SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Antické vojenstvo – Semestrálny projekt

Predmet: Aplikácie smart terminálov
Garant predmetu: doc. Ing. Kvetoslava Kotuliaková, PhD.
Študijný program: multimediálne informačné a komunikačné technológie
Študijný odbor: informatika
Školiace pracovisko: Ústav multimediálnych informačných a komunikačných technológií

2022/2023 **Bc. Viktor Szitkey**

Obsah

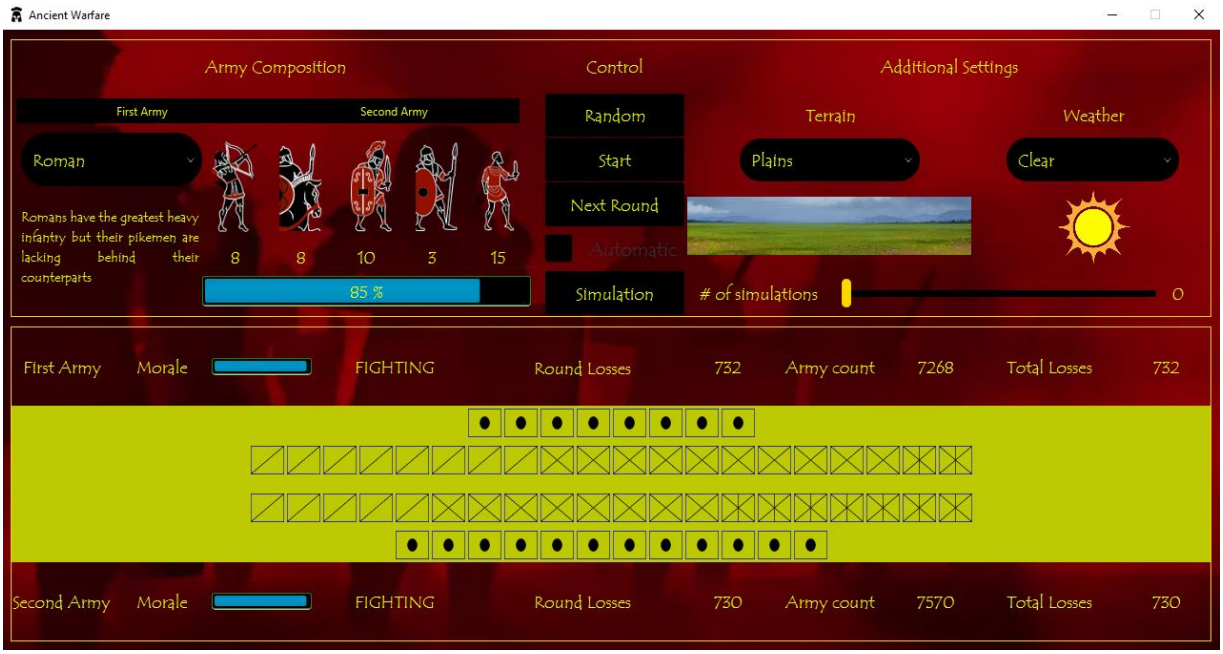
Úvod

Na nasledujúcich stranách je opísaná realizácia projektu, ktorého výsledkom je program vytvorený v programovacom jazyku Java. Hlavnou úlohou programu bolo prezentovať princípy objektovo orientovaného programovania.

Pre tému programu bol zvolený simulátor/hra, ktorá má simulovať boj armád antického sveta.

V prvej kapitole je opísaný program, jeho ovládanie a základné časti. Spomenutá je aj kalibračná funkcia ktorá umožňuje štatisticky vizualizovať výsledky nastavených parametrov ako aj možnosť rýchleho vytvorenia dokumentácie z komentárov cez JavaDoc. Funkčnosť programu je následne vysvetlená na triednom diagrame, ktorého princípy sú taktiež opísané.

Druhá kapitola opisuje OOP (Object Oriented Programming) princípy použité pri tvorbe programu, so zameraním sa na polymorfizmus. Pre zachovanie rozsahu a stručnosti sú vynechané opisy niektorých princípov (program ich však obsahuje/využíva) ako konvencia mien, objekt, trieda, metóda, konštruktor, a pod. V prípade potreby pochopenia nespomenutých princípov alebo snahy o pochopenie vymenovaných je vhodné sa odkázať napr. na [1].



