistemului in fișiere de configurații

Poate folosit in cod legacy ca și refactorizare

Micșorează cuplarea dintre clasă și dependințele sale

Dezavantaje

Crează clienți care necesită detalii de configurare

Poate face codul mai greu de trackuit, deoarece separă comportamentul de construcție

Librăriile sunt realizate prin reflection

Dependency injection

Verbul cel mai apropiat este “a da”

```
val geam = Geam()
val usa = Usa()
val casa = Casa(geam, usa)
```

Service Locator

Verbul cel mai apropiat este “a lua”

```
val casa = serviceLocator.get(Casa::class)
```

Este ușor de implementat

Koin

Facilități și rezultate

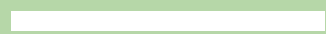
Library	Setup Kotlin	Setup Java	Inject Kotlin	Inject Java
Koin	11,28 ms	12,22 ms	0,25 ms	0,25 ms
Kodein	64,97 ms	64,91 ms	7,58 ms	7,59 ms
Katana	10,37 ms	10,43 ms	1,93 ms	1,90 ms
Custom	4,32 ms	4,32 ms	0,65 ms	0,80 ms
Dagger	0,01 ms	0,01 ms	0,23 ms	0,20 ms

Scris in Kotlin, fără generare de cod

Foarte rapid in comparație cu alte librării

Folosește factory si single pentru a furniza dependințe

Design patterns



Ce sunt

Soluții refolosibile, generale, pentru probleme recurente intr-un context de design software

O descriere sau template cum ar trebui rezolvată o problemă

Best practices

Design Patterns: Elements of Reusable Object-Oriented Software (1994) scris de către Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides conține 23 de pattern-uri pentru diferite probleme.

Creaționale

Sunt acele care creează obiecte, oferind programului mai multă flexibilitate in deciderea de creare de obiecte.

Builder

Conceput pentru a furniza o soluție flexibilă pentru diferite creații de obiecte

Separă construcția unui obiect de reprezentarea ei

Avantaje

Se poate varia reprezentarea internă a unui obiect

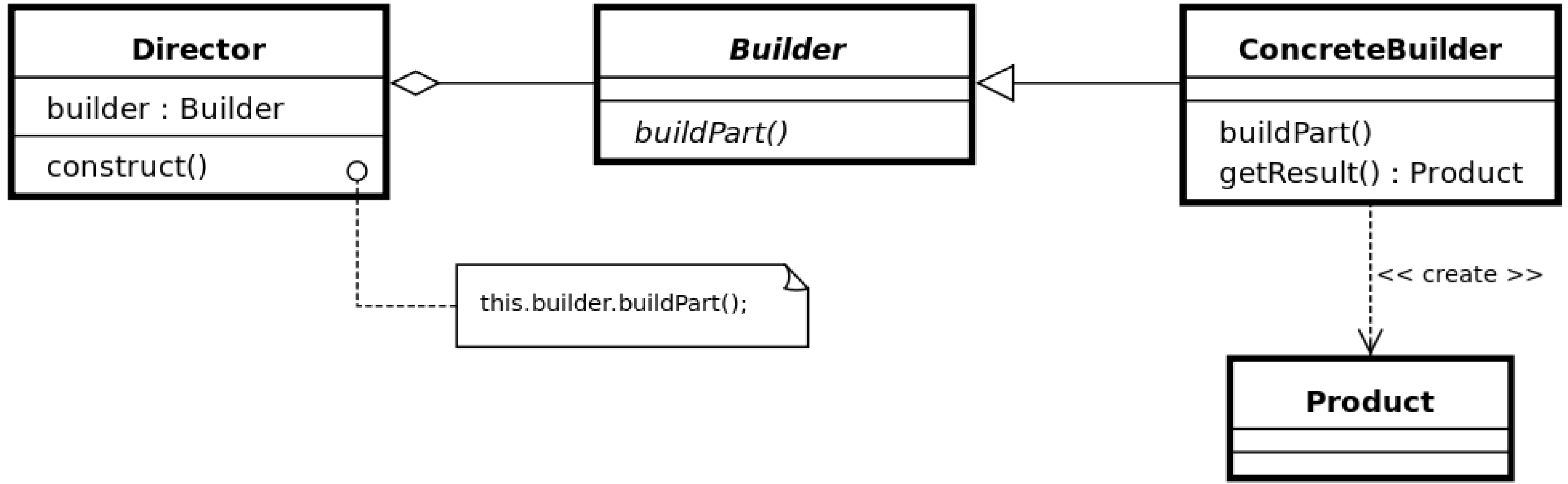
Encapsulează codul de construcție și reprezentare

