SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DESIGN PATTERNS, CREATIONAL PATTERNS
SEMINÁRNA PRÁCA

2022

Marián Šebeňa
Dominik Vozár
Patrik Šály
Martin Smetanka
Simon Youssef

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DESIGN PATTERNS, CREATIONAL PATTERNS
SEMINÁRNA PRÁCA

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

Bratislava 2022

Marián Šebeňa
Dominik Vozár
Patrik Šály
Martin Smetanka
Simon Youssef

Obsah

Úvod

Návrhové vzory sa zaoberajú spôsobom vytvárania objektov. Tieto vzory sa používajú vtedy, keď je potrebné prijať rozhodnutie v čase vytvárania objektov triedy

Nemenný kód nie je dobrý programátorský prístup a Niekedy je potrebné zmeniť povahu objektu podľa povahy programu. V takýchto prípadoch si musíme vziať na pomoc tvorivé návrhové vzory, ktoré poskytujú všeobecnejší a flexibilnejší prístup.

Návrhové vzory su programátorske konvencie, ktoré pomáhajú udržiavať princípy objektovo orientovaného programovania.

V dokumentácii sa venujeme nasledovným tvorivým návrhovým vzorom a ich implementácii:

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Pre implementáciu vzorov sme vytvorili jednoduchú konzolovú aplikáciu, ktorá simuluje vytváranie objednávok jedla. Používateľ si vyberie typ jedla, ktoré sa následne vytvorí jedným s horeuvedených návrhových vzorov.

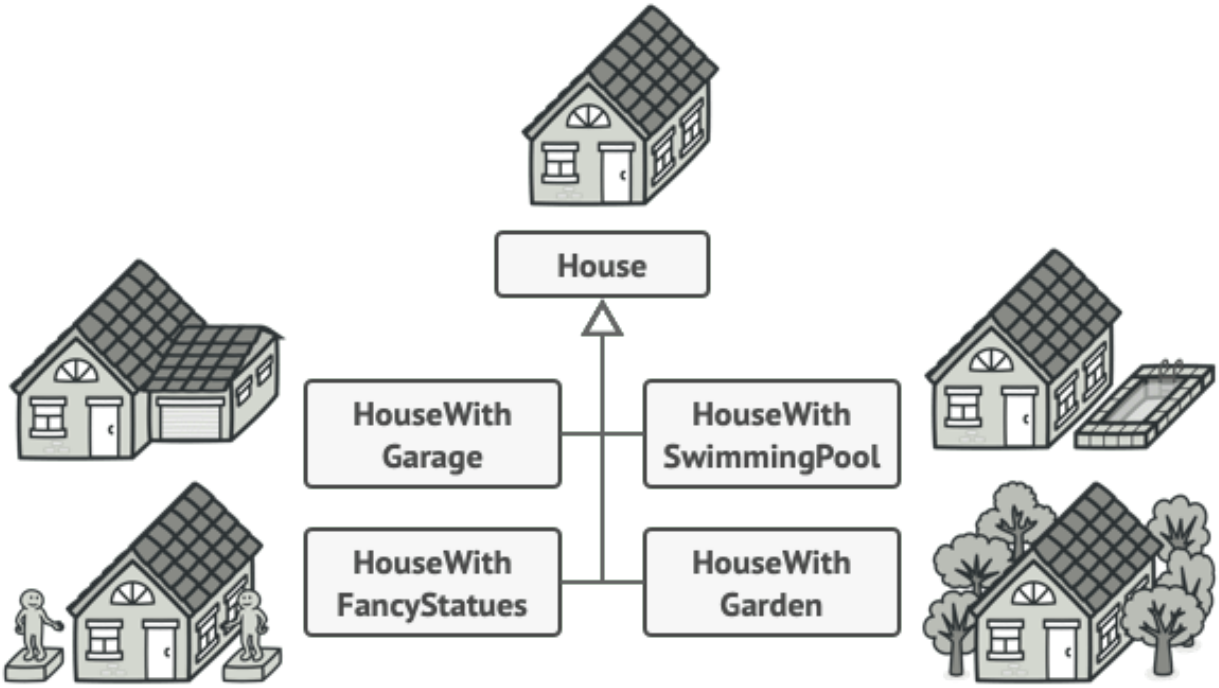
1 Builder pattern

Vzor Builder je návrhový vzor, ktorý poskytuje flexibilné riešenie na vytváranie objektov. Builder oddeľuje konštrukciu komplexného objektu od jeho reprezentácie. Vytvára komplexný objekt pomocou jednoduchých objektov tým, že poskytuje postupný prístup.

1.1 Využitie vzoru Builder

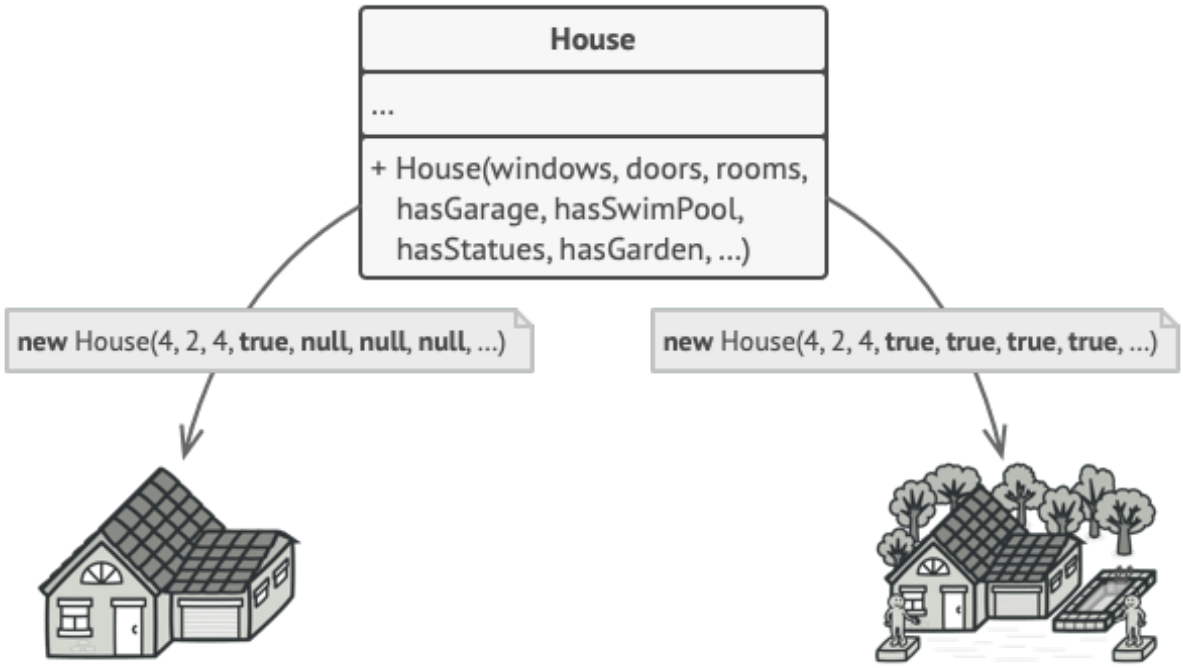
Ako príklad využitia návrhového vzoru Builder si môžme zobrať vytvorenie objektu Dom. V normálnom prípade ak by sme chceli postaviť jednoduchý dom, stačí nám postaviť 4 steny, podlahu, strechu, nainštalovať dvere a osadiť okná.

V prípade, že chceme náš dom vylepšiť napríklad o garáž alebo o bazén, tak jedným z najjednoduchších riešení je rozšíriť základnú triedu Dom a vytvoriť súbor podtried, ktoré pokryjú všetky kombinácie parametrov. S týmto riešením by sme ale skončili s veľký počtom podtried. Taktiež pridanie ďalšieho parametru by si vyžadovalo rozšírenie existujúcej hierarchie.



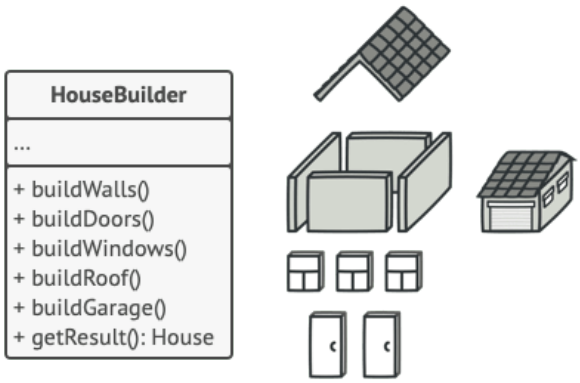
Obr. 1: Ukážka rozšírenia triedy Dom podtriedami.

Ďalším riešením by bolo vytvoriť obrovský konštruktor priamo v triede Dom so všetkými parametrami, ktoré patria k domu. V tomto riešení by väčšina parametrov nebola vždy využitá.



Obr. 2: Ukážka využitia konštruktora v triede Dom.

