SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

NÁVRHOVÉ VZORY (STRATEGY, TEMPLATE METHOD, VISITOR) & SOLID

SEMINÁRNA PRÁCA

2022 Samuel Dubovec, Alan Dolán, Adam Kuchcik, Erik Masný, Marek Lunter

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

NÁVRHOVÉ VZORY (STRATEGY, TEMPLATE METHOD, VISITOR) & SOLID

SEMINÁRNA PRÁCA

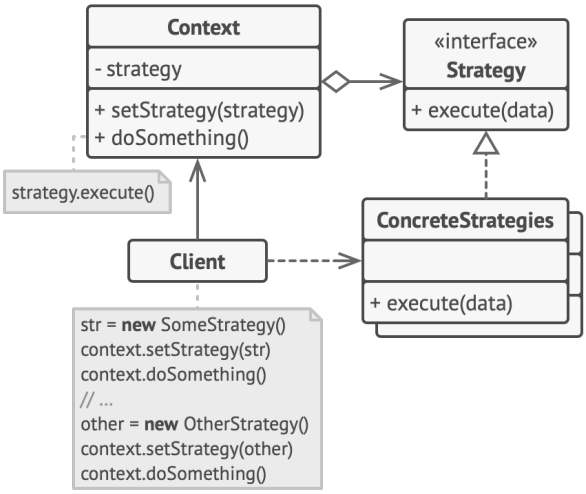
Študijný program:	Aplikovaná informatika
Predmet:	I–ASOS – Architektúra softvérových systémov
Prednášajúci:	RNDr. Igor Kossaczký, CSc.
Cvičiaci:	Ing. Stanislav Marochok

Bratislava 2022 **Samuel Dubovec, Alan Dolán, Adam Kuchcik, Erik Masný, Marek Lunter**

Obsah

1 Strategy pattern

Strategy je návrhový vzor, ktorý nám umožní definovať skupinu algoritmov, kde každý algoritmus bude definovaný v samostatnej triede. Následne vieme algoritmy vďaka využitiu kompozície namiesto dedičnosti zamieňať aj počas behu aplikácie. [1, 2]



Obr. 1: Strategy UML diagram

Context drží v sebe referenciu na jednu z konkrétnych Strategy, ktorá implementuje Strategy interface, čo nám dáva k prístup k execute(data) metóde v Context triede. Tá následne po zavolaní metódy doSomething() zavolá execute(data) metódu pre konkrétne nastavenú Strategy. Klient má možnosť si zvoliť konkrétnu Strategy sám. Taktiež vďaka použitiu kompozície je možné meniť Strategy počas behu programu. V rámci tried konkrétnych Strategy je definovaná samotná implementácia logiky.

1.1 Problém

Strategy nám pomáha riešiť problém keď máme skupinu algoritmov, ktoré robia tú istú vec, no každý iným spôsobom. Príkladom je platba kreditnou kartou. Visa, American Express čí Master card, každý ma svoj algoritmus vykonania platby definovaný svojim (iným) spôsobom. Taktiež čo sa týka validácie platby, uplatnenie zľiav resp. odmien pre rôzne typy zákazníkov. V tomto prípade môžeme uplatniť Strategy pattern. Klienta, v našom prípade je to trieda Customer z pohľadu banky, zaujíma jediná vec, a to je vykonanie platby. Ako je platba vykonaná, validovaná, resp. aké odmeny sú zákazníkovy pripísane, za využívanie napr. práve Visa karty, je implementačný detail, ktorý rieši konkrétna Strategy trieda. Klient vie vykonať platbu cez Context triedu, v našom prípade sa jedná o Payment triedu, ktorá v sebe drží referenciu na aktuálne zvolenú Strategy.