Dezavantaje

Necesită crearea de un Builder pentru diferite tipuri de clase

Clasa builder trebuie să fie mutabilă

Nu funcționează dependency injection cu ușurință



UML

Factory

Folosește metode de factory pentru a crea obiecte fără a se specifica clasa exactă pe care vrem să instanțiem.

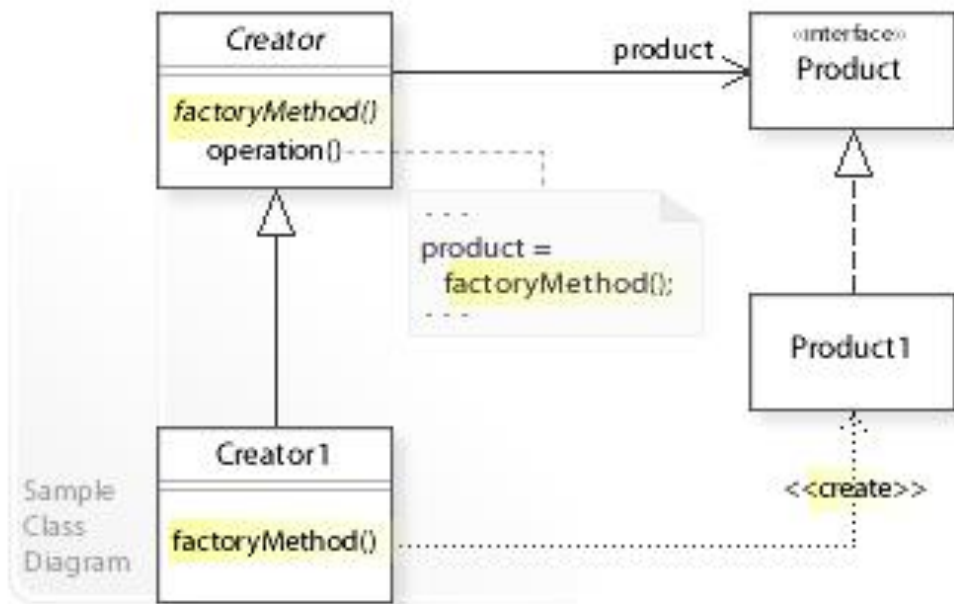
Respectă principiul SOLID

Avantaje

Ajută la construirea unei clase cu o componentă de un tip care nu a fost predefinit

Ajută la crearea de subclase la un părinte a cărui tip de component nu a fost predefinit

Se pote obține un cod mai ușor de citit unde există mai mulți constructori



UML

Structurale

Folosite pentru compoziția obiectelor, utlizând inheritance pentru a compune interfețe și pentru a defini metode de a compune obiecte pentru funcționalități noi.

Adapter

Deseori numit și Wrapper

Permite ca interfața unui obiect existent să fie folosit ca și o altă interfață

Este folosit pentru a face clasele existente să funcționeze cu altele fără a se modifica codul lor sursă

Avantaje

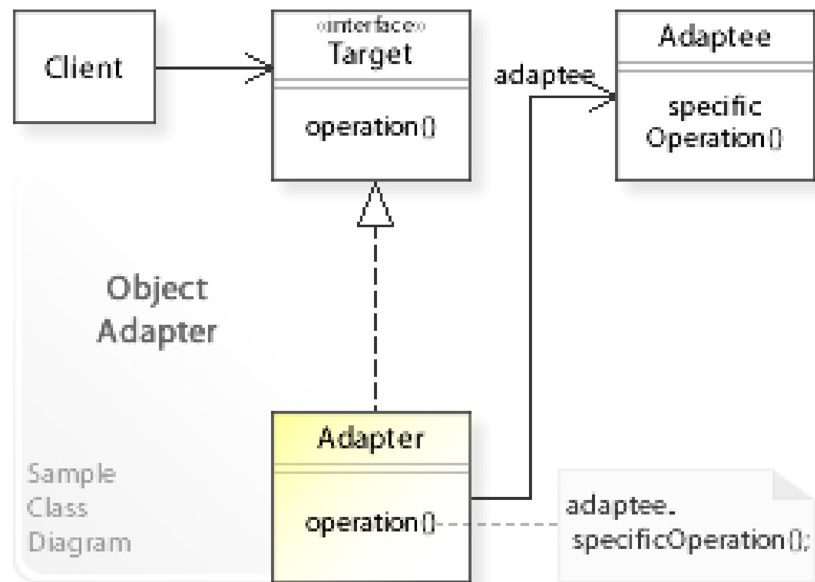
Foarte folositoare in cazuri in care se rescrie aplicația, dar este necesară respectarea contractului până când se migrează tot codul