Vzor Builder navrhuje, aby sa vyčlenil kód konštrukcie objektu z jeho vlastnej triedy a presunul sa do samostatných objektov nazývaných buildery. Tento návrhový vzor usporiadava konštrukciu objektu do súboru takzvaných krokov. Ak chceme vytvoriť objekt, tak vykonáme postupnosť týchto krokov na objekte builder. Nie je potrebné volať všetky kroky, stačí zavolať len tie, ktoré sú potrebné na vytvorenie konkrétnej konfigurácie objektu. Nie všetky kroky konštrukcie budú rovnaké, niektoré môžu vyžadovať odlišnú implementáciu. V tom prípade môžeme vytvoriť niekoľko rôznych tried Builder, ktoré implementujú rovnaký krok, ale rôznym spôsobom.



Obr. 3: Ukážka využitia vzoru Builder v triede Dom.

1.2 Director

Trieda director je samostatná trieda, ktorá sa používa na vyčlenenie série volaní krokov Buildera, ktoré sa používa na zostavenie výrobku. Trieda director definuje poradie, v ktorom sa majú vykonávať kroky buildera, zatiaľ čo builder poskytuje implementáciu týchto krokov. Samotná trieda director nie je vyžadovaná. Kroky buildera sa môžu volať aj priamo z kódu klienta, avšak trieda director môže byť dobrým miestom na zadefinovanie rôznych konštrukčných rutín, ktoré môžu byť opakovane použité v kóde.

1.3 Výhody a nevýhody

Medzi výhody vzoru Builder patrí:

- Objekty sa môžu konštruovať postupne, spúšťať rekurzívne alebo odkladať.
- Rovnaký konštrukčný kód je možné opakovane použiť pri konštrukcií rôznych produktov.
- Poskytuje lepšiu kontrolu nad procesom konštrukcie.

Medzi nevýhody patrí:

- Celková komplexita kódu sa zvyšuje, z dôvodu, že vzor Builder vyžaduje vytvorenie nových tried.
- Duplicita kódu, pretože Builder kopíruje polia z triedy, z ktorej vytvára objekt.

1.4 Implementácia

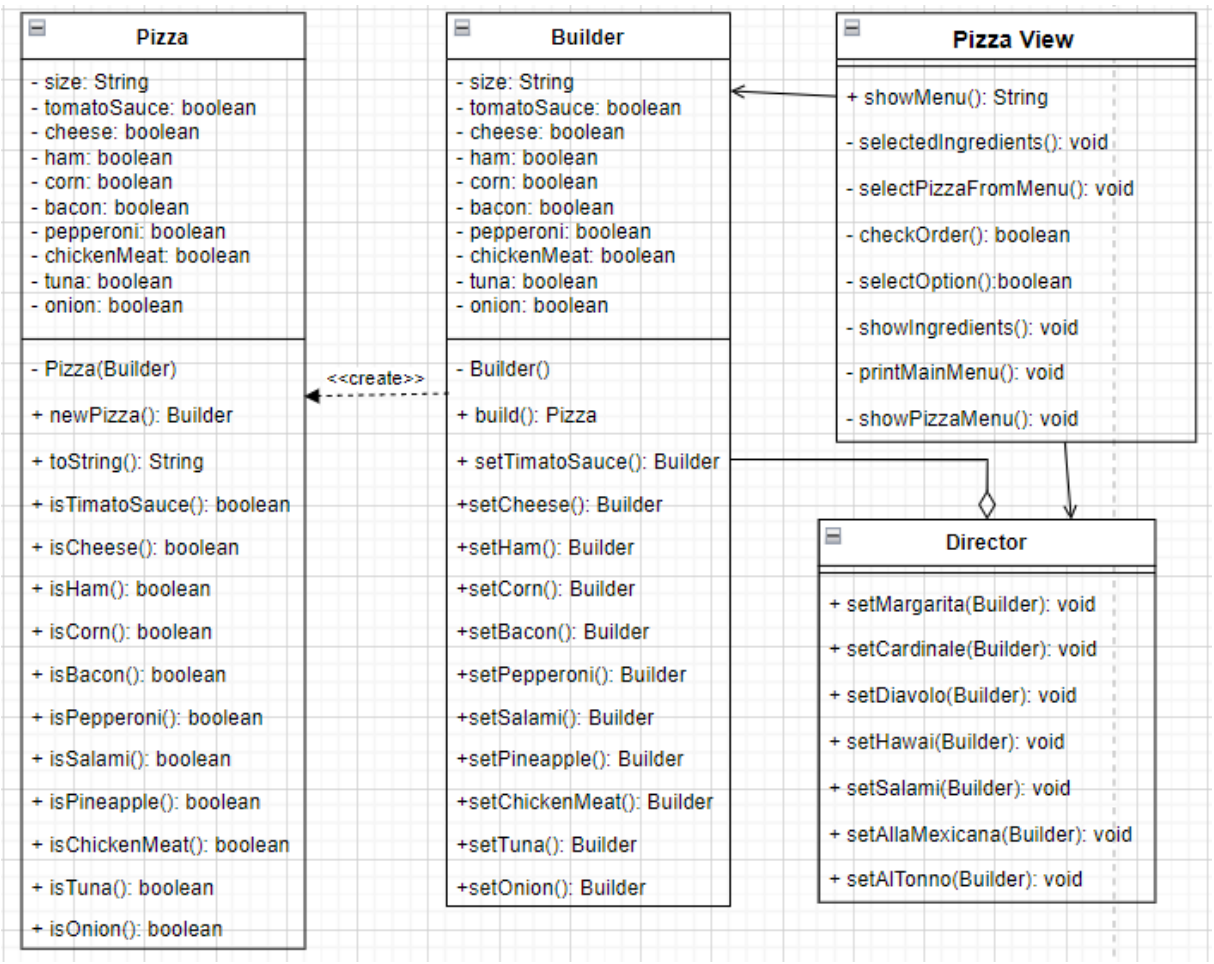
Na ukážku vzoru Builder sme vytvorili jednoduchú konzolovú aplikáciu, ktorá umožní si vybrať ingrediencie na pizzu a jej veľkosť. Aplikácia pozostáva z štyroch tried - Pizza, Builder, Pizza View a Director. V triede Pizza sa nachádza statická metóda newPizza na vytvorenie nového objektu Builder, taktiež aj metóda toString na vypísanie ingrediencii na pizzi a jej veľkosti. Trieda Pizza obsahuje aj konštruktor, ktorý prijíma objekt Builder a nastaví svojím premenným rovnakého hodnoty, aké sa nachádzajú v zadanom objekte Builder.

V rámci triedy Pizza sa nachádza statická trieda Builder, ktorá slúži na zostavenie objektu. Pozostáva z viacerých metód na nastavenie premenných, ktoré vracajú objekt Builder. Obsahuje aj metódu build, ktorá vytvorí nový objekt Pizza s nastavenými údajmi objektu Builder a taktiež metódu reset na vyresetovanie ingrediencii. Trieda Builder obsahuje aj privátny konštruktor, ktorý slúži na to aby bolo možné vytvárať nové inštancie iba z Builder triedy.

Trieda Director sa skladá z viacerých metód, ktoré prijímajú objekt Builder a zostavujú preddefinovanú pizzu danému Builderu.

Trieda Pizza View slúži na zobrazenie jednoduchého konzolového rozhrania pre používateľa, kde si môže vyskladať vlastnú pizzu, prípadne si vybrať nejakú preddefinovanú.

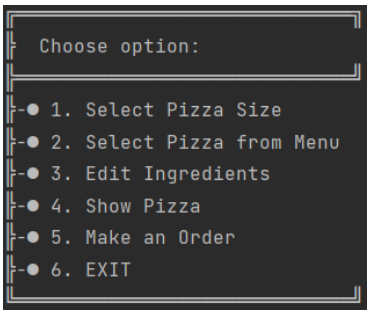
UML diagram opísanej implementácie môžete vidieť na obrázku č. 4.



Obr. 4: Uml diagram opísaného kódu.

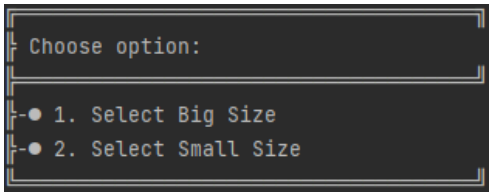
1.4.1 Používateľské rozhranie

Pri spustení kódu sa spustí jednoduchá interaktívna konzolová aplikácia. Konzolová aplikácia prijíma na vstup čísla, bez bodky, ktoré sú zobrazené v jednotlivých možnostiach, po zadaní čísla je potrebné ešte stlačiť enter. Na začiatku sa zobrazí základné menu, ktoré môžete vidieť na obrázku č.7. Základné menu obsahuje 5 možností ako je vidno na obrázku, pri stlačení čísla, ktoré sa nachádza pri danej možnosti sa vykoná popísaná možnosť, vo väčšine prípadov sa zobrazí ďalšie menu, v prípade, že sa stlačí číslo 6, tak program sa ukončí.



