SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 1663

# Implementacija modela strateške igre

Marko Stanić

Zagreb, lipanj 2018.

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 15. ožujka 2018.

# DIPLOMSKI ZADATAK br. 1663

Pristupnik: **Marko Stanić (0036473130)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Implementacija modela strateške igre**

Opis zadatka:

Opisati problematiku modeliranja strateških igara za dva igrača s obzirom na uvjete igranja i izvedbu poteza. Istražiti mogućnosti uporabe strojnog učenja u modeliranju računalnih igrača strateških igara i navesti postojeća rješenja. Ostvariti modularni programski sustav za simulaciju i prikaz tijeka i ishoda strateške igre za više igrača. U sustav dodati mogućnost prikaza te proizvoljnog modela računalnog igrača. Ispitati učinkovitost postupaka strojnog učenja za oblikovanje računalnog igrača na primjeru strateške igre. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:
Prof. dr. sc. Domagoj Jakobović

Djelovođa:
Doc. dr. sc. Boris Milašinović

Predsjednik odbora za diplomski rad profila:
Izv. prof. dr. sc. Igor Mekterović

# SADRŽAJ

# 1. Uvod

Razvojem područja evolucijskih algoritama javlja se potreba za sve složenijim i dobro definiranim funkcijama dobrote. Iako se za testiranje i pokazivanje efikasnosti takvih algoritama najčešće koriste matematičke funkcije, nekad je korisno pokazati kako se određeni algoritam snalazi u optimizaciji funkcije s vrlo velikim brojem dimenzija, kao što je računalna igra.

Međutim, današnje računalne igre najčešće nisu razvijene s namjerom da ih igra stroj, već čovjek. Zbog toga takve igre imaju bogata korisnička sučelja čija funkcionalnost je često ispreplitana s funkcionalnošću same logike igre, pa je adaptacija takvih igara u svrhe strojnog učenja ili evolucijskih algoritama vrlo teško izvediva. Nadalje, bogato korisničko sučelje je često uzrok sporosti igre, zbog čega je vrlo teško obraditi više instanci igre u kratkom vremenu. Iako navedena tehnička ograničenja znatno otežavaju primjenu umjetne inteligencije u razvoju igara, to nije potpuno nemoguće. Primjerice, za igru *Defense of the Ancients 2* (slika 1.1) je razvijen bot koji može pobijediti vještog ljudskog protivnika. [2]

**Slika 1.1:** Prikaz igre *Defense of the Ancients 2*

Iako je takav bot sposoban pobijediti igru u strogo ograničenim uvjetima (ograničen skup pravila te igranje protiv jednog protivnika), smatra se značajnim postignućem.

Ovaj rad opisuje izradu programskog modela strateške igre koji može poslužiti za slične svrhe.

# 2. Strateška igra

## 2.1. Definicija strateške igre

Strateška igra je vrsta stolne ili računalne igre u kojoj ishod ovisi o igračevoj sposobnosti analize situacije i donošenja odluka.

Jedna od najpoznatijih stolnih strateških igara je šah. Radi se o strateškoj igri koja je potekla iz Indije, a prvi puta se spominje oko 600. godine. Dvoje igrača igraju na poteze, pri čemu svaki potez mogu pomaknuti jednu figuru, ali svaka vrsta figure ima određena pravila po kojima se smije kretati po ploči ili napadati. Da bi igrač mogao dobro pratiti protivnika, mora biti sposoban razmišljati nekoliko poteza unaprijed. Dakle, za donošenje odluke o sljedećem potezu nije dovoljno razmotriti trenutno stanje igre, već moguća stanja u kojima bi se igra nalazila nakon određenih poteza. [1]

## 2.2. Problematika razvoja umjetne inteligencije za strateške igre

Ako se na strateški način razmišljanja gleda iz perspektive programera, jasno je vidljivo da se radi o stablu odluka (engl. *decision tree*). Međutim, nakon nekoliko razina stabla, računanje potencijalnih ishoda za računala postaje vrlo dugotrajno, pa je za složenije strateške igre potrebno upotrijebiti alternativne metode za donošenje odluka u igri.

Primjerice, moguće je upotrijebiti neuronsku mrežu kod koje ulazi opisuju trenutno stanje igre, a izlaz određuje naredbu koju će virtualni igrač izvesti. [3] Nadalje, moguće je evoluirati stablo odluka korištenjem tehnika poput genetskog programiranja.

# 3. Model strateške igre

## 3.1. Prva verzija

Početkom razvoja, model je bio zamišljen kao pojednostavljena verzija igre *Warcraft: Orcs and Humans*. Dakle, implementiran je prototip dvodimenzionalne igre koja se sastojala od samostalnog modela i grafičkog sučelja (slika 3.1). Svaki igrač je kontrolirao vojsku likova koji su morali poraziti vojsku njegovog protivnika, a likovi su puteve po okruženju nalazili pomoću algoritma za pronalaženje najkraćeg puta A*.

**Slika 3.1:** Prikaz korisničkog sučelja prve verzije modela

Međutim, zbog kontinuirane prirode i vrlo visoke složenosti igre, ustanovljeno je da bi za potpuno ostvarenje takve igre trebalo više vremena nego je za to predviđeno u sklopu ovog rada. Nadalje, zbog složenosti prototipa nije bilo realistično očekivati da će metode strojnog učenja nad modelom proizvesti smisleno rješenje na današnjem prosječnom računalu.

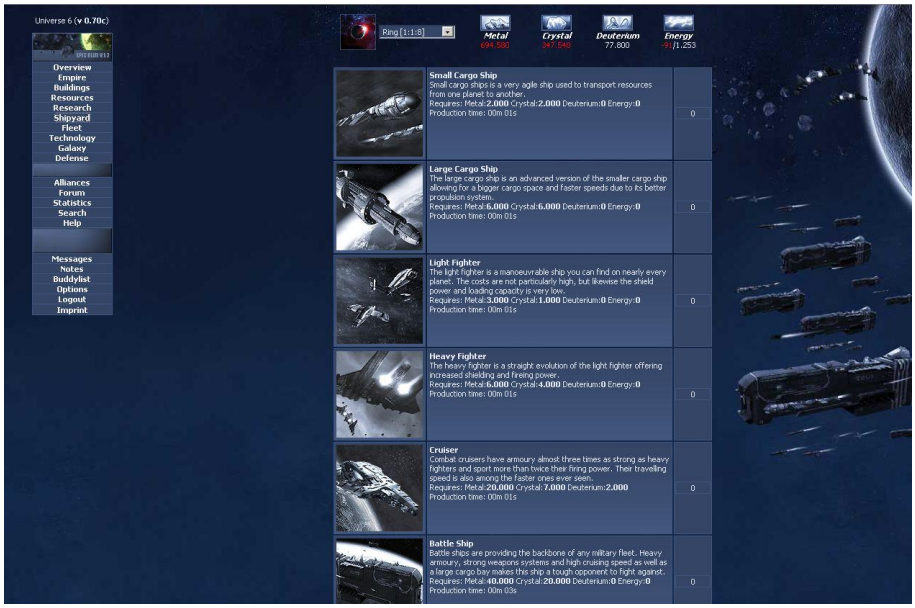
Zbog toga je odlučeno da se implementira jednostavniji i laganiji model.