1.2 Riešenie

V našej aplikácii sme definovali si definovali Payment triedu, ktorá nám reprezentuje Context. Trieda Payment má v sebe referenciu na jednu z konkrétnych Strategy, v našom prípade je Strategy reprezentovaná rozhraním ICardStrategy. Konkrétne implementácie ICardStrategy sú napr. VisaCardStrategy alebo AmericanExpressCardStrategy. V týchto triedach je definovaná jedna funkcia pay(cardNumber, amount), ktorá musí byť implementovaná, keďže je definovaná v ICardStrategy rozhraní a konkrétne stratégie implementujú toto rozhranie. Context (Payment) má následne možnosť pri vytváraní inštancie triedy si definovať Strategy, ktorú chce použiť. Strategy je možné meniť aj počas behu programu cez setter metódu.
Keďže sme použili jazyk Kotlin, ktorý podporuje lambda funkcie, tak sme si vytvorili aj triedu LambdaCustomer (Lambda len z dôvodu odlíšenia od triedy Customer, ktorá využíva Strategy pattern). Trieda LambdaCustomer má v sebe referenciu na triedu LambdaPayment, ktorá reprezentuje Strategy formou funkcie, ktorú berie ako parameter cez konštuktor. V tele funkcie sa nachádza všetka logika, ktorá je následne po zavolaní funckie payByCreditCard v LambdaPayment vykonaná. Tento spôsob je možné použiť napríklad v prípade, keď je logika vykonania daného algoritmu (u nás platby) jednoduchá. Týmto spôsobom sa zbavíme všetkých ďalších tried a rozhraní (interfacov).

1.3 Výhody a nevýhody

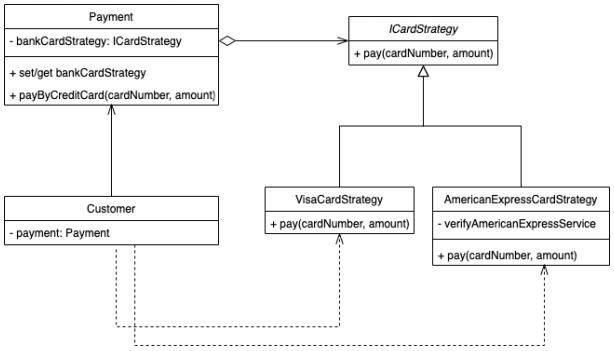
Výhody:

- Zmena typu algoritmu počas behu aplikácie
- Implementácia konkrétneho algoritmu je separovaná od kódu, ktorý ho používa
- Nahradenie dedenia kompozíciou
- Open/Closed princíp. Pridanie nových strategy (algoritmov) bez zmeny triedy, ktorá ich využíva

Nevýhody:

- Pri menšom počte algoritmov, ktoré sa nemenia často pridáme nové classy/interface, ktoré nám spravia kód zbytočne komplexnejším
- Klient musí poznať rozdiel medzi jednotlivými Strategy a musí si zvoliť správny
- V prípade modernejších jazykov môžeme Strategy treidy nahradit lambda (anonymnými) funkciami. Týmto sa vyhneme vytváraniu nových tried a rozhraní

1.4 Implementácia



Obr. 2: Strategy UML diagram platieb kartou

V aplikácii nám triedu Client reprezentuje trieda Customer.

```
class Customer(
    var payment: Payment,
    var lambdaPayment: LambdaPayment? = null,
)
```

Customer využíva triedu Payment, ktorá repreznetuje Context. Trieda Payment má v sebe referenciu na ICardStrategy, čo je rozhranie, ktoré je implementované všetkými konkrétnymi Strategy triedami. Taktiež má v sebe funkcie payByCreditCard(number, amount), ktorá len volá metódu pay(amount, number) z aktuálne zvolenej Strategy triedy.

```
class Payment(
    var bankCardStrategy: ICardStrategy,
) {
    fun payByCreditCard(number: String, amount: Double) {
        bankCardStrategy.pay(number, amount)
    }
}
```

VisaCardStrategy:

```
class VisaCardStrategy : ICardStrategy {
    override fun pay(cardNumber: String, amount: Double) {
        println("VISA card payment")
    }
}
```

AmericanExpressCardStrategy:

```
class AmericanExpressCardStrategy(
    private val verifyAmericanExpressCardService: VerifyAmericanExpressCardService
    ) : ICardStrategy {
    override fun pay(cardNumber: String, amount: Double) {
        try {
```

```
            verifyAmericanExpressCardService.verifyCard(cardNumber)
            println("American Express card payment")
        } catch (e: Exception) {
            println("American Express card payment not successful")
        }
    }
}
```

AmericanExpressCardStrategy využíva aj triedu VerifyAmericanExpressCardService, ktorá slúži len na ukážku, že jednotlivé Strategy môže mať injectnuté rôzne triedy.

```
class VerifyAmericanExpressCardService {
    fun verifyCard(number: String) {
        if (number.isEmpty() || number.length < 10) {
            throw Exception()
        }
    }
}
```

Čo sa týka použitia lamba resp. anonymných funkcií, tak tireda LambdaCustomer je implementovaná nasledovne:

```
class LambdaCustomer(
    var payment: LambdaPayment
)
```