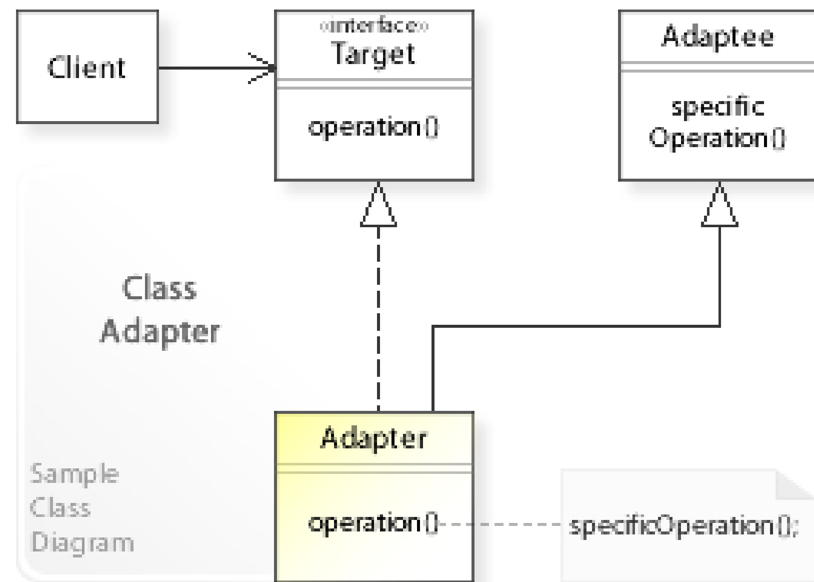
Se poate realiza o structură mai bună a noii clase care are performanțe mai bune

Dezavantaje

De multe ori conversia rezultă foarte mult cod pentru a putea instanția noua clasă

Deoarece se creează și vechea instanță, dacă acela realizează operații costisitoare, va impacta viteza de conversie și de rulare a programului