## 3.2. Druga verzija

Druga verzija igre zamišljena je kao modularna jednodimenzionalna igra. Potezi su diskretizirani, koncept dvodimenzionalnog okruženja i nalaženja puteva je izbačen, a ishod borbe se računa trenutno, bez utjecaja igrača.

### 3.2.1. Pregled modela

Model je konstruiran po uzoru na nekad popularnu mrežnu igru *OGame* (slika 3.2). Sudjeluju dva igrača, od kojih svaki kontrolira jedno carstvo. Carstvo se sastoji od zgrada, flote i tehnologije.




**Slika 3.2:** Korisničko sučelje igre *OGame*

#### Zgrade

Zgrada je jedinica koja se nalazi u carstvu. Unutar jednog carstva može postojati po jedna zgrada svakog tipa. Svaka zgrada ima mjeru razvijenosti (engl. *level*). Funkcija zgrade ovisi o njezinoj definiciji u pravilima igre.

#### Flota

Flota se sastoji od brodova, koji su borbene jedinice. Svaki brod ima određene vrijednosti napada, oklopa i štitova koje su definirane u pravilima. Brodovi sadrže i tzv. *rapidfire* svojstvo - listu brodova protiv kojih imaju pravo višestruko napadati unutar

jednog borbenog poteza. Svaki brod ima određeni kapacitet spremnika koji predstavlja količinu resursa koje može otuđiti neprijatelju ako pobijedi.

#### Tehnologija

Tehnologija je mjera razvijenosti različitih aspekata carstva. Utjecaji ovise o definiranim pravilima igre.

### 3.2.2. Pregled tijeka igre

Igra nema definiran kraj - određivanje kriterija završetka je prepušteno klijentskoj aplikaciji.

Svaki potez, igrač bira jednu od mogućih radnji:

- izgradnja određene zgrade
- istraživanje određene tehnologije
- napad na neprijatelja
- preskakanje poteza

Odabirom bilo koje od radnji na potez dolazi idući igrač. Izgradnja zgrade ili istraživanje tehnologije ne moraju biti trenutni - jednom kada su započeti, mogu trajati i nekoliko poteza, ovisno o pravilima. Konkretan model borbe je također definiran u pravilima.

# 4. Programska implementacija

Model je ostvaren u programskom jeziku C++, pri čemu su korištene i mogućnosti iz novijih verzija C++ standarda (C++14, C++17). Kao sustav za izgradnju projekta je korišten CMake, koji omogućuje stvaranje uputa za kompilator na vrlo širokom rasponu operacijskih sustava.

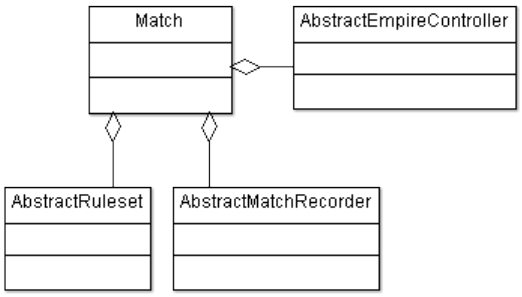
Sustav je dizajniran u skladu s RAII principom - svako rezerviranje resursa automatski inicijalizira taj resurs. Drugim riječima, sve što neki objekt treba imati za korištenje mora biti inicijalizirano ili dostavljeno u konstruktoru, bez potrebe za naknadnim pozivanjem neke metode nakon njegovog instanciranja.

Da se izbjegne obrađivanje pogrešaka propagiranjem statusnih kodova (kao primjerice, u programskom jeziku C), korišteni su tzv. pametni pokazivači koji garantiraju otpornost koda na iznimke, tj. omogućuju da memorija koju objekt zauzima bude oslobođena u slučaju prestanka dosega pokazivača (npr. kod bacanja iznimke).

## 4.1. Arhitektura modela

Logika modela je enkapsulirana u razred Match (slika 4.1), koji predstavlja instancu igre. Da bi model funkcionirao, kod njegova stvaranja mu moraju biti proslijeđeni sljedeći podaci:

- cijeli broj koji predstavlja *seed* generatora slučajnih brojeva
- pokazivač na instancu skupa pravila naslijeđenog od *AbstractRuleset*
- instance upravljača naslijeđene od *AbstractEmpireController*
- instanca komponente za snimanje igre, naslijeđena od *AbstractMatchRecorder*




**Slika 4.1:** UML prikaz razreda Match

## 4.2. Razred *Match*

Razred *Match* predstavlja logiku jedne instance igre. Sadrži referencu na skup pravila (*AbstractRuleset*), instancu komponente za bilježenje tijeka igre (*AbstractMatchRecorder*) i dvije instance upravljača (*AbstractEmpireController*).

Nakon instanciranja, logika jednog ciklusa igre se izvodi pozivom metode *Tick*. Unutar metode *Tick*, izvodi se metoda *OnTick* carstva čiji igrač je na potezu te se brojač poteza uvećava za jedan.

## 4.3. Razred *AbstractRuleset*

*AbstractRuleset* je bazna klasa za skupove pravila. Za stvaranje konkretnog skupa pravila, potrebno je naslijediti klasu te implementirati sljedeće metode:

```
/// Returns the reference to the ruleset's building factory
virtual const AbstractBuildingFactory* GetBuildingFactory() const = 0;

/// Returns the reference to the ruleset's research factory
virtual const AbstractResearchFactory* GetResearchFactory() const = 0;

/// Returns the concrete combat resolver
virtual const AbstractCombatResolver* GetCombatResolver() const = 0;
```

Kao što je vidljivo iz priloženog sučelja, potrebno je implementirati i dvije varijante obrasca tvornice vezane uz skup pravila koji se stvara, te klasu koja koja služi kao simulator borbe.

Za brodove ne postoji tvornica, već je ta informacija zadržana u samom skupu pravila, a njihova registracija se vrši u konstruktoru klase izvedene od *AbstractRuleset*.

### 4.3.1. Razred *AbstractBuildingFactory*

Razred *AbstractBuildingFactory* je zadužen za stvaranje instance konkretne zgrade (*AbstractBuilding*) na temelju njezinog jedinstvenog identifikatora.

```
/// Creates an instance of a building of specified class identifier
std::unique_ptr<AbstractBuilding> Create(BuildingClassIdentifier id) const;
```

Registracija tipova zgrada se provodi u konstruktoru izvedene klase. Primjerice, za razred *DefaultBuildingFactory*:

```
DefaultBuildingFactory::DefaultBuildingFactory()
{
    this->RegisterPrototype(std::make_unique<MetalExtractor>())
    //...
}
```

Dakle, moguće je implementirati konkretnu tvornicu samo nasljeđivanjem i implementacijom konstruktora.