Obrázok 1-1 Hlavná obrazovka GUI programu

1 Program

Program bol napísaný v programovacom jazyku Java. Editovaný a spúšťaný (otestovaný) v IDE Eclipse. GUI (Graphical User Interface) programu bol napísaný použitím modulov knižnice JavaFX19. GUI bol pre finálnu podobu upravený pomocou .css formátovania. Na usporiadanie jednotlivých elementov GUI, bol použitý program Scene Builder. Postup ako spustiť tento program v rovnakom prostredí za účelom ďalšieho testovania alebo úpravy je uvedený na [2] pre nainštalovanie Eclipse IDE s podporou JavaFX a na [3] pre nainštalovanie a nastavenie programu Scene Builder.

1.1 Základná myšlienka

Program slúži ako jednoduchý simulátor/hra pre simuláciu boja vojsk antických armád. Vzhľadom na rozsah projektu boli zo všetkých možných faktorov ovplyvňujúci takúto simuláciu vybrané len niektoré faktory ako napríklad: zloženie vojska, terénne a atmosférické okolnosti.

1.2 Funkcie

Vo verzii 2.0 simulátor pracuje s nasledovnými možnosťami:

1. Simulácie jednej bitky
    a. Výber druhu armády a jej kompozície (rôzne druhy jednotiek od kavalérie, cez pechotu po lukostrelcov)
    b. Nastavenie okolností bitky:
        i. Počasie
        ii. Terén
    c. Spustenie simulácie
    d. Vyhodnocovanie boja počas aj po bitke
2. Simulácie viacerých bitiek
    a. Nastavenie počtu simulácii (vo v.2.0 max. 100)
    b. Zápis výsledkov do súboru (result.txt)

1.3 Výpis do súboru

Pre prípadne odladťovanie jednotlivých hodnôt simulácie, napríklad zabezpečenie aby jedna frakcia nebola príliš nevyvážená, je možné spustiť tzv. mód simulácie, ktorý v pozadí odsimuluje určitý počet simulácii (bojov) a následne výsledok zapíše do súboru *result.txt*. Je možné do tohto súboru zapisovať výsledky viacerých simulácii a sledovať celkovú štatistiku.

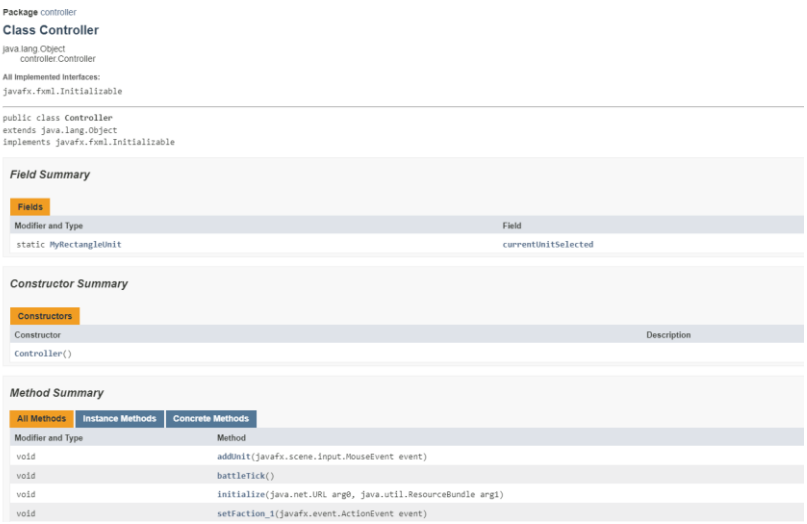
```
Number of victories per simulation and per faction:
Roman faction: 27,16
Carthaginian faction: 22,76
Greek faction: 25,41
Celtic faction: 24,67

-------------------------------------------------------------------------------------------------------------
1. simulation
#       Roman faction: |02|     Carthaginian faction: |02|      Greek faction: |01|     Celtic faction: |04|   Civil wars: |00| #
-------------------------------------------------------------------------------------------------------------
2. simulation
#       Roman faction: |08|     Carthaginian faction: |12|      Greek faction: |18|     Celtic faction: |11|   Civil wars: |00| #
-------------------------------------------------------------------------------------------------------------
3. simulation
#       Roman faction: |07|     Carthaginian faction: |06|      Greek faction: |06|     Celtic faction: |09|   Civil wars: |00| #
-------------------------------------------------------------------------------------------------------------
4. simulation
#       Roman faction: |00|     Carthaginian faction: |01|      Greek faction: |02|     Celtic faction: |00|   Civil wars: |00| #
-------------------------------------------------------------------------------------------------------------
```