Trieda LambdaPayment má v sebe referenciu paymentStrategy na funkciu (String, Double) -> Unit. táto funkcia reprezentuje kód konkrétnej platby. Funkcia payByCreditCard(nubmer, amount) len aplikuje funkciu paymentStrategy na parametre number a amount.

```
class LambdaPayment(
    var paymentStrategy: ((String, Double) -> Unit),
) {
    fun payByCreditCard(number: String, amount: Double) {
        paymentStrategy(number, amount)
    }
}
```

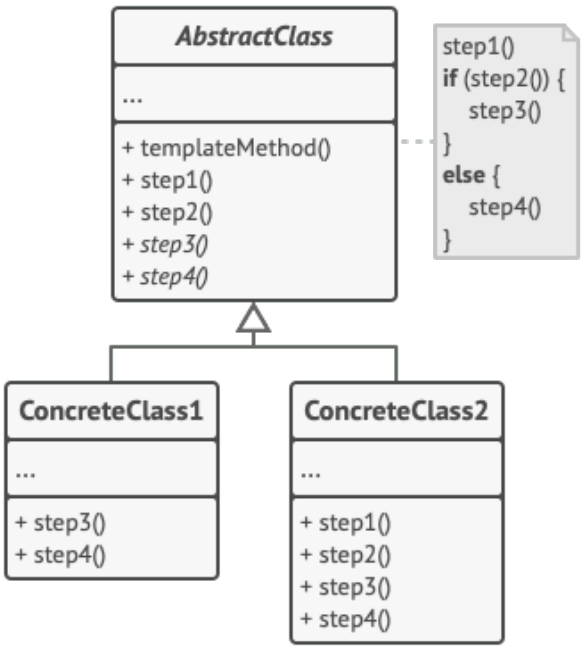
Čo sa týka ukážky kódu, vytvorenie nejakej Strategy, jej zmenu a použitie anonymných funkcií, výsledný kód vyzerá nasledovne:

```
val creditCardNumber = "12345"
val bank = Bank(
    Customer(
        payment = Payment(
            VisaCardStrategy()
        )
    )
)
// Strategy
bank.customer.payment.payByCreditCard(creditCardNumber, 10.0)

bank.customer.payment.bankCardStrategy =
```

```
    AmericanExpressCardStrategy(
        VerifyAmericanExpressCardService()
    )
bank.customer.payment.payByCreditCard(creditCardNumber, 12.0)
// Strategy

// Strategy with lambdas
val lambdaPayment = LambdaPayment { cardNumber: String, amount: Double ->
    if (cardNumber.isNotEmpty() && amount > 0.0) {
        println("MasterCard payment successful")
    } else {
        println("Payment failed")
    }
}
bank.customer.lambdaPayment = lambdaPayment
bank.customer.lambdaPayment!!.payByCreditCard(creditCardNumber, 15.0)
// Strategy with lambdas
```

2 Template method pattern

Template method je návrhový vzor, kde v abstraktnej triede sa navrhne algoritmus a podtriedy tento algoritmus dedia a môžu jeho samostatné časti prepisovať, avšak kostra algoritmu musí ostať zachovaná. Ako vidíme máme zadefinovanú abstraktnú triedu, ktorá v



Obr. 3: Template method UML diagram

sebe drží metódy, ktoré su postupnosťou nejakého algoritmu. Podtriedy konkrétne metódy dedia, avšak nevytvára ju nové metódy iba upravujú pôvodné zdedené. Template method navrhuje, aby sme algoritmus rozdelili na sériu krokov, tieto kroky prmenili na metódy a vložili tieto metódy do jednej triedy. Tieto metódy môžu byť abstraktné alebo môžu mať nejaku predvolenú implementáciu. Na použitie algoritmu sa predpokladá, že klient použije podtriedu, ktorá implementuje všetky abstraktné metódy a v prípade potreby prepíše niektoré z voliteľných.

2.1 Problém

V aplikácií bankovníctva vedieme výplaty zamestnancov. Máme rôžne pozície ako bankár alebo manažér a títo zamestnanci dostávajú výplaty podľa jedného algoritmu. Avšak manažéri majú iné výplaty a taktiež môžu dostávať iné bonusy a preto je vhodné zvoliť template method.

2.2 Riešenie

V aplikácií sme zadefinovali abstraktnú triedu PayingEmployees, ktorá ma zadefinované metódy, tie ktoré sa budú v podtriedach veľmi líšiť sme zadefinovali abstraktné a tie, ktoré budú mať veľmi podobné až identické majú už napísanú implementáciu. Máme vytvorené dve potriedy PayingManager a PayingBanker, ktoré dedia od triedy PayingEmployees a musia prepísať abstraktné metódy. Taktiež pre ukážku trieda PayingManager prepisuje aj nie abstraktnú metódu.