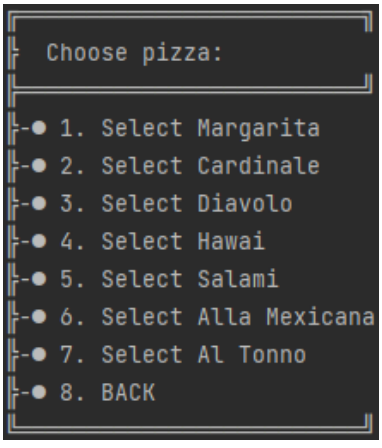
Obr. 5: Základné menu v UI.

Pri stlačení čísla 1 sa zobrazí menu, ktoré môžete vidieť na obrázku č.6 . Toto menu sa skladá z 2 možností. Pri stlačení čísla 1 sa nastaví veľkosť pizze na veľkú a pri stlačení čísla 2 sa nastaví veľkosť pizze na malú. Po vybratí ľubovoľnej z možnosti sa zobrazí pôvodné základné menu.



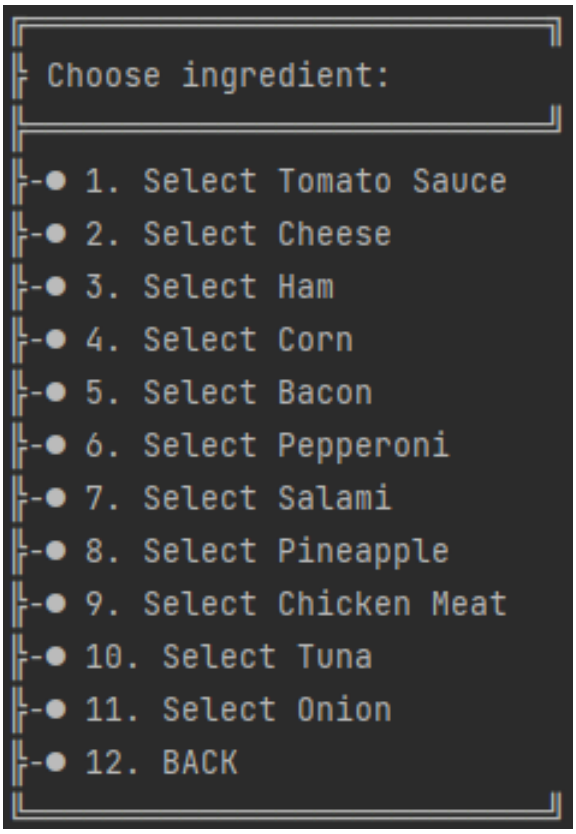
Obr. 6: Menu na výber veľkosti pizze v UI.

Pri výbere možnosti 2 v základnom menu sa zobrazí menu pízz, ktoré majú už preddefinované ingrediencie. Toto menu môžete vidieť na obrázku č.7. V tomto menu sa nachádza 7 pízz s rôznymi ingredienciami. Po stlačení ľubovolného čísla z menu sa objaví základné menu. Pri výbere jednej z pízz v menu sa vyresetujú všetky pôvodne nastavené ingrediencie a nastavia sa konkrétne ingrediencie, ktoré má daná pizza zadefinované.

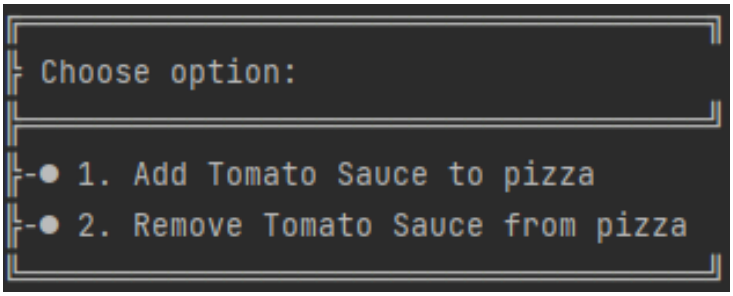


Obr. 7: Menu na výber preddefinovaných pízz v UI.

Pri stlačení čísla 3 v základnom menu sa zobrazí menu ingrediencií, prostredníctvom ktorého je možné pristúpiť k pridávaniu alebo odoberaniu, konkrétnych ingrediencii na pizze. Toto menu môžete vidieť na obrázku č.8 , nachádza sa tam 11 rôznych ingrediencii. Pri stlačení jedného z čísiel 1 až 11 sa zobrazí menu na úpravu konkrétnej ingrediencii. Toto menu obsahuje 2 možnosti ako môžete vidieť na obrázku č.9 . Pri stlačení čísla 1 sa pridá daná ingrediencia na pizzu, v opačnom prípade odoberie. Pri výbere ľubovoľnej z možností sa zobrazí pôvodné menu ingrediencii. Pri stlačení čísla 12 v menu ingrediencii sa zobrazí základné menu.

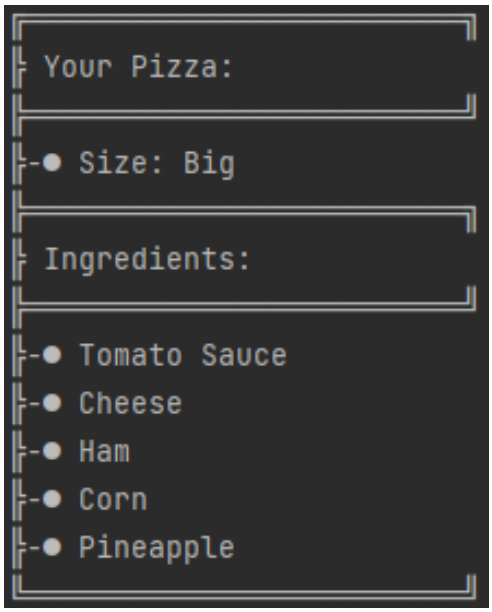


Obr. 8: Menu na výber ingrediencii v UI.



Obr. 9: Menu na úpravu konkrétnej ingrediencie v UI.

Ak sa stlačí číslo 4 v základnom menu, tak sa zobrazí používateľova pizza s vybranými ingredienciami a veľkosťou. Zároveň sa zobrazí aj základné menu. Ukážku tejto možnosti môžete vidieť na obrázku č.10 .



Obr. 10: Zobrazenie vybranej pizze v UI.

Pri stlačení čísla 5 v základnom menu, sa vykoná objednávka a zobrazí sa hlavné menu aplikácie a objednávka sa uloží do súboru.

2 Singleton Pattern

Vzor Singleton je návrhový vzor, ktorý uisťuje, že trieda má iba jednu inštanciu a, že je ku nej globálny prístup. Z definície vyzerá ako jednoduchý návrhový vzor, avšak implementácia je zložitejšia. Trieda Singleton musí byť zabezpečená proti vytváraniu viac ako jednej inštancie triedy, modifikácií, refaktoringu, deserializácií alebo proti klonovaniu. Návrhový vzor Singleton taktiež zabezpečuje výkonnosť aplikácie, keď nevytvára nové inštancie triedy. Získanie inštancie z cache pamäti je vždy optimálnejšie z hľadiska výkonu aplikácie ako vytváranie nových inštancií pokiaľ ich nie je treba. Klienty často ani nevedia, že pristupujú zakaždým k rovnakému objektu a nie k novému. Medzi ďalšie výhody Singlton patternu patrí, že dokážeme opakujúcu sa funkcionalitu implementovať do jednej triedy, z ktorej sa následne môže vykonávať

2.1 Využitie vzoru Singleton

Príkladom využitia návrhového vzoru Singleton môže byť vytvorenie jednotného prístupu ku zdieľanému úložisku, ako je databáza, súbor, ... Návrhový vzor Singleton môžeme využiť aj, keď chceme zvýšiť kontrolu nad globálnou premennou.