Obrázok 1-1 Výstup simulácie v súbore

1.4 JavaDoc

Jednotlivé časti programu boli okomentované formulkou `/** obsah komentára */`, to je nevyhnutné pre použitie automaticky generovanej dokumentácie tzv. JavaDoc poskytovanej IDE Eclipse (mimo iných). Ide o API dokumentáciu generovanú z komentárov. Na vytvorenie tejto dokumentácie bol použitý postup z [1]. Výsledná dokumentácie sa následne dá prehľadne zobraziť v prehliadači.


Package controller

Class Controller

java.lang.Object
controller.Controller

All Implemented Interfaces:
javafx.fxml.Initializable

public class Controller
extends java.lang.Object
implements javafx.fxml.Initializable

Field Summary

Fields

Modifier and Type	Field
static MyRectangleUnit	currentUnitSelected

Constructor Summary

Constructors

Constructor	Description
Controller()	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method
void	addUnit(javafx.scene.input.MouseEvent event)
void	battleTick()
void	initialize(java.net.URL arg0, java.util.ResourceBundle arg1)
void	setFaction_1(javafx.event.ActionEvent event)

Obrázok 1-2 JavaDoc popis triedy Controller.java

1.5 MVC

Pri tvorbe programu bol použitý tzv. MVC (Model View Controller) vzor. Ide o prehľadný spôsob delenia kódu a jeho jednotlivých častí. Jednotlivé časti programu ako triedy, rozhrania, .fxml súbory, obrázky atď. sú zadelené do jednej z troch kategórii a to buď:

- Model – obsahuje dáta a logiku pre aktualizovanie Controller-a.
- Controller – slúži ako úroveň medzi Model a View, ktoré by mali byť striktne oddelené. Je zodpovedný za riadenie programu.
- View – prezentačná vrstva programu zodpovedná za aktualizáciu viditeľných informácii napr. v GUI.

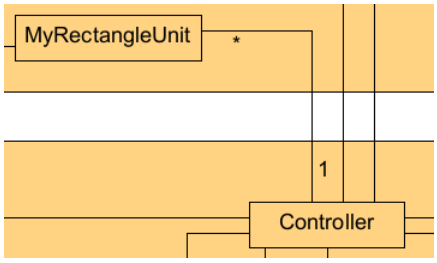
1.6 UML (Unified Modelling Language) programu

Pre tvorbu programu a vizualizáciu jeho funkčnosti bol zostavený UML diagram zostavený pomocou Eclipse pluginu Umlet. UML diagram slúži pre vizualizáciu programu, ideálne ešte pred jeho napísaním. Ide však len o jeden zo spôsobov vizualizácie programu, tzv. Class Diagram. Taktiež je možné použiť: Component Diagram, Object Diagram, Use Case Diagram, atď. Popis k týmto ako aj ďalším možnostiam je dostupný na [4] .

1.6.1 Class diagram

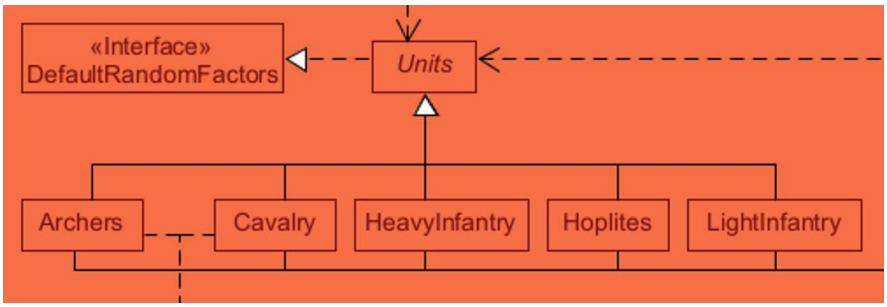
Pre pochopenie akéhokoľvek UML Class diagramu je nutné poznať používanú symboliku a nomenklatúru. Nasleduje popis základným vzťahovým symbolom:

- **Asociácia (Association)** – medzi triedami znamená, že triedy spolu interagujú. Je vhodné špecifikovať početnosť (multiplicity), to znamená koľko objektov interaguje vo vzťahu medzi triedami. Na obrázku je uvedený príklad v ktorom trieda *Controller.java* interaguje s triedou *MyRectangleUnit.java*. Početnosť tohto vzťahu je opísateľná ako veľa inštancii *MyRectangleUnit* interaguje iba s jedným *Controller* a opačne.