2.3 Výhody a nevýhody

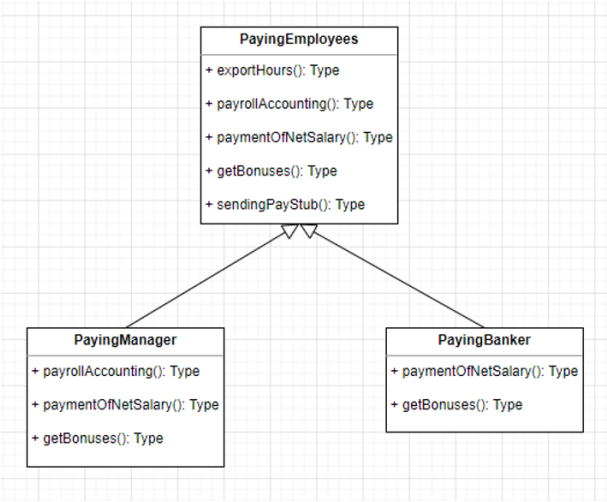
Výhody:

- Klientom môžeme povoliť aby prepísali iba určité časti veľkého algoritmu
- Duplicitný kód môžeme napísať len v abstraktnej triede

Nevýhody:

- Ťažšie sa udržiava pri väčšom algoritme s viac metódami
- Niektorých klientov môže obmedzovať poskytnutá „kostra" algoritmu.

2.4 Implementácia



Obr. 4: Template method UML diagram platenia zamestnancov

Trieda PayingEmployees

```
class Customer(
    fun exportHours(){}
    fun payrollAcoounting(){}
    abstract fun paymentOfNetSalary(){}
    abstract fun getBonuses(){}
    fun sendingPayStub(){}
)
```

Trieda PayingManager

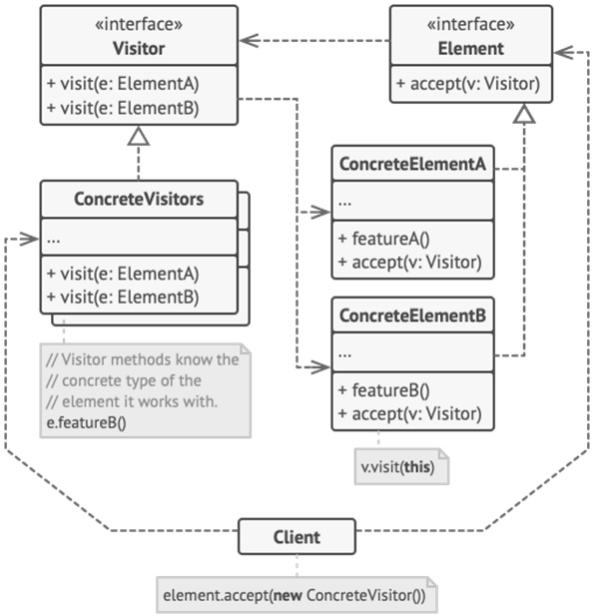
```
class Customer(
    fun payrollAcoounting(){}
    abstract fun paymentOfNetSalary(){}
    abstract fun getBonuses(){}
)
```

Trieda PayingBanker

```
class Customer(
    abstract fun paymentOfNetSalary(){}
    abstract fun getBonuses(){}
)
```

3 Visitor

Návrhový vzor Visitor, slúži ako nástroj pre vytváranie nových operácií prislúchajúcich k istému objektu bez toho aby sme menili jeho štruktúru.



Obr. 5: Visitor UML diagram

Visitor interface pozostáva z viacero metód, najčastejšie s označením 'visit(Element)', ktoré ako parameter akceptujú element, ktorý rozširujeme. Každý konkrétny Visitor, ktorý implementuje tento interface obsahuje jedinečnú funkcionalitu priamo závislú na vstupnom elemente.

Elementy, ktoré rozširujeme s využitím Visitor-a musia obsahovať metódu, najčastejšie s označením 'accept(Visitor)', ktorá ako parameter akceptuje Visitor-a rozširujúceho funkcionalitu daného elementu. Táto metóda vyvoláva metódu 'visit()' na Visitor nachádzajúci sa v parametri metódy a pošle inštanciu samého seba ako parameter.