### 4.3.2. Razred *AbstractBuilding*

Razred *AbstractBuilding* predstavlja zgradu u nekom carstvu. Zgrade je moguće izgraditi određenom količinom resursa definiranom u metodi *GetBaseCost*, nakon čega se pojavljuju u određenom carstvu počevši s određenim stupnjem razvijenosti. Nakon što je zgrada izgrađena, svaki potez se izvršava njena metoda *OnTick*, koja se mora implementirati u naslijeđenoj klasi. Zgradu je moguće i unaprijediti (engl. *upgrade*), što zahtijeva određenu količinu resursa definiranu metodom *GetUpgradeCost*.

Zgrade mogu proizvoditi resurse, brodove, biti preduvjet za istraživanje tehnologije ili mogu imati proizvoljnu ulogu definiranu implementacijom metode *OnTick*.

### 4.3.3. Razred *AbstractResearchFactory*

Implementacija razreda *AbstractResearchFactory* je identična implementaciji razreda *AbstractBuildingFactory*, ali umjesto instance zgrade stvara instancu tehnologije (*AbstractResearch*).

### 4.3.4. Razred *AbstractResearch*

Razred *AbstractResearch* predstavlja tehnologiju koja se istražuje u nekom carstvu. Tehnologije je moguće istražiti ulaganjem resursa, čime se povećava njihov stupanj

razvijenosti. Cijena u resursima je definirana metodama *GetBaseCost* i *GetUpgradeCost* Za razliku od zgrada, tehnologija nema definiran skup radnji svaki potez, već konkretan učinak tehnologije na igru ovisi o pravilima.

### 4.3.5. Razred *AbstractCombatResolver*

Razred *AbstractCombatResolver* služi za simulaciju borbe u igri, te na temelju ishoda izvodi određene radnje. Za implementaciju je potrebno implementirati metodu *Resolve*:

```
virtual void Resolve(Match& match, Empire& attacker, Empire& defender) const = 0;
```

## 4.4. Razred *ShipClass*

Razred *ShipClass* sadrži informacije o određenoj vrsti broda. Svaka vrsta broda sadrži osnovne vrijednosti njegovih atributa - napad (*BaseAttack*), oklop (*BaseHull*), štit (*BaseShields*) i kapacitet tereta (*BaseCargoCapacity*). Uz osnovne atribute, u ovom razredu je spremljena i *rapidfire* informacija, te jedinstveni identifikator broda i ime.

Razred također služi i kao tvornica za konkretan brod kojeg razred opisuje.

Izdvajanje ovih informacija iz same reprezentacije broda u memoriji je učinjeno da se uštedi na memorijskom prostoru, budući da u jednoj instanci igre tipično ima mnogo brodova.

### 4.4.1. Struktura *Ship*

Struktura Ship predstavlja reprezentaciju broda koja se smješta u flotu i sudjeluje u borbi. Sadrži samo referencu na odgovarajuću instancu razreda *ShipClass* te vrijednosti atributa koje na odgovarajuće vrijednosti inicijalizira razred deriviran iz razreda *AbstractCombatResolver*.

## 4.5. Struktura *ResourceAmount*

Struktura *ResourceAmount* predstavlja količinu resursa. Postoje dva resursa - metal i kristal, a ova struktura enkapsulira logiku rukovanja tim resursima te pruža jednostavno korištenje u aritmetičkim izrazima i serijalizaciju u tok podataka (engl. *stream*) radi ispisa.

## 4.6. Razred *Empire*

Razred *Empire* predstavlja carstvo u vlasništvu nekog igrača. Sadrži listu zgrada (*AbstractBuilding*), tehnologija (*AbstractResearch*), flotu (*Ship*) i količinu resursa (*ResourceAmount*) koju igrač posjeduje.

Sadrži instancu razreda *AbstractEmpireController*, koji predstavlja sučelje za uzimanja ulaznih parametara od igrača te izvođenje akcija na temelju njih.

Metoda *OnTick* se izvodi svaki potez, a ona izvodi metodu *OnTick* svih zgrada, te nakon toga zahtijeva naredbu od instance razreda *AbstractEmpireController*.

### 4.6.1. Razred *AbstractEmpireController*

Razred *AbstractEmpireController* je sučelje za dobivanje naredbi od igrača. Igrač može biti čovjek ili program, ovisno o implementaciji sučelja. Za implementaciju je potrebno naslijediti razred te implementirati metodu *FetchCommand*:

```
/// Decide the course of action and return the command to be executed
virtual std::unique_ptr<AbstractCommand> FetchCommand(const Empire& empire) = 0;
```

Budući da je upravljač vezan za carstvo, kod instanciranja je potrebno proslijediti referencu na carstvo:

```
AbstractEmpireController(Empire& empire) : empire(empire) {};
```

Međutim, instanca carstva se stvara tek unutar konstruktora razreda *Match*, koji već očekuje da mu se proslijedi informacija o odgovarajućim upravljačima koji će se koristiti. Jedan način je da se to ostvari je tzv. lijena inicijalizacija (engl. *lazy initialization*), ali u skladu s *RAII* principom, to je loša praksa i objekt mora biti inicijaliziran u potpunosti bez da sadrži neko polu-ispravno stanje nakon inicijalizacije.

Zbog toga se koristi uzorak *creator*, koji omogućuje stvaranje instance razreda koji služi kao tvornica jedne vrste objekta. Na ovaj način moguće je razriješiti kružne reference u inicijalizaciji objekata bez posezanja za lijenom inicijalizacijom.

```
class AbstractMatchRecorderCreator
{
public:
    virtual ~AbstractMatchRecorderCreator() = default;

    virtual std::unique_ptr<AbstractMatchRecorder> Create(Match& match) const = 0;
};

template<class T>
class MatchRecorderCreator : public AbstractMatchRecorderCreator
```

```
{
public:
    std::unique_ptr<AbstractMatchRecorder> Create(Fercraft::Match& match) const override
    {
        return std::make_unique<T>(match);
    }
};
```

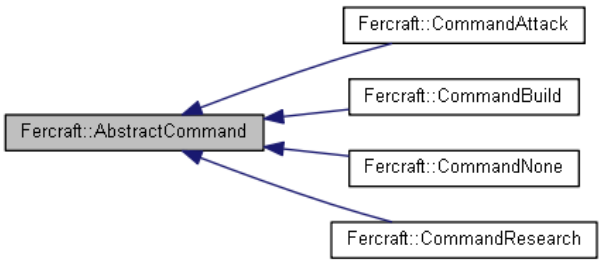
Dakle, razred *Empire* ne prima referencu na sam upravljač, već instancu njegovog kreatora.

### 4.6.2. Razred *AbstractCommand*

Razred *AbstractCommand* je bazni razred za naredbe koje proizvodi *AbstractEmpireController*. Predstavlja jednu naredbu igrača, te sadrži jedinstveni identifikator vrste naredbe te naredbu *Execute* koja implementira logiku same naredbe:

```
/// Returns the concrete command class's unique identifier
virtual CommandIdentifier GetIdentifier() const = 0;

/// Executes the command over the specified empire
virtual void Execute(Empire& empire) = 0;
```

Navedene metode su implementirane u razredima *CommandAttack*, *CommandBuild*, *CommandResearch* i *CommandNone* (slika 4.2).