UML

Composite

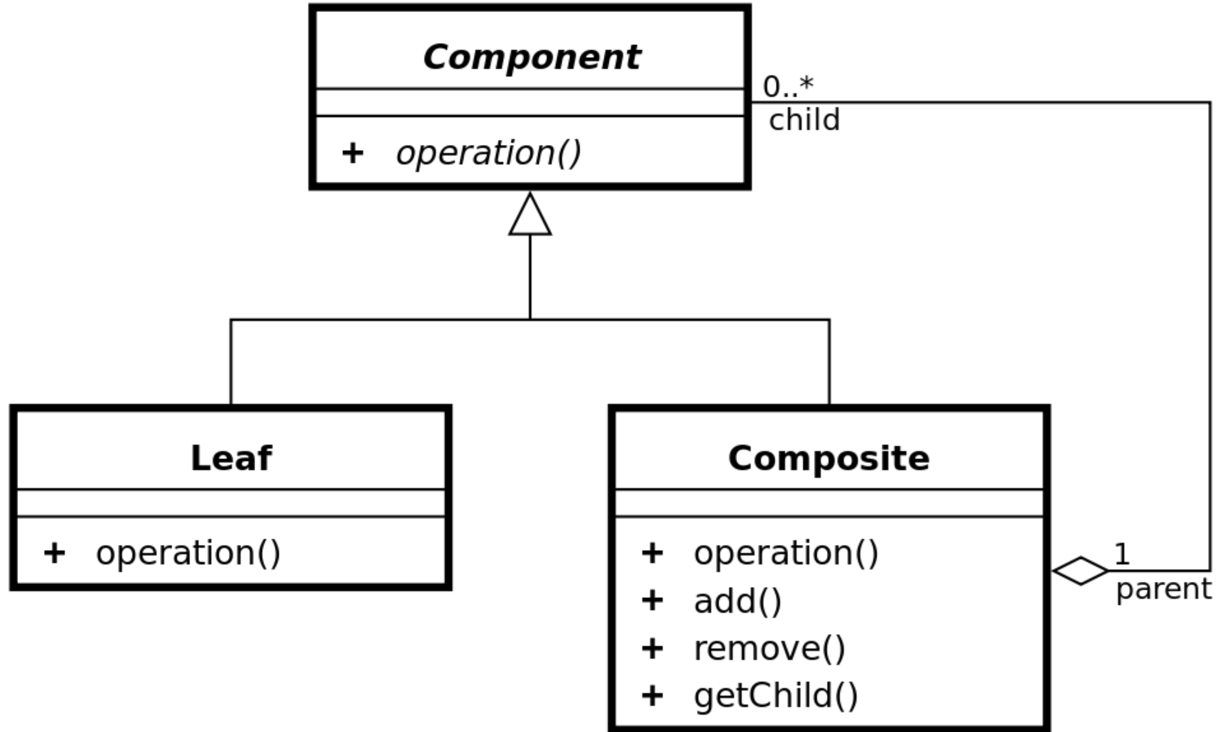
Este un pattern de partiționare

Descrie un grup de obiecte care sunt tratate in același mod ca și o singură instanță al aceluiași tip de obiect

Se poate folosi atunci când clienții ignoră diferențele dintre compoziții de obiecte și obiectele individuale

Avantaje

Obiectul care conține bucățile separate este unul mai mic, astfel fiind mai ușor de înțeles.



UML

Comportamentale

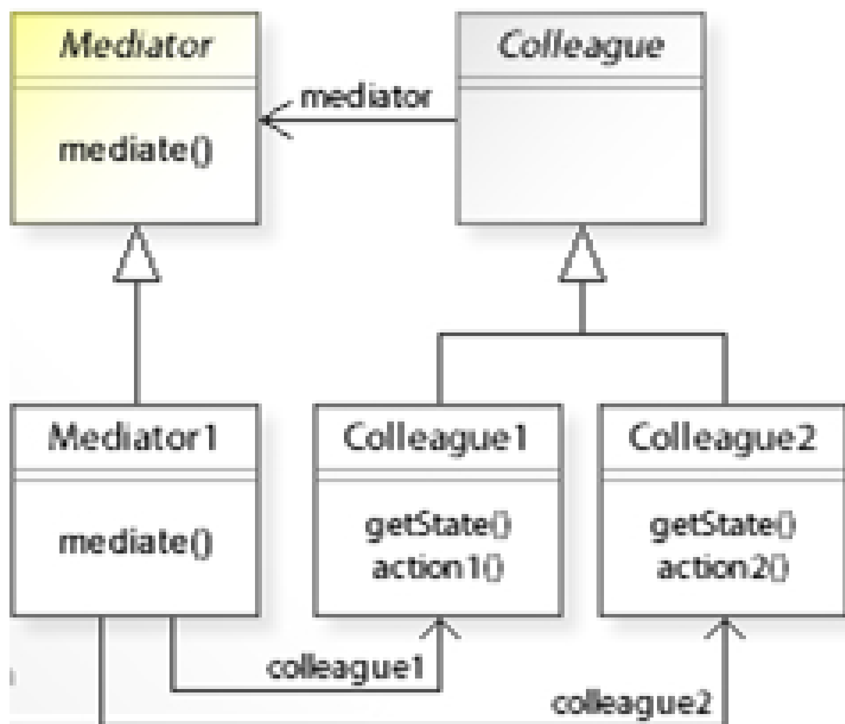
Folosite pentru comunicarea intre obiecte.