Teda konkrétny Visitor rozširuje funkcionalitu Elementov v metódach 'visit(Element)' a Element akceptuje túto funkcionalitu pomocou metódy 'accept(Visitor)'.

3.1 Problém

Môžeme si zobrať do úvahy situáciu, kedy máme k dispozíciu niekoľko naimplementovaných tried, ktoré dedia od rovnakej triedy typu 'interface'. V prípade, že sa rozhodneme rozšíriť daný 'interface' a nevyužiť Visitor ako návrhový vzor, vzniká situácia kedy musíme každú tried aktualizovať na základe novej funkcionality. Ak budeme takto pokračovať aj pri

ďalších nových rozšíreniach stáva sa takto implementovaný kód ťažko udržateľným a taktiež porušujeme 'open-closed' princíp, keďže modifikujeme originálny kód.

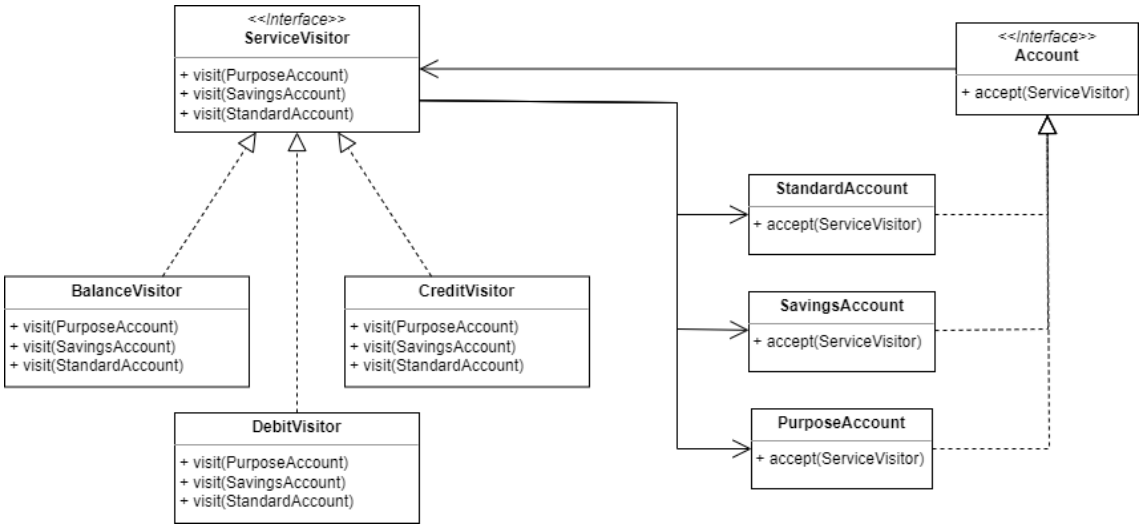
3.2 Riešenie

Ako príklad riešenia tohto problému sme zvolili situáciu, kedy predpokladáme, že exituje niekoľko typov bankových účtov (bežný, sporiaci a účelový) a niekoľko služieb (príjem a výber finačných prostriedkov a kontrola stavu na účte) vzťahujúcich sa na tieto bankové účty.

Každý špecifický bankový účet je reprezentovaný triedou, ktorá zároveň implementuje základnú triedu bankového účtu. Namiesto toho aby sme pre každý bankový účet vytvárali funkcionalitu priamo v tejto hierarchii vytvorili sme novú hierarchiu pre spomínané služby, kde podobne každá služba je reprezentovaná triedou, ktorá implementuje základnú triedu služby.

Triedam služieb sme pridelili metódy 'visit(arg)', ktoré na základe bankového účtu v parametri metódy, vykonajú požadovanú službu.

Triedam bankových účtov sme pridelili metódu 'accept(ServiceVisitor)' vyvolávajúcu metódu 'visit()' z objektu služby, ktorý sa nachádza v parametri metódy.



Obr. 6: Visitor UML diagram banky

3.3 Implementácia - ukážka

Účelový účet

```
class PurposeAccount : Account {
    override fun accept(serviceVisitor: ServiceVisitor) {
        serviceVisitor.visit(this)
```

```
    }
}
```

Služba zobrazenia stavu účtu

```
class BalanceVisitor : ServiceVisitor {
    override fun visit(purposeAccount: PurposeAccount) {
        println("Showing balance of purpose account.")
    }
    override fun visit(savingsAccount: SavingsAccount) {
        println("Showing balance of savings account.")
    }
    override fun visit(standardAccount: StandardAccount) {
        println("Showing balance of standard account.")
    }
}
```

Ukážka využitia visitora

```
fun runVisitorExample() {
    println("\nRunning visitor example...")

    val purposeAccount: Account = PurposeAccount()
    val balanceVisitor: ServiceVisitor = BalanceVisitor()

    purposeAccount.accept(balanceVisitor)
}
```

3.4 Výhody a nevýhody

Výhody:

- Spĺňa single-responsibility princíp.
- Spĺňa open-closed princíp.
- Hromadí užitočné informácie o objektoch na jedno miesto.