**Slika 4.2:** Prikaz naslijeđivanja razreda *AbstractCommand*

Implementacije razreda naredbi su vrlo jednostavne. Naredba *CommandNone* preskače potez, *CommandResearch* pokreće istraživanje tehnologije čiji identifikator je proslijeđen kao parametar kod instanciranja naredbe, *CommandBuild* pokreće izgradnju zgrade čiji identifikator je proslijeđen kao parametar kod instanciranja naredbe, a *CommandAttack* inicira borbu.

## 4.7. Razred *AbstractMatchRecorder*

Razred *AbstractMatchRecorder* predstavlja sučelje preko kojeg instanca igre bilježi tijek igre. Definiran je skup događaja koji se bilježi, te za svaki postoji metoda u sučelju.

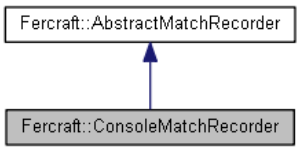
Primjerice, postoje sljedeće metode:

```
virtual void LogBuildingTick(const Empire& empire, const AbstractBuilding* building);
virtual void LogBuildingUpgrade(const Empire& empire, const AbstractBuilding* building);
virtual void LogResearchUpgrade(const Empire& empire, const AbstractResearch* research);
```

Kada se u instanci igre dogodi određeni događaj, poziva se jedna od metoda sučelja, čime se bilježi tijek igre.

### 4.7.1. Razred *ConsoleMatchRecorder*

Razred *ConsoleMatchRecorder* je jednostavna implementacija razreda *AbstractMatchRecorder* (slika 4.3) koja bilježi tijek igre na standardni izlaz. Format ispisa je zamišljen da omogući jednostavno parsiranje i analizu tijeka vanjskom aplikacijom ili skriptom.



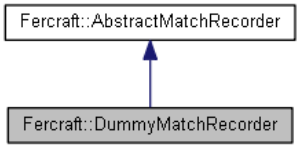

**Slika 4.3:** Prikaz naslijeđivanja razreda *ConsoleMatchRecorder*

Primjer ispisa kojeg generira ovaj razred je sljedeći:

```
------ EMPIRE_0 --- TURN 000000000 ------
0 BUILDING_TICK: EMPIRE_0, METAL_EXTRACTOR
0 RESOURCE_CHANGE: EMPIRE_0, (100,0), delta=(100,0)
0 BUILDING_TICK: EMPIRE_0, CRYSTAL_EXTRACTOR
0 RESOURCE_CHANGE: EMPIRE_0, (100,100), delta=(0,100)
0 BUILDING_UPGRADE: EMPIRE_0, CRYSTAL_EXTRACTOR(1)
0 RESOURCE_CHANGE: EMPIRE_0, (50,0), delta=(-50,-100)
------ EMPIRE_1 --- TURN 000000001 ------
1 BUILDING_TICK: EMPIRE_1, METAL_EXTRACTOR
1 RESOURCE_CHANGE: EMPIRE_1, (100,0), delta=(100,0)
1 BUILDING_TICK: EMPIRE_1, CRYSTAL_EXTRACTOR
1 RESOURCE_CHANGE: EMPIRE_1, (100,100), delta=(0,100)
```

### 4.7.2. Razred *DummyMatchRecorder*

Razred *DummyMatchRecorder* je implementacija razreda *AbstractMatchRecorder* (slika 4.4) koja ne radi ništa, već je namijenjena aplikacijama u kojima se istovremeno izvodi više instanci igre, primjerice u paraleliziranim evolucijskim algoritmima, kod kojih bi bilo kakvo bilježenje tijeka igre znatno usporavalo rad programa.




**Slika 4.4:** Prikaz naslijeđivanja razreda *DummyMatchRecorder*

## 4.8. Definirani skup pravila

Definiran je skup pravila kao primjer za proširenje modela.

### 4.8.1. Zgrade

Definirano je šest tipova zgrada.

#### Rudnik metala

Rudnik metala (engl. *Metal extractor*) proizvodi određenu količinu metala svaki potez. Količina metala koju rudnik proizvodi se računa po sljedećoj formuli:

$$BaseAmount + BaseAmount \cdot (Level - 1) \cdot 0.25,$$

gdje je *BaseAmount* osnovna količina metala koju rudnik proizvodi kada ima stupanj razvijenosti (engl. *Level*) jednak 1. Rudnik metala se na početku igre nalazi u carstvu kao zgrada stupnja razvijenosti jedan.

#### Rudnik kristala

Rudnik kristala (engl. *Crystal extractor*) proizvodi određenu količinu kristala svaki potez. Količina kristala koju rudnik proizvodi se računa po sljedećoj formuli:

$$BaseAmount + BaseAmount \cdot (Level - 1) \cdot 0.25,$$

gdje je *BaseAmount* osnovna količina kristala koju rudnik proizvodi kada ima stupanj razvijenosti (engl. *Level*) jednak 1. Rudnik kristala se na početku igre nalazi u carstvu kao zgrada stupnja razvijenosti jedan.

### Tvornice brodova

Definirane su 4 tvornice brodova koje se na početku igre ne nalaze u carstvu. Dakle, za svaku definiranu vrstu broda postoji jedna tvornica. Svaka tvornica ima brojač koji se uvećava svaki potez. Kada brojač dosegne trajanje proizvodnje (engl. *production duration*), u flotu igrača se dodaje novi brod odgovarajuće vrste.

Nakon izgradnje, unaprijeđivanje zgrade dijeli trajanje proizvodnje broda na pola od prethodne vrijednosti. Dakle, nakon određenog broja unaprijeđenja, trajanje proizvodnje će postati 0, što znači da će zgrada proizvesti novi brod svaki potez.