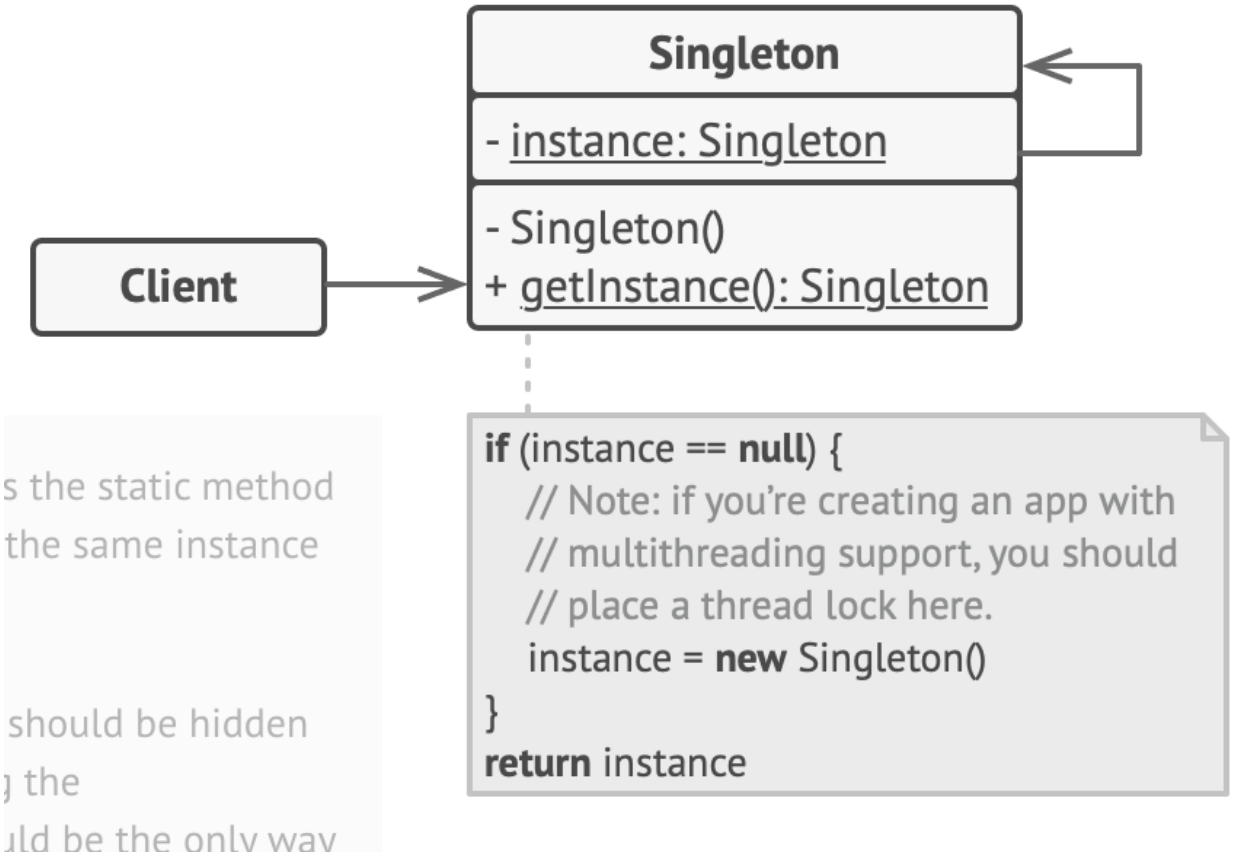
2.2 Štruktúra

Na obrázku nižšie môžeme vidieť štruktúru Singleton patternu. Klient môže pristupovať k inštancií triedy len pomocou metódy getInstance(). Trieda getInstance zistí, či je vytvorená inštancia triedy Singleton. Ak nie je vytvorená, následne sa vytvorí a vráti sa pre klienta. Ak je vytvorená, iba sa vráti. Objekt sa ukladá v privátnej statickej premennej instance.

2.3 Implementácia

Singleton implementujeme v nasledujúcich krokoch.

- Vytvorenie privátnej statickej premennej na uloženie inštancie Singletonu.
- Deklarovanie public static metódy, ktorá bude zodpovedná, za vytvorenie a vrátenie singletonu pokiaľ ešte neexistuje alebo vrátenie singletonu, pokiaľ už jeho inštancia je vytvorená.
- Označenie konštruktora ako privátny.
- Dodatočná ochrana pri multivláknových aplikáciach. Vytvorenie serializable bloku v jazyku Java.



Obr. 11: Štruktúra Singleton patternu

- Zabránenie refaktorizácie.
- Zabránenie deserializácie.
- Zabránenie cloningu.

2.4 Výhody a nevýhody

Medzi výhody vzoru Builder patrí:

- Môžeme si byť istý, že trieda má iba jednu inštanciu.
- Získame globálny prístup k inštancii.
- singleton je vytvorený iba jedenkrát. Iba vtedy, keď ho potrebujeme.

Medzi nevýhody patrí:

- Porušuje zásadu Single Responsibility Principle. Vzor rieši dva problémy v čase.

- Vzor Singleton môže maskovať zlý dizajn. Napríklad keď komponenty vedia o sebe príliš veľa.
- Vzor si vyžaduje špeciálne zaobchádzanie vo viacvláknovom prostredí, aby viaceré vlákna nevytvorili viac Singleton inštancií.
- Môže byť zložité testovať klientsky kód Singletonu, pretože mnohé testovacie rámce sa pri výrobe falošných objektov spoliehajú na dedičnosť. Keďže konštruktor triedy singleton je súkromný a prepísanie statických metód je vo väčšine jazykov nemožné, budete musieť vymyslieť kreatívny spôsob, ako zosmiešniť singleton. Alebo jednoducho nepíšte testy. Alebo nepoužívajte vzor Singleton.

2.5 Kód

Na ukážku vzoru Singleton v praxi sme vytvorili simuláciu do databázového pripojenia. Trieda DataSource v konečnom dôsledku dokáže iba zapisovať objednávky do textového súboru. Zápis objednávok z aplikácie využívajú implementované komponenty predstavujúce rôzne jedlá. Pri vytvorení inštancie Singletonu sme simulovali vytvorenie databázového pripojenia s testovacím oneskorením 1000ms. Singleton pattern sme naprogramovali podľa sekcie Implementácia v sekcií vyššie. Avšak sme nemuseli zabezpečovať aplikáciu proti serializácii ani cloningu pretože neimplementujeme rozhrania Clonable a Serializable.V nasledovnom obrázku vidíme metódu getInstance(), ktorá zabezpečuje funkcionalitu takzvanej lazy inicializácie.

```
public static DataSource getInstance() {
    if (instance == null){
        instance = new DataSource();
    }
    return instance;
}
```

Obr. 12: Lazy inicializácia Singletonu

Na obrázku nižšie môžeme vidieť implementáciu simulácie databázového pripojenia a zabezpečenie proti reflexii.

Na konci sme implementovali metódu, ktorá uzatvára spojenie.

```
private DataSource() {
    if (instance != null) {
        throw new RuntimeException();
    }
    try {
        connection = new BufferedWriter(new FileWriter( fileName: "orders.txt",  append: true));
        Thread.sleep( millis: 1000);
    } catch (Exception e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

Obr. 13: Simulácia databázového pripojenia a zabezpečenie proti reflexii

```
public void closeFileConnection() {
    try {
        connection.close();
    } catch (IOException ignored) {}
}
```

Obr. 14: Uzatvorenie spojenia so súborom

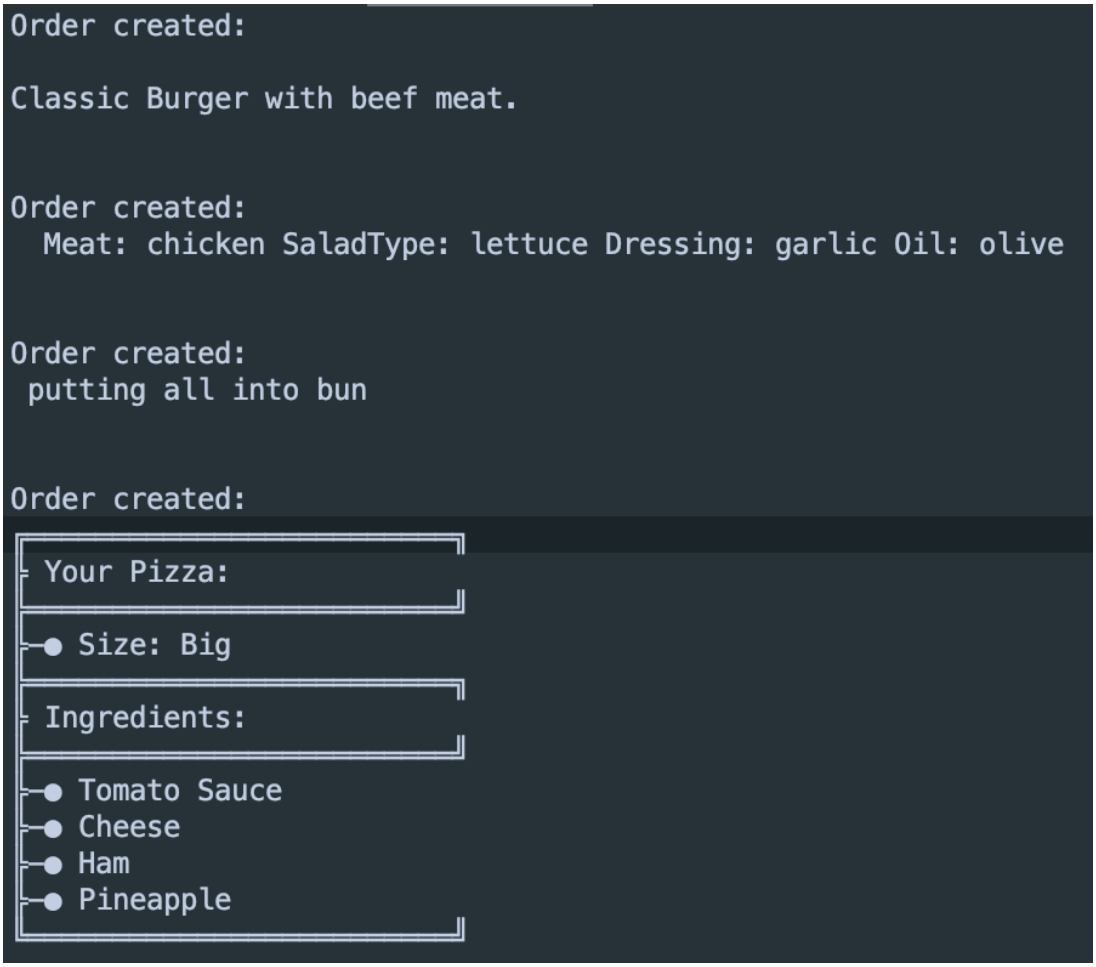
Náš singleton okrem zabezpečenia svojej jedinečnosti a globálneho prístupu implementuje funkciu writeData(). Funkcia writeData zabezpečuje zápis objednávok do súboru orders.txt.

Implementáciu funkcie writeData môžeme vidieť na nasledovnom obrázku.