Nevýhody:

- Aktualizácia všetkých „visitorov“ pri pridaní/odstránení prvkov do/z hierarchie.
- Visitor nemusí mať prístup k „private“ premenným a metódam.
- Kód pre jeden objekt je rozložení v jednotlivých „visitoroch“.

4 SOLID

V softvérovom inžinierstve je SOLID skratka pre päť princípov návrhu, ktoré majú urobiť objektovo orientované návrhy zrozumiteľnejšími, flexibilnejšími a udržiavateľnejšími. Tieto zásady sú podmnožinou mnohých zásad, ktoré propaguje americký softvérový inžinier a inštruktor Robert C. Martin a ktoré prvýkrát predstavil v roku 2000 vo svojom článku Design Principles and Design Patterns.

SOLID princípy sú:

- The single-responsibility principle
- The open–closed principle
- Liskov substitution principle
- The interface segregation principle
- The dependency inversion principle

4.1 The single-responsibility principle

The single-responsibility principle je princíp počítačového programovania, ktorý hovorí, že každá trieda, modul alebo funkcia v programe by mala mať jednu zodpovednosť/účel v programe. Bežne používaná definícia znie: "každá trieda by mala mať len jeden dôvod na zmenu".

4.2 The open–closed principle

Tento prinzíp hovorí, že softvérové entity (triedy, moduly, funkcie atď.) by mali byť otvorené pre rozšírenie, ale uzavreté pre modifikáciu, to znamená, že takáto entita môže umožniť rozšírenie svojho správania bez toho, aby sa zmenil jej zdrojový kód. Názov open-closed sa používa vo dvoch významoch. Oba spôsoby využívajú na riešenie zdanlivej dilemy zovšeobecnenia (napríklad dedičnosť alebo delegované funkcie), ale ciele, techniky a výsledky sú odlišné.

4.3 Liskov substitution principle

Tento princíp uvádza, že objekty rodičovskej triedy by sa mali dať nahradiť objektami jej podtried bez toho, aby sa aplikácia rozbila. Inými slovami, chceme, aby sa objekty našich podtried správali rovnako ako objekty našej nadtriedy.

4.4 Interface segregation princpiple

Interface segregation principle je princíp, ktorý hovorí o tom, že žiaden kód by nemal byť závislý na metódach, ktoré musí implementovať. V praxi to znamená, že objekty typu interface by mali byť dostatočne granulárne na to, aby odovdené objekty nemusli implementovať metódy, ktoré nie sú nutné. Je preto lepšie rozdeliť komplexné interface objekty na viacero menších, ktoré obsahujú iba prislúchajúce metódy. Odvodené objekty následne implementujú všetky potrebné objekty typu interface.

4.5 Dependency inversion principle

Dependency inversion principle je princíp, ktorý hovorí, že objekty vyššieho levelu by nemali byť závislé na objektoch nižšieho levelu. V prípade že objekt vyžaduje využitie iného objektu ako závislosti, mala by byť táto závislosť abstraktná a nahraditeľná iným typom danej závislosti.

4.6 Dependency injection

Dependency injection je softvérový návrhový vzor, ktorý hovorí o tom, že závislosti pre daný objekt by mali byť doň vkladané, a nie byť ním vytvárané. Takýmto spôsobom dokážeme zabezpečiť nezávislosť medzi objektami a zbavíme sa zodpovednosti za vytváranie závislostí v danom objekte.

Zoznam použitej literatúry

1. GURU, Refactoring (zost.). *Strategy: Strategy design pattern* [online]. Refactoring guru [cit. 2022-10-09]. Dostupné z : `https://refactoring.guru/design-patterns/strategy`.
2. ROBSON, Eric Freeman Elisabeth. *Head First Design Patterns, 2nd Edition: Head First Design Patterns, 2nd Edition*. Second. O'Reilly Media, Inc): O'Reilly Media, Inc, 2021. ISBN 978-1-492-07800-5.

Prílohy