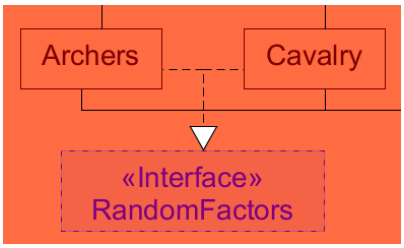
Obrázok 1-3 Príklad asociácie

- **Dedenie (Inheritence)** – trieda alebo skupina tried dedí ak lepšie špecifikuje všeobecnejšiu triedu a dedí znej spoločné vlastnosti. Príklad: každý tip jednotky má iné vlastnosti, jazdec sa v mnohom líši od lukostrelca. V niečom sú však rovnaký/podobný, to bude definované v spoločnej triede.



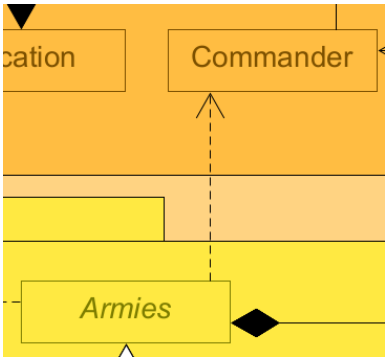
Obrázok 1-4 Príklad dedenia

- **Realizácia (Realisation)** – vyjadruje vzťah medzi objektom rozhrania (Interface) a objektami, ktoré toto rozhranie implementujú. Je rozhranie je abstraktná trieda ktorej metódy musia byť prekonané triedami ktoré ho implementujú.



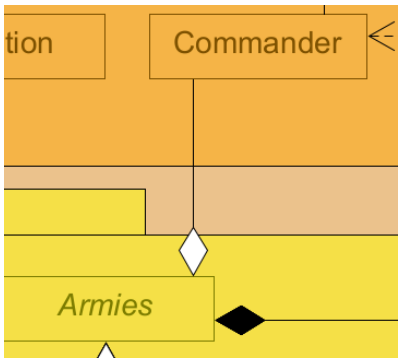
Obrázok 1-5 Príklad realizácie

- **Závislosť (Dependency)** – je priamy vzťah poukazujúci na závislosť elementu na inom elemente. Ako príklad je možné uviesť inštanciu Armies.java ktorej vlastnosti závisia od použitej inštancie *Commader.java*.



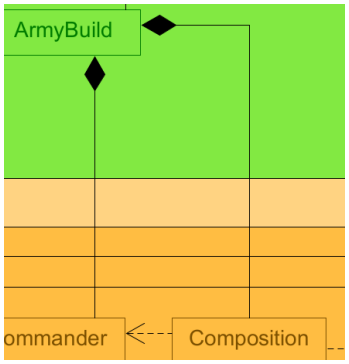
Obrázok 1-6 Príklad závislosti

- **Agregácia (Aggregation)** – v prípade že je inštancia triedy komponent inej triedy ide o agregáciu. V prípade že sa trieda obsahujúcu inú triedu vymaže, jej agregované komponenty sú stále použiteľné. Príklad: Veliteľ armády nie je nevyhnutne vymazaný ak vymažeme armádu, môže byť prevelený k inej armáde ak bola ta stará rozpustená.



Obrázok 1-7 Príklad Agregácie

- **Kompozícia (Composition)** – podobná agregácii, avšak objekt v kompozičnom vzťahu sa taktiež vymaže, v prípade vymazania inštancie objektu ktorý ho obsahuje. Príklad: kompozícia armády existuje len dokiaľ existuje inštancia *ArmyBuild*, ktorá ju obsahuje.