```
@Override
public void writeData(String orderName) {
    try {
        connection.write( str: "Order created:\n " + orderName + "\n\n\n");
        System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

Obr. 15: Funkcia writeData()

Obsah súboru orders.txt môžeme vidieť na nasledovnom obrázku.



Obr. 16: Obsah súboru orders.txt

2.6 Príklady implementácie Singletonu

V nasledovnej sekcií sme vypracovali príklady implementácie Singletonu v jazyku Java. Programátor si môže vybrať ktorú implementáciu si zvolí a ako ju zúži a rozšíri. V každom jazyku implementácia singletonu môže vyzerať trochu inak. Príklady singlton patternu si môžete pozrieť aj na našom Githube.

2.6.1 Klasická implementácia s lenivou inicializáciou

Je vhodná pre jednoduchú jednovláknovú aplikáciu. Svoju lenivú inicializáciu zabezpečuje funkcia getInstance(), ktorá vytvára inštanciu len v prípade požiadavky od klienta. Pokiaľ je inštancia vytvorená, funkcia vráti už existujúcu inštanciu. Avšak obsahuje v sebe zraniteľnosť pri použiti Reflection, ktorý využíva aj framework Spring.

```
package org.example.SingletonExamples;

public class ClassicLazy{
    3 usages
    private static ClassicLazy instance;
    1 usage
    private ClassicLazy(){}
    public static ClassicLazy getInstance(){
        if (instance == null){
            instance = new ClassicLazy();
        }
        return instance;
    }
}
```

Obr. 17: Singleton s lenivou inicializáciou

2.6.2 Implementácia s lenivou inicializáciou rozšírená pre multi vláknové aplikácie

Implementácia je rozšírená a zabezpečuje správny chod pre multivláknové aplikácie pomocou bloku synchronized. Táto implementácia ale môže spôsobovať nízky výkon aplikácie, pokiaľ sa o ňu dopytuje vysoké množstvo vlákien

```
package org.example.SingletonExamples;

import org.example.DAO.DataSource;

public class MultiThreadLazy {
    3 usages
    private static MultiThreadLazy instance;
    1 usage
    private MultiThreadLazy(){}

    public static MultiThreadLazy getInstance() {
        if (instance != null) {
            return instance;
        }
        synchronized(DataSource.class) {
            if (instance == null) {
                instance = new MultiThreadLazy();
            }
            return instance;
        }
    }
}
```

Obr. 18: Singleton s lenivou inicializáciou rozšírený pre multi vláknové aplikácie

2.6.3 Bill Poughova implementácia

Predstavuje jednoduchšiu implementáciu Singletonu za použitia vnorenej triedy. Keď hlavná trieda je načítaná v pamäti, jej vnorená tireda ešte nie je. Keď klient zavolá funkciu getInstance(), až potom sa vytvorí nová inštancia. Následná stavba Singletonu je obľúbená pre jej jednoduchosť, keďže nevyžaduje ani synchronizáciu.

```
package org.example.SingletonExamples;

public class BillPough {

    private BillPough() {
    }

    private static class SingletonHelper {
        private static final BillPough INSTANCE = new BillPough();
    }

    public static BillPough getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Obr. 19: Bill Poughova implementácia

2.6.4 Implementácia zabezpečená proti Reflection

Predísť Reflection API môžeme aj tým, že pridáme do konštruktora podmienku, ktorá overuje inštanciu, či je null. Ak inštancia nie je null, uskutoční sa výnimka. Iné z riešení je použitie ENUM.

```
package org.example.SingletonExamples;

import java.io.Serializable;

public class PreventReflection {
    private static PreventReflection instance;

    private PreventReflection(){
        if (instance != null) {
            throw new RuntimeException();
        }
    }

    public static PreventReflection getInstance(){
        if (instance == null){
            instance = new PreventReflection();
        }
        return instance;
    }
}
```

Obr. 20: Implementácia zabezpečená proti Reflection

2.6.5 Implementácia zabezpečená proti Serialization a Deserialization

Aby sme zabránili serializácií Singletonu ktorý implementuje rozhranie Serializable, musíme implementovať funkciu readResolve() nasledovne.

```
package org.example.SingletonExamples;

import java.io.Serializable;

public class PreventSerialization implements Serializable {
    private static PreventSerialization instance;

    private PreventSerialization(){}

    public static PreventSerialization getInstance(){
        if (instance == null){
            instance = new PreventSerialization();
        }
        return instance;
    }

    protected Object readResolve() { return getInstance(); }
}
```

Obr. 21: Implementácia zabezpečená proti Serialization a Deserialization

2.6.6 Implementácia zabezpečená proti Cloning

Aby sme zabránili serializácií Singletonu ktorý implementuje rozhranie Cloneable, musíme implementovať funkciu clone() nasledovne.

```
package org.example.SingletonExamples;

import java.io.Serializable;

public class PreventCloning implements Cloneable {
    private static PreventCloning instance;

    private PreventCloning(){}

    public static PreventCloning getInstance(){
        if (instance == null){
            instance = new PreventCloning();
        }
        return instance;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException  {
        throw new CloneNotSupportedException();
    }
}
```

Obr. 22: Implementácia zabezpečená proti Cloning

3 Prototype patter

Vzor prototypu deleguje proces klonovania na skutočné objekty, ktoré sa klonujú. Vzor deklaruje spoločné rozhranie pre všetky objekty, ktoré podporujú klonovanie. Toto rozhranie nám umožňuje klonovať objekt bez spájania kódu s triedou tohto objektu. Takéto rozhranie zvyčajne obsahuje iba jednu metódu klonovania.

Implementácia metódy klonovania je vo všetkých triedach veľmi podobná. Metóda vytvorí objekt aktuálnej triedy a prenesie všetky hodnoty polí starého objektu do nového. Môžete dokonca kopírovať súkromné polia, pretože väčšina programovacích jazykov umožňuje objektom prístup k súkromným poliam iných objektov, ktoré patria do rovnakej triedy.

3.1 Využitie vzoru Prototype

Návrhový vzor prototype využívame hlavne v prípade, keď potrebujeme vytvoriť väčšie množstvo takmer rovnakých objektov jednej triedy. V tom prípade implementujeme metodu clone(), ktorá nám zabezpečí pohodlné vytváranie kópii objektov bez manuálneho prístup mimo triedy. Práve aj preto, keď nastane situácia, že nemáme prístup k atribútom triedy, vieme vytvoriť danú kópiu objektu.

3.2 Návrh

V diagrame vidíme, že klient hovorí prototypu, aby sa naklonoval a vytvoril objekt. Prototyp je trieda a deklaruje spôsob samotného klonovania. MeatSalad, CheeseSalad a VegetableSalad implementujú operáciu na klonovanie.

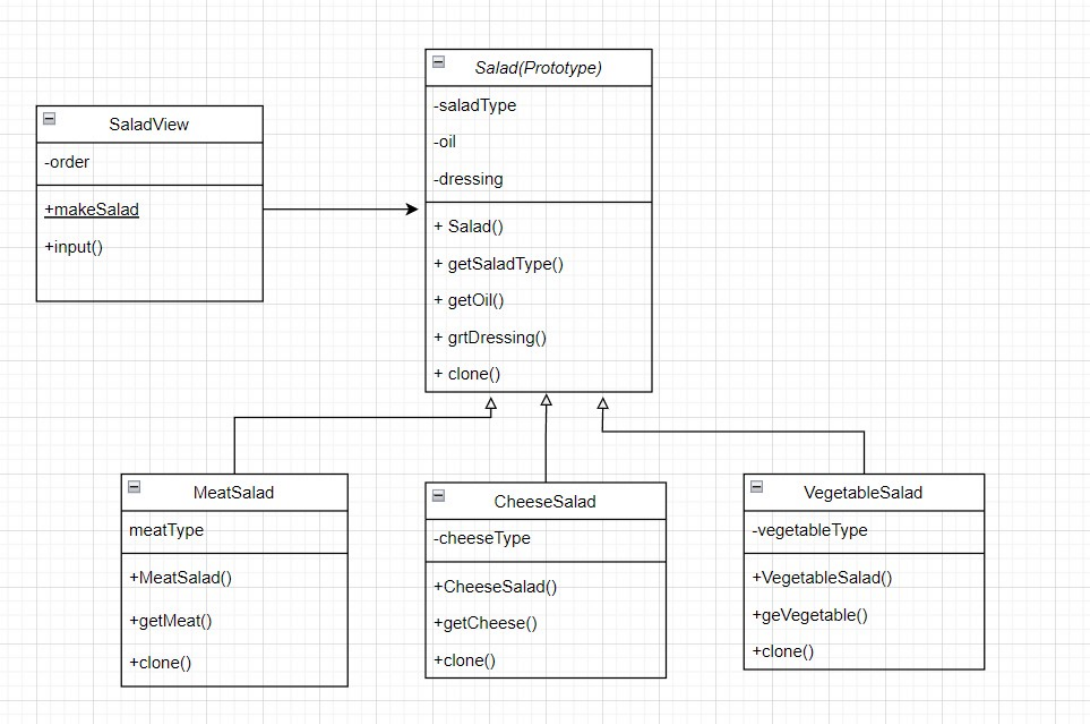
3.3 Implementácia

V implementácii sme zvolii dva prístupy vytvárania objektov:

- Klonovanie - trieda MeatSalad, VegetableSalad
- Vytváranie objektu mimo triedy - trieda CheeseSalad Nie všetky objekty je možné kopírovať druhým spôsobom, pretože niektoré polia objektu môžu byť súkromné a nie sú viditeľné zvonku samotného objektu.

  V metódach pre klonovanie sme vtvorili preťaženú metódu clone(), ktorá následne z vnútra triedy vytvorila kopiu a vrátila.

  Pri vytváraní objektu mimo tried sme pracne pommocou get funkií získali všetky atribúty pôvodnej triedy a na základe získaných informácii pomocou konštruktora vytvorili nový objekt.



Obr. 23: Uml diagram pre prototype.

Pre každý typ šalátu sme si vytvorili jeden objekt, prototyp. Z ktorého sme následne tvorili klony. Tieto klony sme následne uppravovali podľa požiadaviek používateľa a vytvorili objednávku.