### Unaprijeđenje zgrada

Trošak za unaprijeđenje zgrade se računa po sljedećoj formuli:

$$UpgradeCost = BaseCost \cdot e^{Level \cdot 0.5},$$

odnosno po sljedećoj formuli za zgrade koje se nalaze u carstvu na početku igre:

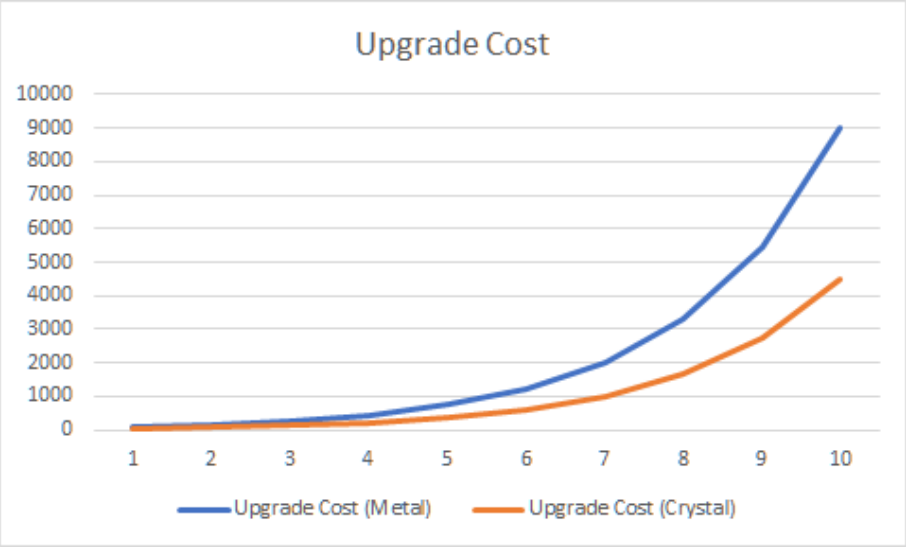
$$UpgradeCost = BaseCost \cdot e^{(Level-1) \cdot 0.5},$$

gdje je *UpgradeCost* trošak unaprijeđenja, *BaseCost* osnovni trošak izgradnje zgrade, a *Level* trenutni stupanj razvijenosti zgrade. Budući da se resurs sastoji od dvije komponente (metal i kristal), konačni trošak u resursima je izračunat za svaku komponentu posebno. Primjerice, za količinu resursa koja se sastoji od 100 jedinica metala i 50 jedinica kristala, troškovi kroz 10 stupnjeva razvijenosti su vidljivi u tablici 4.1.

**Tablica 4.1:** Primjer troškova razvoja kroz stupnjeve razvijenosti

| Level | Upgrade Cost (Metal) | Upgrade Cost (Crystal) |
|---|---|---|
| 1 | 100 | 50 |
| 2 | 164 | 82 |
| 3 | 271 | 135 |
| 4 | 448 | 224 |
| 5 | 738 | 369 |
| 6 | 1218 | 609 |
| 7 | 2008 | 1004 |
| 8 | 3311 | 1655 |
| 9 | 5459 | 2729 |
| 10 | 9001 | 4500 |

Grafički prikaz primjera je vidljiv na slici 4.5.




**Slika 4.5:** Graf troškova razvoja za navedeni primjer

### 4.8.2. Tehnologija

Svrha tehnologije u definiranom skupu pravila je poboljšavanje atributa brodova u borbi. Definirane su sljedeće tehnologije:

- Tehnologija oružja, koja poboljšava napad brodova
- Tehnologija oklopa, koja poboljšava fizički oklop brodova

– Tehnologija štitova, koja poboljšava energetske štitove brodova

Cijena unaprijeđenja tehnologije je definirana formulom koja se koristi za računanje troškova unaprijeđenja zgrada.

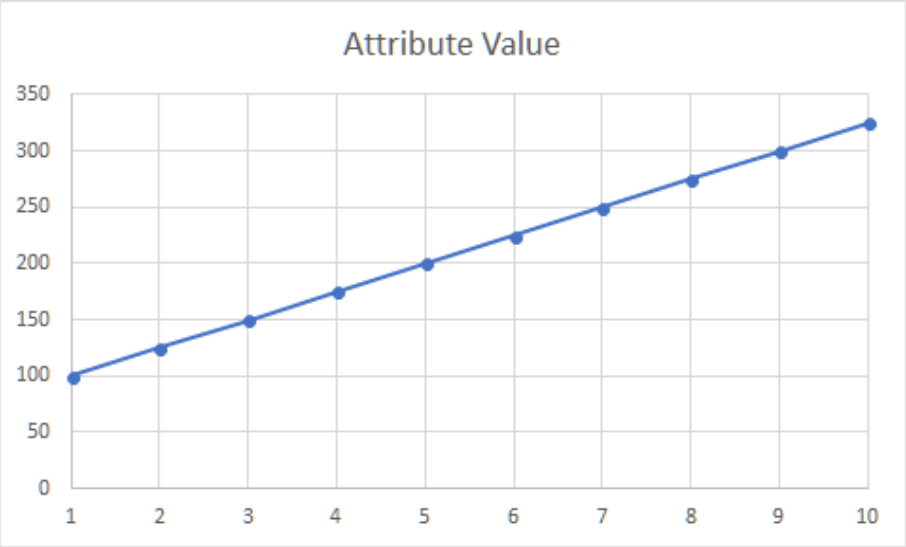
Svaka dodatna razina unaprijeđenja uvećava atribut broda za 25% osnovne vrijednosti tog atributa, što znači da se radi o linearnom rastu. Budući da su troškovi unaprijeđenja eksponencionalni, igrač mora pažljivo razmotriti kada je isplativo uložiti resurse u istraživanje tehnologije.

Primjerice, za osnovnu vrijednost atributa jednaku 100, istraživanje tehnologije kroz 10 razina će postići vrijednosti atributa opisane tablicom 4.2.

**Tablica 4.2:** Primjer dobitaka istraživanja kroz razine

| Level | Attribute Value |
|---|---|
| 1 | 100 |
| 2 | 125 |
| 3 | 150 |
| 4 | 175 |
| 5 | 200 |
| 6 | 225 |
| 7 | 250 |
| 8 | 275 |
| 9 | 300 |
| 10 | 325 |

Grafički prikaz primjera vidljiv je na slici 4.6.




**Slika 4.6:** Graf dobitaka istraživanja za navedeni primjer

### 4.8.3. Sustav borbe

Sustav borbe je definiran po uzoru na sustav borbe iz igre *OGame*. Borbu može inicirati igrač koji je trenutno na potezu. Jednom kada je borba inicijalizirana, uspoređuju se flote igrača. Ako bilo koji od igrača nema nijedan brod, pobjeđuje suprotni igrač. Ako oba igrača nemaju brodove, borba je neriješena, tj. nema posljedica.

Borba ima šest rundi. Prvo napada flota napadača - svaki brod slučajno odabire jedan neprijateljski brod kojeg će napasti, te izvodi napad. Napad prvo radi štetu energetskom štitu broda, a kada se štit iscrpi, napad radi štetu fizičkom oklopu broda. Energetski štitovi se obnavljaju na početku svake runde, ali fizički oklop ostaje oštećen za vrijeme trajanja borbe. Brodovi posjeduju tzv. *rapidfire* listu. Ako brod napadne brod protiv kojeg ima *rapidfire*, postoji vjerojatnost da će dobiti pravo još jednog napada na neki slučajni brod u istoj rundi. Jedan unos u *rapidfire* listi je sadržaja (brod, N), gdje je N vrijednost koja predstavlja prosječan broj napada koje će brod izvesti pomoću *rapidfirea* ako svaki puta odabere metu ciljanog tipa.