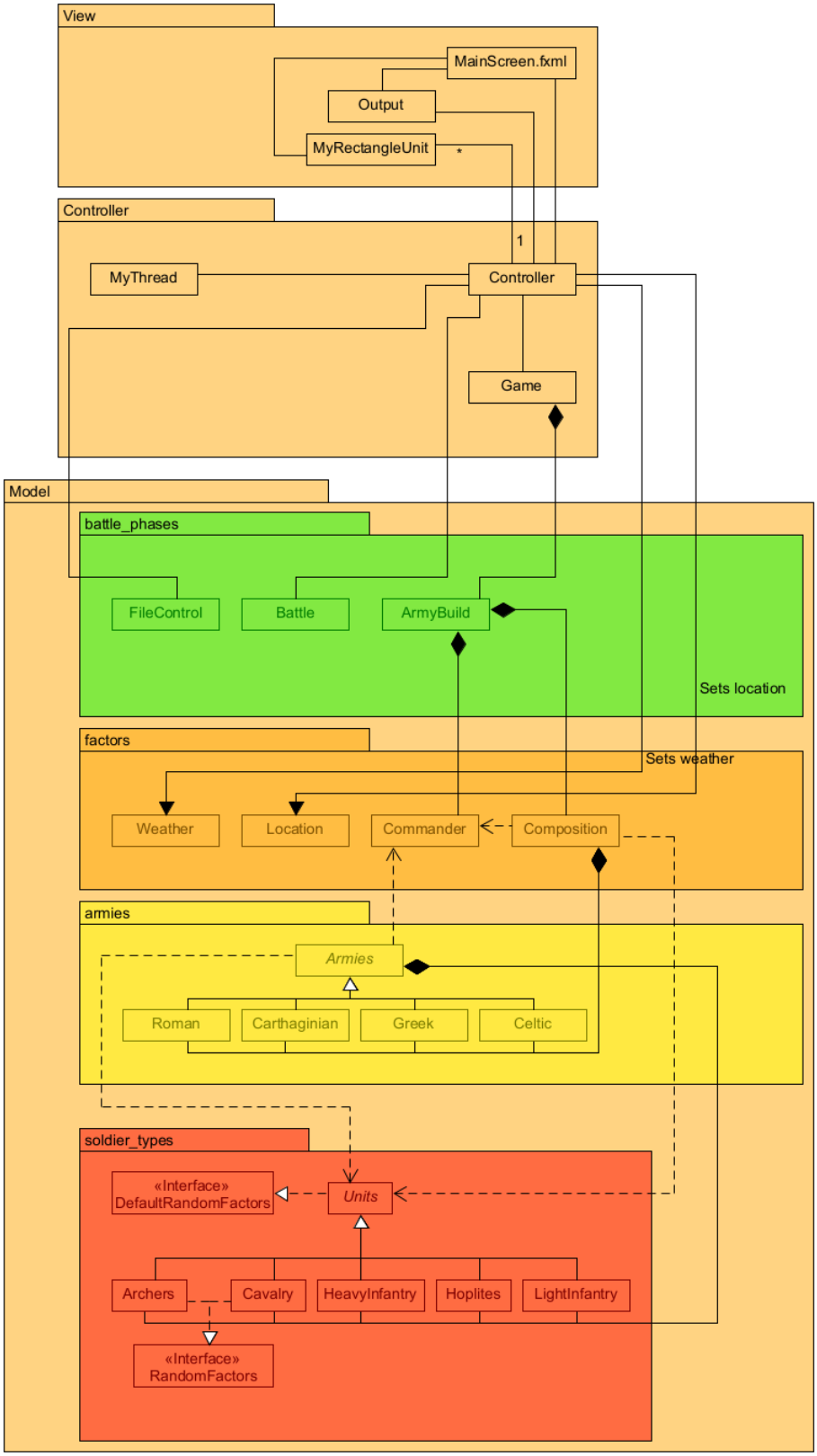


Obrázok 1-8 Príklad kompozície

1.6.2 Popis programu pomocou UML Class diagramu

Rýchle zhrnutie činnosti programu je teda možno vidieť na diagrame, a to nasledovne:

1. *Game.java* vytvorí inštancie dvoch *ArmyBuild.java*
2. Taktiež vytvorí JavaFX Stage pre vytvorenie GUI
3. GUI elementy sú špecifikované v *MainScreen.fxml* a ovládane z *Controller.java*
4. Inštancie armád sú ďalej nastavované v *Controller.java* a to nastavením z údajov získaných z GUI.
5. Controller však len volá funkcie z tried v Model-y.
6. Pre inštanciu sa počas behu programu nastaví:
    a. Frakcia (*Faction.java*)
    b. Veliteľ – neimplementované v v.2.0 (*Commander.java*)
    c. Kompozícia (*Composition.java*), v ktorej sa vytvorí inštancia armády (*Army.java*) a do ktorej sa uložia jednotlivé časti armády (*Units.java*)



Obrázok 1-9 UML programu

2 Použité princípy objektovo orientovaného programovania

Pri tvorbe programu bolo využitých niekoľko v Jave často používaných princípov. Nasleduje ich krátky opis ako aj použitie na príklade z programu. Agregácia, kompozícia a rozhranie boli spomenuté a vysvetlené pri UML diagrame.