```
// object cloning
MeatSalad meatSaladClone = (MeatSalad) meatSalad.clone() ;

System.out.println("  Enter type of meat:");
String meat = reader.readLine();
// decision if we want to have totally same object or slightly different
if (!Objects.equals(meatSaladClone.getMeat(), meat)){
    meatSaladClone.setMeat(meat);
}
// order making
order = " Meat: " + meatSaladClone.getMeat() + " SaladType: " + meatSaladClone.getSalad() +
        " Dressing: " + meatSaladClone.getDresing() + " Oil: " + meatSaladClone.getOil() ;
```

Obr. 24: View pre MeatSalad.

3.4 Výhody a nevýhody

Tento vzor je užitočný, keď sa náš nový objekt len mierne líši od nášho existujúceho. V niektorých prípadoch môžu mať inštancie len niekoľko kombinácií stavu v triede. Takže namiesto vytvárania nových inštancií môžeme vytvoriť inštancie s príslušným stavom

vopred a potom ich klonovať, kedykoľvek budeme chcieť.

Objekty môžeme klonovať bez spojenia s ich konkrétnymi triedami. Môžete sa zbaviť opakovaného inicializačného kódu v prospech klonovania vopred vytvorených prototypov. Zložité objekty môžete vyrábať pohodlnejšie.

Vzor prototypu, rovnako ako každý iný dizajnový vzor, by sa mal používať iba vtedy, keď je to vhodné. Keďže klonujeme objekty, proces sa môže stať zložitým, keď existuje veľa tried, čo vedie k neporiadku.

4 Factory pattern

Návrhovy vzor Factory, slúži na vytváranie rodiny príbuzných objektov, ktoré zdieľajú rovnaké metódy. Tento vzor poskytuje rozhranie na vytváranie objektov v triede, ale umožňuje jej dedeným triedam meniť typ objektov, ktoré budú vytvorené.Dedené triedy implementujú pôvodú Factory metódu na výber triedy, ktorej objekty je potrebné vytvoriť. Uprednostňuje tak vyvolanie metódy namiesto priameho volania konštruktora na vytváranie objektov. Tento prístup zlepšuje čitateľnosť kódu a vytvára priestor pre rozšírenie našej aplikácie bez potreby väčšieho zásahu do pôvodeho kódu.

4.1 Využitie vzoru Factory

Factory metóda oddeľuje konštrukčný kód produktu od kódu, ktorý produkt skutočne používa. Preto je jednoduchšie rozšíriť konštrukčný kód produktu nezávisle od zvyšku kódu. Napríklad, ak chcete do aplikácie pridať nový typ produktu, stačí vytvoriť novú podtriedu tvorcu a prepísať v nej továrenskú metódu. Často sa stretávate s potrebou opätovného použitia existujúcich objektov namiesto toho, aby sme ich zakaždým znova vytvárali. S týmto sa stretávame hlave pri práci s veľkými objektmi náročnými na prostriedky, ako sú pripojenia k databáze, súborové systémy a sieťové prostriedky. Takáto situácia je stvorená pre aplikáciu Factory metódy, ktorá dokáže vytvárať nové objekty a taktiež znovu použiť už existujúce.

4.2 Návrh

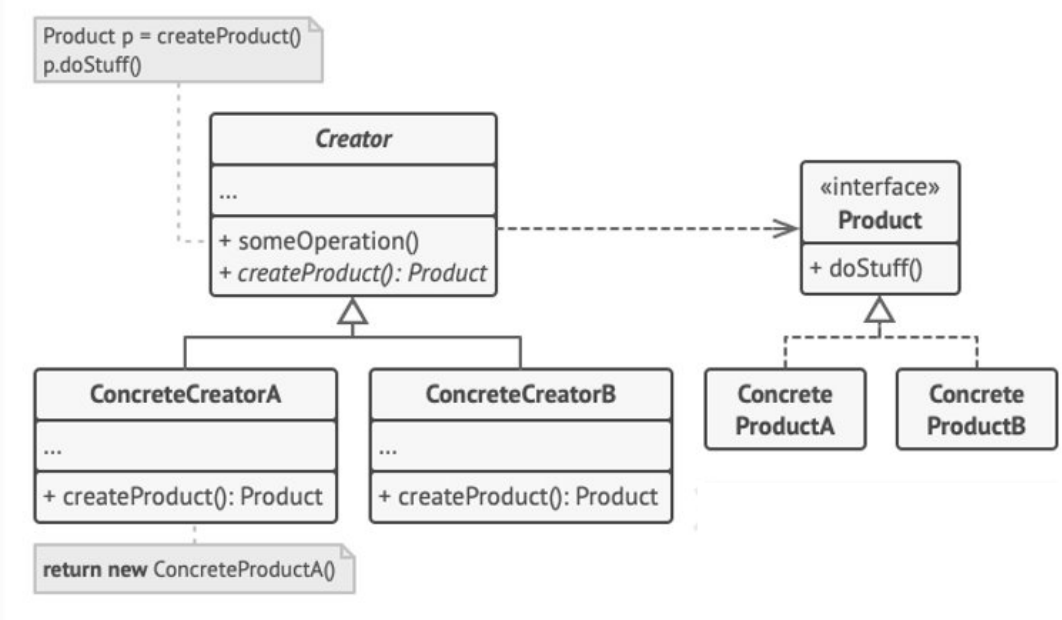
Štruktúru Factory metódy môžeme rozdeliť do rôzných tried, ktoré splňajú rôzné úlohy. Na obrázku nižšie si môžeme všimnúť diagram zobrazujúci základný vzor Factory metódy.

Na obrázku 26 je možné vidieť tieto triedy:

- **Creator**
  - Deklaruje Factory metódu, ktorá vracia nové objekty.
  - Je dôležité, aby sa návratový typ tejto metódy zhodoval s `interface`-om produktu.
- Product
  - Produkt deklaruje interface, ktoré je spoločné pre všetky objekty, ktoré môže vytvoriť Creator a jeho podtriedy.
- Concrete Creator

  - Prepíše základnú Factory metódu, aby vrátila iný typ produktu.

- Concrete Product
  - Sú rôzne implementácie `interface`-u produktu.

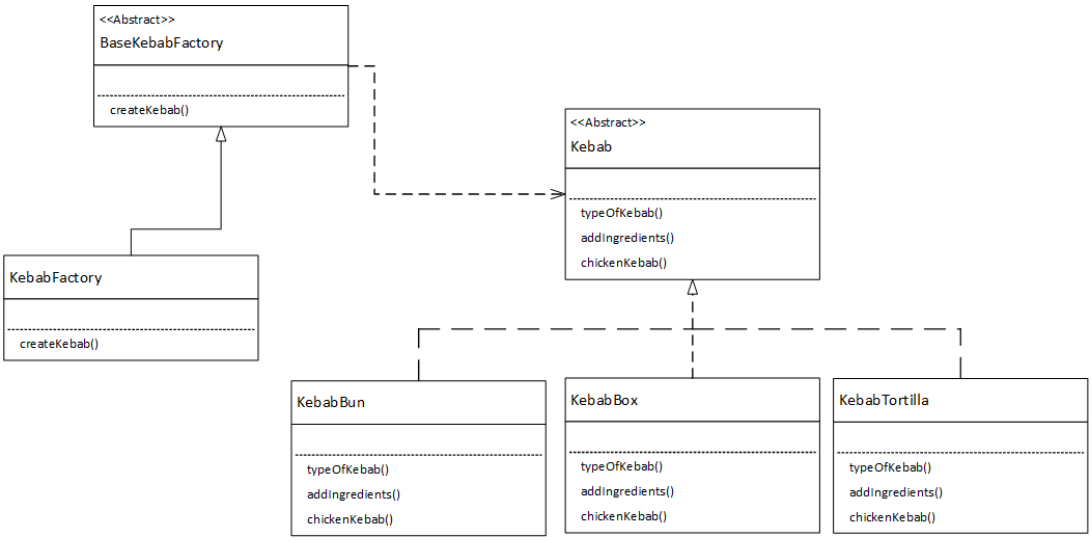


Obr. 25: UML Diagram zobrazujúci všeobecnú Factory.