Mediator

Encapsulează cum interacționează un set de obiecte

Poate altera comportamentul unui program

Obiectele nu mai comunică una cu alta



UML

Strategy

Permite să se selectee un algoritm la runtime

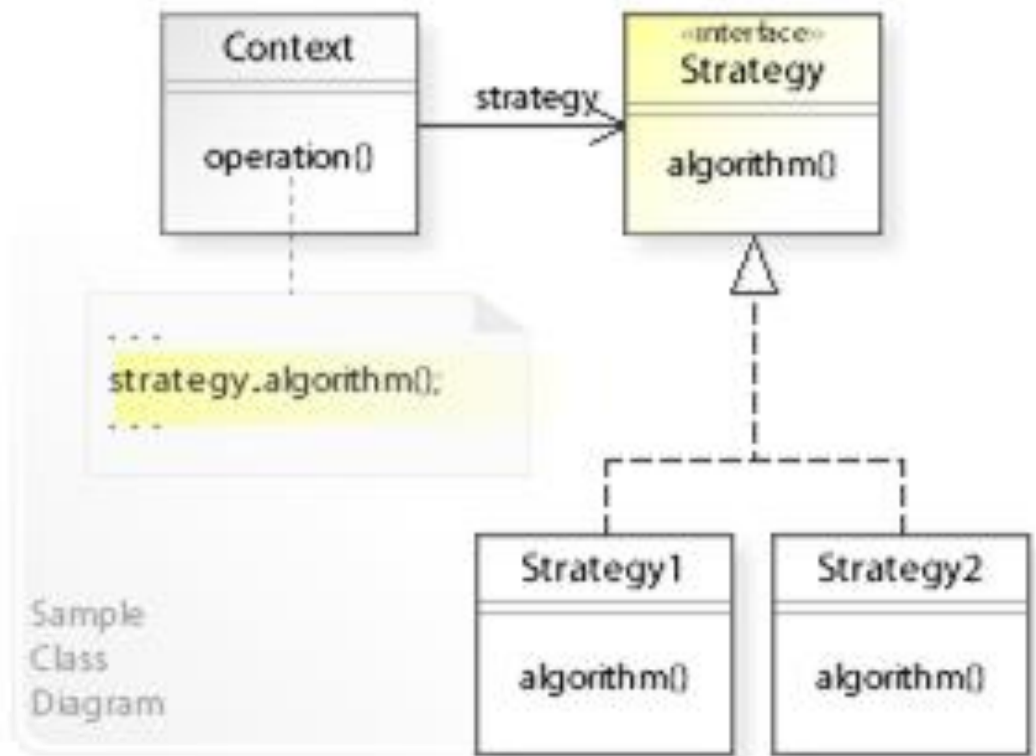
Lasă un algoritm să difere independent de clienții pe care o folosesc

Avantaje

Stochează o referință la cod și returnează acesta

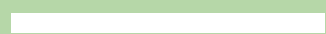
Comportamentul diferit este indicat să se encapsuleze in interfețe

Respectă principiul open/closed din SOLID



UML

Navigație



Problema curentă

Metoda curentă folosește cod in clasa Activity sau Fragment care apelează direct clasa unde vrem să navigăm

Presupune un efort de setare, care deobicei inseamnă timp pierdut de depistare de probleme

Nu sunt separate preocupările intre componentele care creează și care fac filtrarea de punct de navigație

Activity

```
context.startActivity(
Intent(this, DestinatieActivity::class.java)
)
```

Fragment

```
fragmentManager.beginTransaction()
   .replace(R.id.fragment_container,
DestinatieFragment.newInstance())
   .commit()
```