2.1 Prístupnosť (Enkapsulácia)

Jednotlivé metódy a atribúty tried ako aj triedy samotné, sú podľa potreby nastavené s jedným z príznakov, ktoré určujú ich dostupnosť v inej časti programu. Je tak zabezpečené, že sa dané funkcie spúšťajú a premenné menia, len tam kde je to povolené:

Visibility	Default	Public	Protected	Private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Subclass outside the same package	No	Yes	Yes	No
Non-subclass outside the same package	No	Yes	No	No
iq.opengenus.org				

Obrázok 2-1 Zhrnutie rozdielov prístupnosti

2.2 Dedenie

Dedenie je mechanizmus pri ktorom objekt získa všetky vlastnosti rodičovského objektu. Je tak možné vytvárať nové triedy, ktoré sú postavené na už existujúcich triedach a využívať ich metódy a polia.

V programe je použité dedenie napríklad pri definovaní jednotlivých jednotiek ako lukostrelec, jazdec, pešiak. Hodnoty ich atribútov sa môžu líšiť avšak atribúty samotné sú rovnaké. Preto je možné napísať triedu ktorá ich opisuje len raz (*Units,java*), a následne túto triedu dediť (*Archer.java, Cavalry.java,...*).

2.3 Abstraktná trieda

Je trieda deklarovaná s kľúčovým slovom **abstract**. Môže mať abstraktné (nutné prekonať v triede ktorá dedí abstraktnú triedu) a neabstraktné metódy. Nemôže byť iniciovaná, takže aby bola užitočná je nutné ju v dediť v inej triede.

V programe bola použitá v už v spomenutom príklade z jednotkami. Trieda *Units.java* je abstraktná, nikdy nebola vytvorená pretože nejde o armádnu zložku ale len o spoločnú šablónu z ktorej sa dedí.

2.4 Java polymorfizmus

Polymorfizmus znamená „viac tvarovosť“ resp. vykonanie jednej akcie viacerými spôsobmi. V kontexte Java jazyku bol polymorfizmus použitý v nasledujúcich technikách.

2.4.1 Prekonanie metódy

V prípade ak podtrieda obsahuje metódu, ktorú už obsahuje trieda z ktorej dedí, ide o prekonanie metódy.

V programe bola v *Units.java* definovaná metóda určujúca vlastnosti pechoty v určitom terénne. Tie sa nutne líšia v prípade kavalérie. Preto bola v triede *Cavalry.java* vytvorená metóda s rovnakým menom avšak poskytujúca iné hodnoty.

2.4.2 Preťaženie vlastnej metódy/konštruktéra

Prípade ak má trieda dve a viac metód s rovnakým menom ale s inými parametrami, ide o preťaženie metódy.

Programe boli preťažené najmä metódy konštruktéra tried.

```
public Commander(String name, Integer skill, String specialization, String nickname) {
    this.name = name;
    this.skill = skill;
    this.specialization = specialization;
    this.nickname = nickname;
}

public Commander(Integer skill) {
    this.skill = skill;
}
```

Obrázok 2-2 Preťažená metóda konštruktéra

2.4.3 Kľúčové slovo Super

Kľúčové slovo **super** je referenčná premenná odkazujúca sa na objekt rodičovskej triedy. Týmto objektom môže byť atribút, metóda alebo konštruktér, ktoré sa cez **super** volajú z podtriedy.

V programe bol tak pri tvorbe špecifickej jednotky (lukostrelec, jazdec, ...) volaný konštruktér rodičovskej triedy *Units.java*. Nastavili sa tak vlastnosti tejto jednotky.

```
public Archers(int aD, ArmiesEnum allegienceToArmy) {
    super(aD, 2, 200, allegienceToArmy);
}
```

Obrázok 2-3 Použitie kľúčového slova super

2.4.4 Kľúčové slovo Final

Final umožňuje obmedzenie viacnásobného definovania objektu. V prípade premennej, nie je možné hodnotu tejto metódy meniť, a priradenie počas runtime-mu môže prebehnúť len v konštruktéri. Final metódu nie je možné prekonať a final triedu nie je možné zdediť.