Tento vzor sme použili aj pri návrhu našej aplikácie, kde sme použili jednotlive triedy na konštrukciu našej vlastnej Factory metódy. V ďalšej podkapitole sa pozrieme na samotnú implementáciu Factory metódy pre potreby našej aplikácie.

4.3 Implementácia

Pre potreby našej aplikácie sme vytvorili podľa vzoru na obrázku 26.



Obr. 26: UML Diagram zobrazujúci Factory metódu našej aplikácie.

Trieda `BaseKebabFactory` je `Creator`. Táto abstraktná trieda deklaruje metódu `createKebab()`, ktorá bude slúžiť na vytvorenie jednotlivých `Concrete Product`-ov. Samotná definícia je v odvodenej triede `KebabFactory`, ktorá už objekty vytvára, a teda vracia samotný objekt `Kebab`, reps. jeho podtriedy. Objekty vytvárame podľa uživateľského vstupu. Tento vstup je číslo zadané do konzoly v rozmedzí 1-3, kde jednotlivé hodnoty reprezentujú `Concrete Product`-y. Abstraktná trieda Produktu `Kebab` obsahujú metódy, ktoré sú rovnáke pre všetky jej podtriedy, tieto metódy reprezentuju vytvorenie pokrmu kebab. Najzaujímavejšia je abstraktná metóda `typeOfKebab()`. Túto metódu si podtriedy definujú sami. `Concrete Product`-y sú reálne použiteľné produkty, ktoré reprezentujú ako bude kebab zabalený. `KebabBun` reprezentuje kebab zabalený v žemli, `KebabBox` reprezentuje kebab v boxe a `KebabTortilla` kebab v placke. Samotným výstupom je reťazec s oznamom, ako je náš kebab servírovaný a teda aký objekt z rodiny `Kebab` uživateľ vytvoril.

4.4 Výhody a nevýhody

1. Vyhneme sa tesnému spojeniu medzi `Creator`-om a `Concrete product`-ami.
2. `Single Responsibility Principle`. Kód na vytvorenie produktu môžeme presunúť na jedno miesto v programe, čím sa zjednoduší podpora kódu.
3. `Open/Closed Principle`. Do programu môžete zaviesť nové typy produktov bez narušenia existujúceho klientskeho kódu.
4. Kód sa môže stať komplikovanejším. Pretože musíme vytvoriť nové podtriedy.

5 Abstract Factory pattern

Abstract Factory je návrhový vzor, ktorý umnožňuje vytvárať rodiny príbuzných objektov bez nutnosti špecifikovať ich konkrétnu triedu. Abstract Factory je rozšírením návrhového vzoru Factory. Základný návrhový typ Factory vracia konkrétne podtriedy jednej rodiny v závislosti od klientskeho vstupu. Pri koncepte Abstract Factory už nepotrebujeme rozhodovanie, ktorý produkt chceme vytvoriť vzhľadom na to, že pre každú podtriedu máme vlastnú Factory.

5.1 Využitie vzoru Abstract Factory

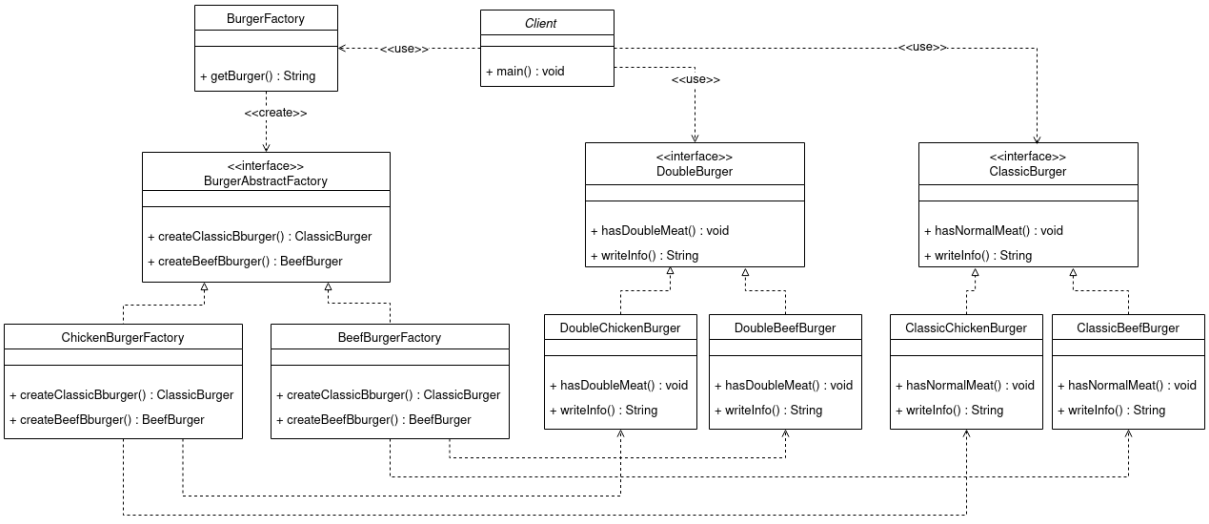
Primárne využitie tohto návrhového vzoru je, keď potrebujeme namodelovať rodinu súvisiacich produktov, pričom každý z členov tejto rodiny môže byť vo viacerých variantách. Zjednodušene teda máme možnosť implementovať rodinu rodín produktov. Príklad využitia môže byť automobilová spoločnosť, ktorá potrebuje vyrábať súčiastky. Jeden Abstract Factory systém môže mať teda vlastné Factory pre všetky súčiastky, ako napríklad ľavé dvere, pravé dvere, predný nárazník a pod. pre všetky modely áut.

5.2 Návrh

Predstavýme si návrh systému, ktorý umožní zákazníkovi objednať si burger podľa možností z ponuky a implementovali sme ho do aplikácie priloženej k dokumnetácií. Dôležitým prvým krokom, ktorý vytvorí lepší prvotný obraz o systéme je matica `platvorma vs produkt` (Obr. č. 27). Pomocou tejto matice následne presne vidíme, aké Factory budeme musieť programovať. Druhým krokom je implementovať všetky abstraktné rozhrania (`interface`), pre typy burgrov - klasický, dvojitý. Tieto rozhrania obsahujú aj všetky metódy, ktoré má výsledný burger používať. Následne implementujeme všetky konkrétne triedy burgrov, kotré implementujú tieto rozhrania - klasický hovädzí burger, dvojitý hovädzí burger a pod. V ďalších krokoch