Primjerice, ako brod vrste *Heavy Fighter* protiv brodova vrste *Light Fighter* ima *rapidfire* koji iznosi 5. To znači da je vjerojatnost da će brod ponovo moći pucati unutar iste runde jednaka:

$$\frac{Rapidfire - 1}{Rapidfire} = \frac{5-1}{5} = \frac{4}{5} = 0.8$$

Nakon što su svi brodovi jednog igrača napali, na potez dolazi flota drugog igrača, te se tako izmjenjuju dok jedan igrač ne pobijedi (uništi sve neprijateljske brodove)

ili dok ne završi svih šest rundi. Ako nakon šest rundi nema pobjednika, borba je neriješena te se obje flote vraćaju u carstva bez posljedica. Ako pobijedi napadač, od poraženog uzima najviše onoliko resursa koliko je ukupan kapacitet njegovih brodova. Šteta načinjena brodovima unutar borbe se ne prenosi u sljedeću borbu, osim ako je brod uništen.

### 4.8.4. Brodovi

Definirane su četiri vrste broda.

#### *Light Fighter*

Slab, ali jeftino i brzo proizvediv brod s malom nosivošću tereta. Nema *rapidfire* protiv nijednog broda. Iako potrošan, koristan je kao mamac - umanjuje šansu da će neprijatelj naciljati vrijedniji brod.

#### *Heavy Fighter*

Jača varijanta prethodnog broda. Posjeduje *rapidfire* vrijednosti 5 protiv brodova klase *Light Fighter*.

#### *Battlecruiser*

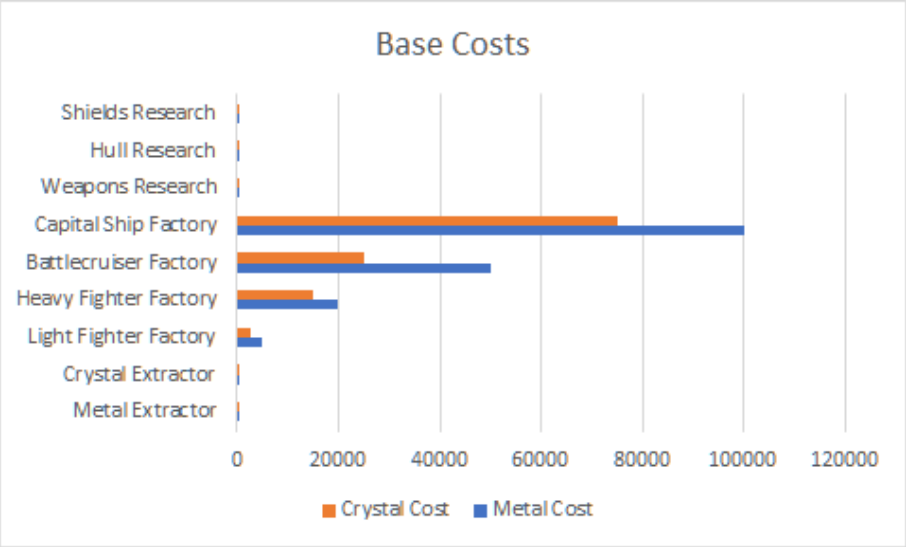
Dobar omjer cijene proizvodnje i vatrene moći. Posjeduje *rapidfire* vrijednosti 5 protiv brodova klase *Heavy Fighter* te vrijednosti 20 protiv klase *Light Fighter*.

#### *Capital Ship*

Najjači brod u igri, ali vremenski najskuplji za proizvodnju. Posjeduje *rapidfire* vrijednosti 5 protiv brodova klase *Battlecruiser*, vrijednosti 20 protiv klase *Heavy Fighter* te 50 protiv klase *Light Fighter*.

### 4.8.5. Pregled troškova

Budući da je predviđeno da prosječna igra traje dugo, osnovne cijene tvornica brodova su po jačini poredane eksponencionalno. Osnovne cijene zgrada i tehnologija su, za razliku od tvornica brodova, puno niže (slika 4.7).

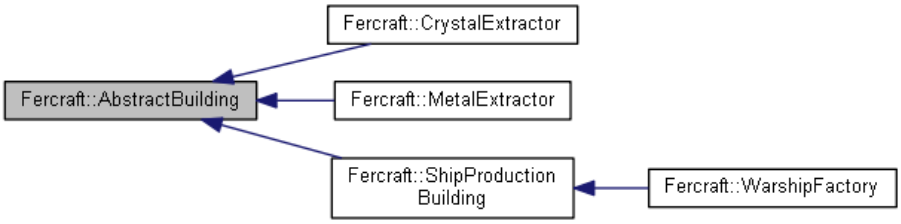



**Slika 4.7:** Graf osnovnih cijena

## 4.9. Implementacija definiranog skupa pravila

### 4.9.1. Implementacija zgrada

Zgrade su implementirane nasljeđivanjem razreda *AbstractBuilding*, uz izuzetak tvornice brodova (slika 4.8).




**Slika 4.8:** Prikaz nasljeđivanja razreda *AbstractBuilding*

Za svaku zgradu su implementirane odgovarajuće metode *GetClassId*, *GetName*, *Clone*, *GetBaseCost*, *GetUpgradeCost* i *OnTick*.

Implementacija metode *OnTick* je gotovo ista kod razreda *MetalExtractor* i *CrystalExtractor*:

```
void MetalExtractor::OnTick(Empire& empire)
{
    AbstractBuilding::OnTick(empire);

    const auto baseAmount = ResourceAmount{ 100, 0 };
    empire.AddResources(baseAmount + baseAmount * (this->GetLevel()-1) * 0.25);
}
```

Za razliku od rudnika, implementacija tvornice brodova je razdvojena u dva razreda radi fleksibilnosti. Razred *ShipProductionBuilding* je apstraktni razred koji ima dvije virtualne metode:

```
virtual ShipClassIdentifier GetShipClassId() = 0;
virtual uint64_t GetBaseProductionDuration() = 0;
```

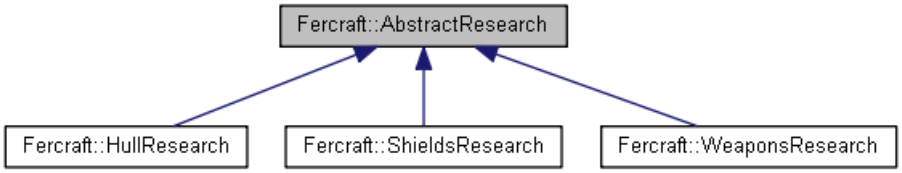
Razred *WarshipFactory* nasljeđuje razred *ShipProductionBuilding* te implementira navedene metode, uz parametrizaciju u konstruktoru:

```
WarshipFactory(BuildingClassIdentifier buildingClassId,
               std::string name,
               const ResourceAmount baseCost,
               ShipClassIdentifier shipClassId,
               uint64_t baseProductionDuration);
```

Na ovaj način je izbjegnuta potreba za implementiranjem četiri implementacije konkretne tvornice brodova - umjesto toga je stvorena jedna generička implementacija kojoj se brod i trajanje proizvodnje proslijeđuju kao parametri.