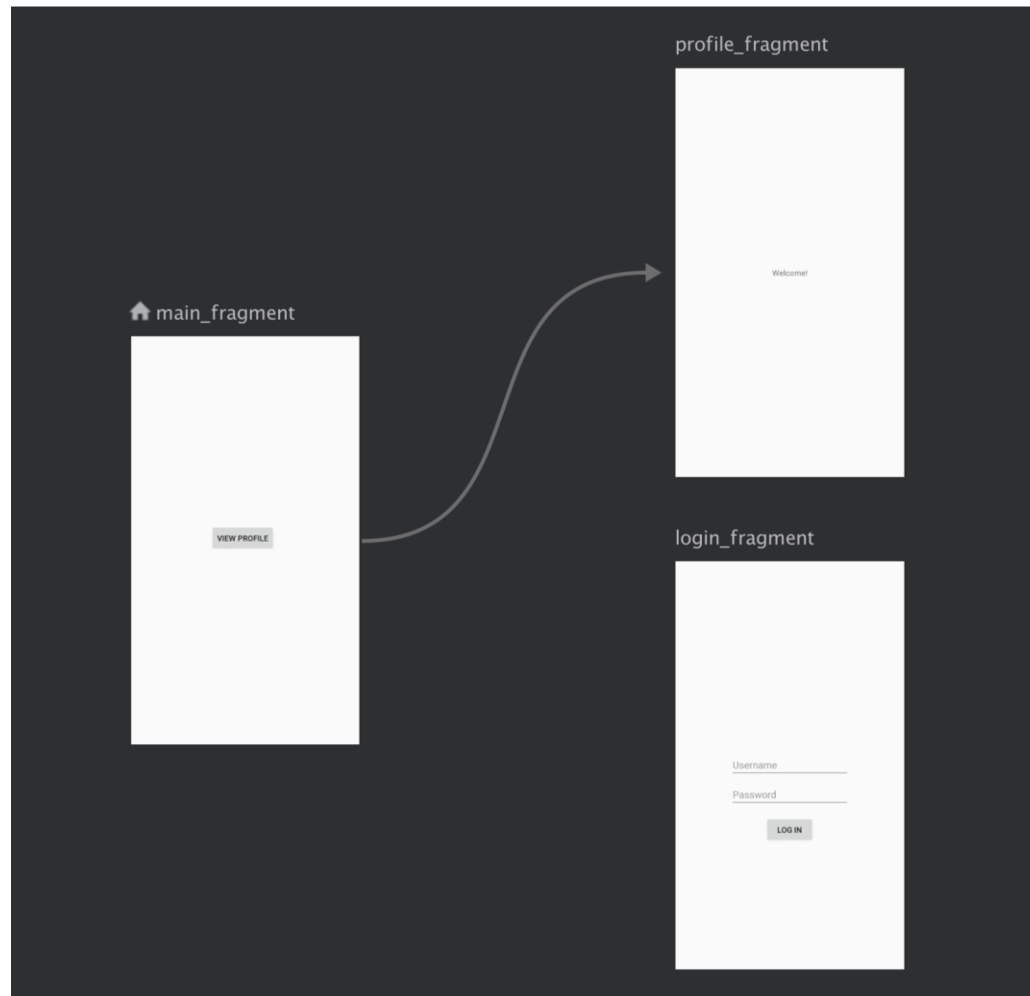
Navigation component

Componente

Graful de navigație - fișier XML care conține toate informațiile referitoare la navigare intre destinații

NavHost - container gol care afișează destinațiile din graf

NavController - un obiect care se ocupă de navigare intr-un NavHost



Navigare condiționată

Se pot seta diferite tipuri de argumente care vor fi procesate de către destinație

```
<fragment android:id="@+id/fragment">
          <argument
                    android:name="numeArgument"
                    app:argType="integer"
                    android:defaultValue="0" />
</fragment>
```

Safe Args

Generează obiecte simple și clase buildere pentru navigare care ajută la siguranța navigării, deoarece tipul pasat când se navighează este garantat ca va fi tipul corect

Se generează o clasă pentru fiecare destinație unde este o acțiune, are la final cuvântul **Destination**

Se generează o clasă pentru fiecare destinație, are la final **Args**

```
val action = ProductFragmentDirections.toDetails(productId)
findNavController().navigate(action)
```

Concluzii

Prin urmare, există un potențial imens de redactare al aplicațiilor Android, prin căutarea unor metode mai simple și mai eficiente de utilizare al bibliotecilor elaborate de specialiști in scopul redactării unor aplicații singulare

Bibliografie

https://en.wikipedia.org/wiki/Kotlin_(programming_language)
https://kotlinlang.org/docs/reference/
https://proandroiddev.com/migrate-to-gradle-kotlin-dsl-in-4-steps-f3e3b27e1f4d
https://developer.android.com/jetpack/docs/guide
https://en.wikipedia.org/wiki/Factory_method_pattern
https://en.wikipedia.org/wiki/Builder_pattern
https://en.wikipedia.org/wiki/Adapter_pattern
https://en.wikipedia.org/wiki/Composite_pattern
https://en.wikipedia.org/wiki/Mediator_pattern
https://en.wikipedia.org/wiki/Observer_pattern
https://en.wikipedia.org/wiki/Strategy_pattern
https://insert-koin.io/
https://developer.android.com/guide/navigation
https://developer.android.com/guide/navigation/navigation-pass-data