V programe boli takto vytvárané a proti zmene kontrolované konštanty.

2.4.5 Kľúčové slovo instanceof

Toto kľúčové slovo sa používa na otestovanie či je daný objekt inštanciou špecifickej triedy.

V programe tak bolo možné rozlíšiť o aký typ jednotky pri danom objekte v poli jednotiek išlo.

```
if(unit instanceof Archers) {
    return AA;
}
else if(unit instanceof Cavalry) {
    return CA;
}
else if(unit instanceof HeavyInfantry) {
    return HA;
}
else if(unit instanceof Pikemen) {
    return PA;
}
else if(unit instanceof LightInfantry) {
    return LIA;
```

Obrázok 2-4 Použitie kľúčového slova instanceof

2.4.6 Runtime polymorfizmus

Prekonania a preťaženie metódy sú príklady compile-time polymorfizmu. Ak sa však rozhoduje ktorá metóda sa má použiť počas behu programu ide o runtime polymorfizmus.

Pre pochopenie je najskôr vhodné vysvetliť pojem **Upcasting**. V prípade ak referencia rodičovskej premennej odkazuje na objekt podtriedy, ide o Upcasting. Ako príklad z programu je možné spomenúť pole jednotiek typu *Units.java*, ktoré obsahuje rôzne inštancie podtried jednotiek.

```
units[0] = new Archers[archers];
units[1] = new Cavalry[cavalry];
units[2] = new HeavyInfantry[heavy];
units[3] = new Pikemen[pikemen];
units[4] = new LightInfantry[light];
```

Obrázok 2-5 Použitie princípu Upcasting

Prípade že bolo napr. potrebné zistiť špecifickú hodnotu jednotky, stačilo použiť upcast inštanciu unit ktorej skutočná inštancia bola automaticky zistená, bez nutnosti špecifikovania podtriedy.

```
firstArmy.getFirstLine().get(position).getUnit().unitDamage(firstArmy.getRoleOfArmy(), firstArmy.getComp().getArmy());
```

Obrázok 2-6 Volanie metódy pri využití polymorfizmu

Predchádzajúce volanie využíva metódu na nasledujúcom výpise. V tejto metóde bolo namiesto *unit* použité kľúčové slovo **this**. Toto slovo odkazuje na premennú v danom objekte.

```
private Double unitDamage(boolean roleOfArmy, Armies army){
    double damage = 0;
    try {
        damage = (army.getDamage(this, roleOfArmy) * this.weatherValue(Weather.getWeather())
                    *this.getMorale()*this.locationValue(Location.getLocation(), roleOfArmy)
                    +this.getNumber());
    } catch (Exception e) {
        return 0.0;
    }
    return damage;
}
```

Obrázok 2-7 Použitie kľúčového slova

Záver

V tejto práci bolo odprezentované riešenie programu ktorý poslúžil na prezentovanie niektorých princípov objektovo orientovaného programovania.

Učinil tak prezentovaním boja armád antického sveta a to vo forme hry, kde je možne nastaviť dve rôzne armády a nechať ich vybojovať simulovanú bitku.

Taktiež je možné nastaviť väčšie množstvo simulácii a uložiť ich výsledok do súboru v prehľadnom formáte, ktorý poslúži ako kalibračný nastroj jednotlivých atribútov programu.

Pre rýchle pochopenie jednotlivých častí programu bola vygenerovaná dokumentácia pomocou JavaDoc nástroja.

V druhej kapitole boli popísané niektoré použite princípy objektovo orientovaného programovania.

Referencie

[1] N. Minh, „CodeJava,“ [Online]. Dostupné na: https://www.codejava.net/ides/eclipse/how-to-generate-javadoc-in-eclipse.

[2] B. Code, „Youtube,“ [Online]. Dostupné na: https://www.youtube.com/watch?v=_7OM-cMYWbQ.

[3] B. Code, „Youtube,“ [Online]. Available: https://www.youtube.com/watch?v=-Obxf6NjnbQ.

[4] freeCodeCamp.org, „Youtube,“ [Online]. Dostupné na: https://www.youtube.com/watch?v=WnMQ8HlmeXc.