### 4.9.2. Implementacija tehnologije

Tehnologije su implementirane nasljeđivanjem razreda *AbstractResearch* (slika 4.9).

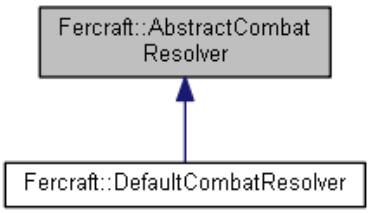



**Slika 4.9:** Prikaz nasljeđivanja razreda *AbstractResearch*

Za svaku tehnologiju su implementirane odgovarajuće metode *GetClassId*, *GetName*, *Clone*, *GetBaseCost*, *GetUpgradeCost*. Za razliku od zgrada, tehnologije nemaju metodu *OnTick* već je učinak tehnologije definiran u razredu *DefaultCombatResolver*.

### 4.9.3. Implementacija sustava borbe

Tehnologija ima izravan učinak na atribute brodova u borbi, a to je definirano u komponenti *DefaultCombatResolver* naslijeđenoj od *AbstractCombatResolver* (slika 4.10).




**Slika 4.10:** Prikaz naslijeđivanja razreda *AbstractCombatResolver*

Atributi brodova s obzirom na stupanj razvoja tehnologije su implementirani na sljedeći način:

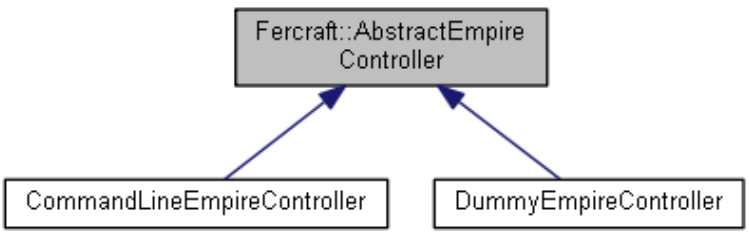
```
int64_t CalculateAttack(int64_t base, uint64_t researchLevel)
{
    return base + static_cast<int64_t>(base * (researchLevel-1) *
                                       RESEARCH_LEVEL_MULTIPLIER);
}

int64_t CalculateShields(int64_t base, uint64_t researchLevel)
{
    return base + static_cast<int64_t>(base * (researchLevel - 1) *
                                       RESEARCH_LEVEL_MULTIPLIER);
}

int64_t CalculateHull(int64_t base, uint64_t researchLevel)
{
    return base + static_cast<int64_t>(base * (researchLevel - 1) *
                                       RESEARCH_LEVEL_MULTIPLIER);
}
```

Gdje je *RESEARCH_LEVEL_MULTIPLIER* konstanta iznosa 0.25.

Implementacija metode *Resolve* je opširna, ali je zapravo izravna preslika sustava borbe opisanog u 4.8.3.

## 4.10. Primjer korištenja modela

Korištenje modela u klijentskoj aplikaciji zahtijeva implementaciju upravljača (*AbstractEmpireController*, slika 4.11) koji će služiti kao virtualni igrač u modelu igre.




**Slika 4.11:** Prikaz naslijeđivanja razreda *AbstractEmpireController*

Dakle, potrebno je naslijediti *AbstractEmpireController* i implementirati metodu *Fetch*. Iako je razred dizajniran s pretpostavkom da upravljač ne posjeduje stanje (engl. *state*), dozvoljeno je implementirati upravljač koji ima stanje ako programer vodi računa da se jedna instanca takvog upravljača koristi unutar točno jedne instance igre.

Priložen je projekt *fercraft-client* koji se nalazi u direktoriju *CMake* projekta. Sadrži implementacije dva upravljača.

Upravljač *CommandLineEmpireController* prima ulazne naredbe putem komandne linije. Služi kao jednostavno korisničko sučelje za ljudskog igrača.

Upravljač *DummyEmpireController* je jednostavan upravljač koji odluke donosi na temelju slučajnog odabira. Prikladan za testiranje modela.

Osim implementacije upravljača, instancirane su dodatne komponente modela kao što je opisano u 4.1.

Priložen je primjer programa u kojem dva upravljača tipa *DummyEmpireController* igraju jedan protiv drugog beskonačno, pri čemu je korišten podrazumijevani sustav pravila i *ConsoleMatchRecorder* kao komponenta za snimanje tijeka igre.

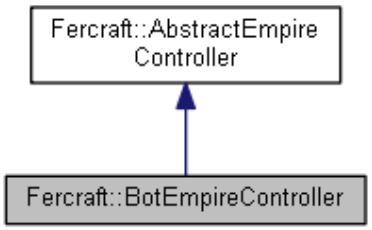
```
int main(int argc, char** argv)
{
    const std::shared_ptr<Fercraft::AbstractRuleset> ruleset =
    std::make_shared<Fercraft::DefaultRuleset>(Fercraft::DefaultRuleset());

    const auto seed = uint32_t(
        std::chrono::system_clock::now().time_since_epoch().count()
    );

    Fercraft::Match match
    {
        seed,
        ruleset,
        std::make_unique<Fercraft::EmpireControllerCreator<DummyEmpireController>>(),
        std::make_unique<Fercraft::EmpireControllerCreator<DummyEmpireController>>(),
        std::make_unique<Fercraft::MatchRecorderCreator<Fercraft::ConsoleMatchRecorder>>()
    };

    while (true)
    {
        match.Tick();
    }

    return 0;
}
```

# 5. Primjer uporabe u strojnom učenju

Funkcionalnost modela demonstrirana je na jednostavnom primjeru strojnog učenja.

## 5.1. Razred *BotEmpireController*

Definiran je razred *BotEmpireController* naslijeđen od *AbstractEmpireController* (slika 5.1) koji predstavlja umjetnog igrača koji donosi odluke na temelju stanja igre.




**Slika 5.1:** Prikaz nasljeđivanja razreda *BotEmpireController*

Logika razreda *BotEmpireController* opisana je sljedećim pseudo-kodom:

```
ako ima još naredbi u redu:
    izvedi naredbu i ukloni je iz reda
inače:
    za sve zgrade redom:
        ako ima dovoljno resursa, izgradi
    za sve tehnologije redom:
        ako ima dovoljno resursa, istraži
    ako je flota veća od neprijateljeve flote:
        napadni
    inače:
        preskoči potez
```

Dakle, definirani upravljač prvo ima skup naredbi koji predstavlja tzv. *build-order*, koji je isti u svakoj instanci igre i služi kao osnova za daljnji razvoj carstva. Nakon što su konzumirane sve naredbe iz reda, upravljač prelazi na donošenje odluke na temelju resursa i veličine flote.

Iako se na prvi pogled čini da bi upravljač mogao zauvijek graditi samo zgrade, razlika u linearnoj efikasnosti zgrade u usporedbi s eksponencijalnim rastom cijene jamči da će nakon nekog vremena postati neisplativo graditi taj tip zgrade, pa će upravljač odlučiti na izgradnju druge, i prijeći na istraživanje te potom borbu.