budeme navrhovať samotné Abstract Factory. Navrhneme rozhranie, ktoré implementuje metódy vytvárania jednotlivých burgrov - hovädzí burger, kurací burger. Po tomto kroku navrhneme konkrétne Factory, ktoré implementujú abstraktné rozhranie. Posledným krokom je vytvoriť v aplikácií volanie, ktoré vytvorí inštanciu potrebnej Factory v závislosti od používateľského vstupu, prípadne prostredia, v ktorom sa nachádzame. Približný návrh v podobe UML diagramu môžete vidieť na obrázku č. 28. Konkrétne časti budú lepšie popísané a viditeľné v časti Implementácia.

Burger	Hovädzí	Kurací
Klasický	Klasický hovädzí burger	Klasický kurací burger
Dvojitý	Dvojitý hovädzí burger	Dvojitý kurací burger

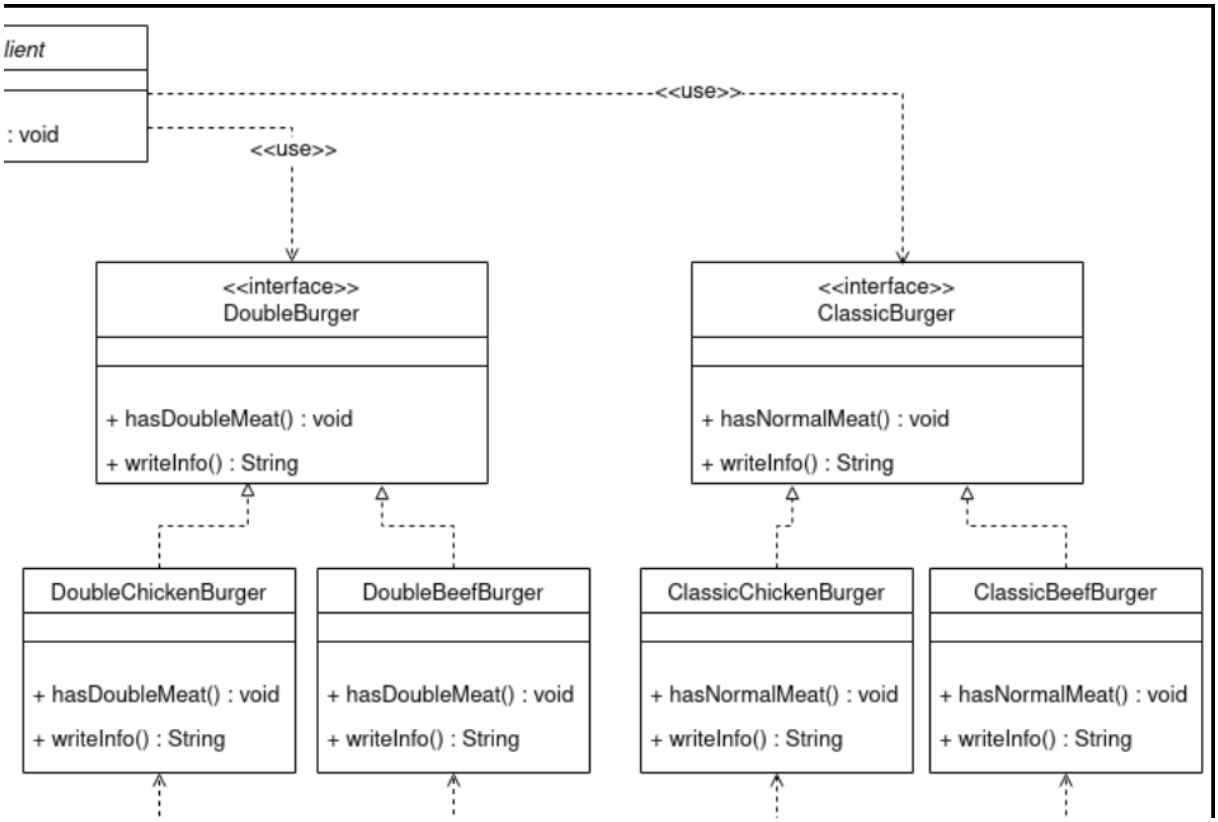
Obr. 27: Matica platforma vs produkt.



Obr. 28: Uml class diagram zobrazujúci Abstract Factory.

5.3 Implementácia

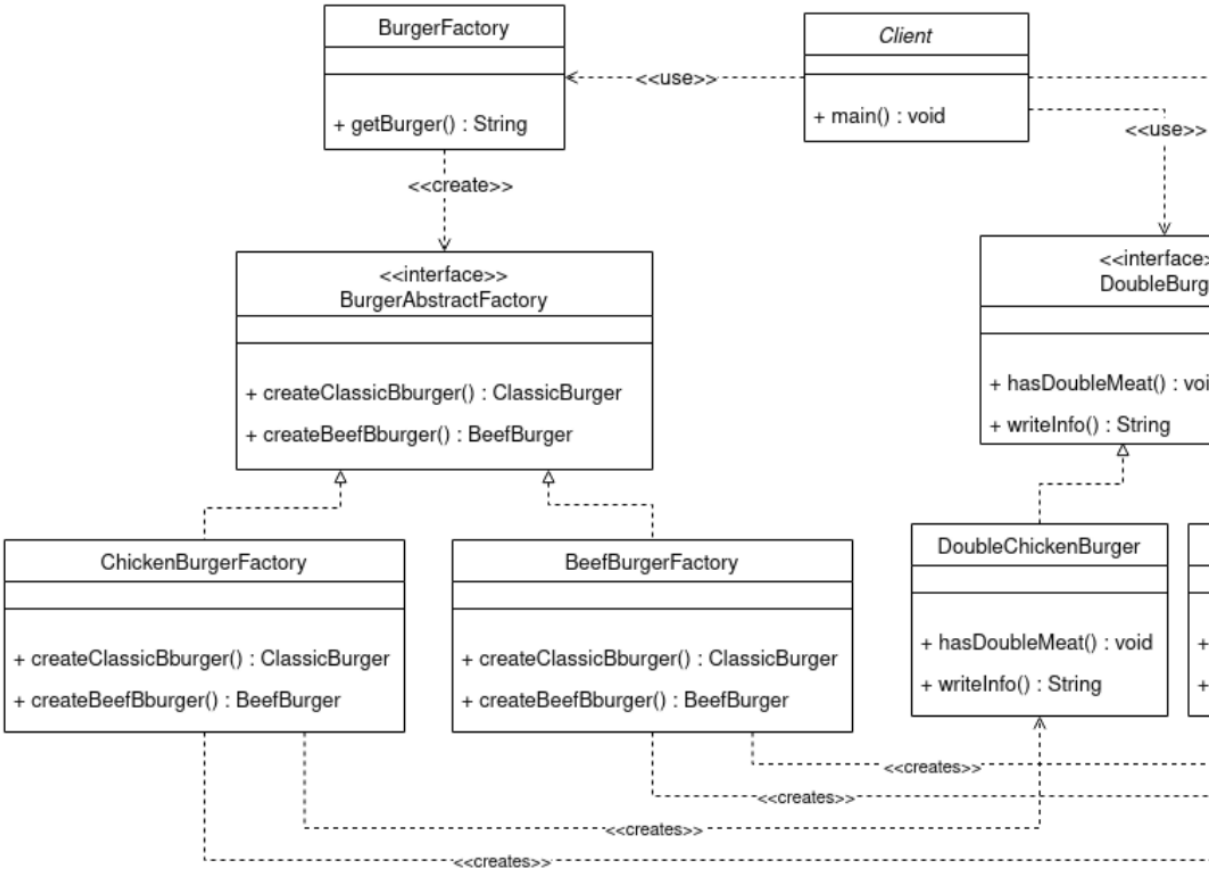
Výsledný podprogram má za úlohu umožniť klientovi výber konkrétneho druhu hamburgera. Ako prvé sme implementovali typy burgrov. Konkrétne sa jednalo o rozhrania používané klientom. Obsahujú metódy špecifické pre daný typ. Následne sme navrhli triedy, ktoré tieto rozhrania implementovali. Túto časť je možné vidieť na výseku Uml diagramu na obrázku č. 29. Ďalším krokom bolo implementovať všetky factory. Začali sme abstraktnou, ktorá predstavovala rozhranie definujúce potrebné metódy pre všetky ostatné. Factrory triedy, ktoré implementovali AbstractFactory obsahujú konštruktory burgrov, ktoré sú súvisiace. To znamená, že pomocou (`ChickenBurgerFactory`), môžem vytvoriť len (`DoubleChickenBurger`) alebo (`ClassicChickenBurger`). Výrez Uml diagramu, ktorý zobrazuje Factory môžete vidieť na obrázku č. 30. Metódy jednotlivých produktov sú v našom programe jednoducho simulované pomocou výpisov.



Obr. 29: Časť Uml diagramu, kde sú triedy implementujúce funkcionalitu.

5.4 Výhody a nevýhody

- Je zaručené, že produkty vytvorené z jednej factory sú kompatibilné.
- Klientsky kód a produkty nie sú úzko prepojené - klient komunikuje len s abstrakt-



Obr. 30: Časť Uml diagramu, kde sú Facotry triedy na vytváranie jedál.

ným rozhraním.

- Je možné jednoducho pridávať nové varianty produktov.
- Vzhľadom na množstvo potrebných tried sa môže zhoršiť čitateľnosť kódu.

Záver

V rámci seminárnej práce sme úspešne naštudovali a pochopili tvorivé návrhové vzory. Spísali sme teoretické princípy a objasnili jednotlivé normy na základe konvencií objektovo orientovaného programovania.

Venovali sme sa teoretickým faktom, štruktúre, výhodám a nevýhodám tvorivých návrhových vzorov.

V návrhovej a implementačnej časti sme v prvom rade predstavili logiku pomocou UML diagramov a predstavili systém na základe ukážok z nášej aplikácie.

Aplikácia je určená pre vytvorenie objednávok jedla. Používatel si môže pomocou používateľského rozhrania vybrať z typov jedla a uzavrieť objednávku, ktorá sa následne vypýše v textovom súbore. Všetky typy jedla uplatňujú tvorivé návrhové vzory.

Prácu sme splnili podľa vopred určených požiadaviek v plnej miere.

Zdroje

- https://refactoring.guru/design-patterns/abstract-factory
- https://www.digitalocean.com/community/tutorials/abstract-factory
- https://www.baeldung.com/java-abstract-factory-pattern
- https://en.wikipedia.org/wiki/Abstract_factory_pattern
- https://refactoring.guru/design-patterns/prototype
- https://refactoring.guru/design-patterns
- https://refactoring.guru/design-patterns/singleton
- https://refactoring.guru/design-patterns/builder
- https://dzone.com/articles/prevent-breaking-a-singleton-class-pattern