Definirani upravljač je potom iskorišten kao protivnik u treniranju linearne neuronske mreže. Iako ne previše efikasan protiv ljudskog igrača, upravljač je dovoljno dobar izazov za protivnika dobivenog nekom od jednostavnijih metoda strojnog učenja.

## 5.2. Struktura neuronske mreže

Definirana je neuronska mreža sa određenim ulazima i izlazima. Broj i veličina skrivenih slojeva je podložno konfiguraciji kroz tekstualne datoteke. Raspored ulaza i izlaza je sljedeći:

### 5.2.1. Ulazi

#### Dostupnosti zgrada i tehnologija

Za svaku zgradu i tehnologiju definiran je jedan ulaz koji može poprimiti vrijednost -1 (ako igrač ne može priuštiti zgradu ili tehnologiju) ili 1 (ako igrač može priuštiti zgradu ili tehnologiju).

#### Omjer veličine flote igrača i protivnika

Omjer može poprimiti vrijednost [-1, 1]. Računa se dijeljenjem veličine manje flote sa veličinom veće flote, pri čemu se rezultatu dodjeljuje pozitivan predznak ako je igrač vlasnik veće flote, odnosno negativan predznak ako je igrač vlasnik manje flote.

### 5.2.2. Izlaz

#### Naredba

Izlaz neuronske mreže je vrijednost u rasponu [-1, 1] koja se preslikava na naredbu koju će upravljač izvesti nad modelom.

## 5.3. Razred *NeuralEmpireController*

Definiran je razred *NeuralEmpireController* naslijeđen od razreda *AbstractEmpireController* koji sadrži logiku mapiranja izlaza linearne neuronske mreže na konkretne naredbe koje se proslijeđuju carstvu.

Budući da postoji 6 tipova zgrade, 3 vrste tehnologije i mogućnost napada, ukupan broj mogućih naredbi koji igrač može izvesti je 10. Neuronska mreža na svojem izlazu proizvodi jedan realan broj u rasponu [-1, 1], i taj broj se mapira na indeks konkretne naredbe na sljedeći način:

```
const int commandIndex = std::min(
9,
static_cast<int>(((this->neuralOutput + 1.0) / 2.0) * 10));
```

## 5.4. Problem stohastike modela

Zbog opisanog sustava borbe, javlja se nedeterminizam. Naime, model nema deterministično ponašanje zbog dva faktora:

- Mehanizma biranja mete brodova
- *Rapidfire* mehanike

Budući da je odabir mete svakog broda slučajan, za različiti *seed* postoji šansa da će ishod bitke drukčiji jer će brodovi potencijalno odabrati drukčije mete nego bi inače odabrali. Uz to, na ishod borbe drastično utječe i *rapidfire* mehanika, čija svaka iteracija povlači ponovni odabir mete, koji je također slučajan.

Iako u teoriji stohastički model upotrijebljen u postupku strojnog učenja neće davati pouzdane rezultate i može narušiti postupak učenja, u praksi se kod igre *OGame* pokazao vrlo uspješnim iako se radi o kompetitivnoj igri. Naime, nedeterminizam u modelu borbe omogućava igraču da pobijedi čak i ako je njegova flota nešto lošija, što znači da igrači trebaju uzeti stohastiku modela u obzir.

Također, u praksi u implementiranom modelu borbe sudjeluje tisuće brodova, što do neke mjere umanjuje učinak generatora slučajnih brojeva jer ishod više ovisi o kompoziciji flota igrača nego o odabiru mete.

U svrhe strojnog učenja, potrebno je jednu instancu igre simulirati nekoliko puta te kao dobrotu uzeti zbroj dobivenih dobrota, čime se umanjuje negativni utjecaj stohastike modela.

## 5.5. Provođenje postupka strojnog učenja

Za demonstraciju strojnog učenja nad modelom korišten je programski okvir *ECF* (*Evolutionary Computation Framework*) koji sadrži implementacije raznih evolucijskih algoritama i implementaciju neuronske mreže.

Kao evaluacijska funkcija se koristi instanca igre u kojoj navedeni upravljač igra protiv upravljača tipa *BotEmpireController*, bez ispisa na standardni izlaz. Krajnja dobrota jedinke je izražena kao razlika resursa igrača kojeg jedinka predstavlja i njegovog neprijatelja:

$$Dobrota = IznosResursaJedinke - IznosResursaProtivnika,$$

gdje je iznos resursa izračunat kao zbroj komponenti:

$$IznosResursa = IznosMetala + IznosKristala$$

Igra traje 1000 poteza. Kod za evaluaciju jedinke je sljedeći:

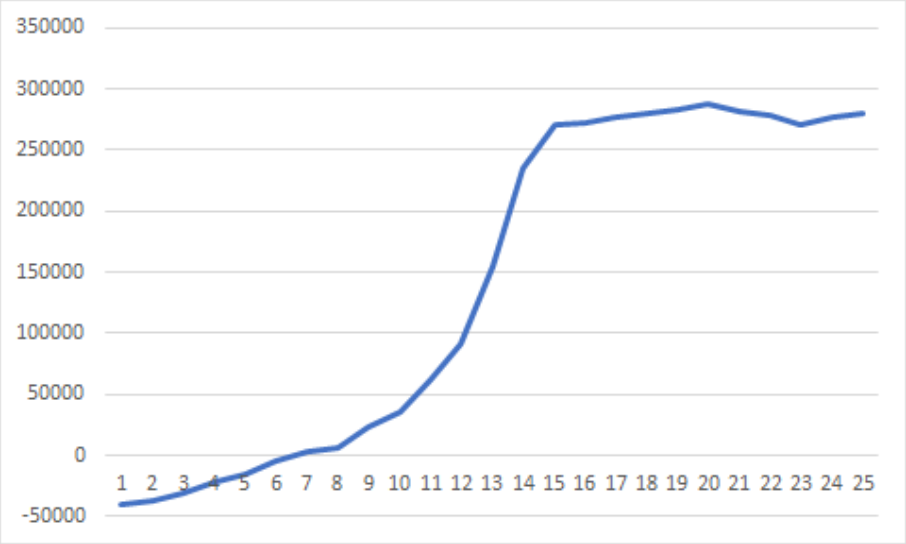
```
Fercraft::Match match
{
    seed,
    ruleset,
    std::make_unique<Fercraft::EmpireControllerCreator<NeuralEmpireController>>(),
    std::make_unique<Fercraft::EmpireControllerCreator<Fercraft::BotEmpireController>>(),
    std::make_unique<Fercraft::MatchRecorderCreator<Fercraft::DummyMatchRecorder>>()
};

// Play for 1000 turns
for (int i = 0; i < 1000; ++i)
{
    gen->setCurrentInput(CalculateNeuralNetworkInputs(match.GetEmpire(0)));

    std::vector<double> output;
    gen->calculateOutput(output);

    NeuralEmpireController* controller = static_cast<NeuralEmpireController*>(
    match.GetEmpire(0).GetEmpireController());

    controller->SetNeuralOutput(output[0]);

    match.Tick();
}

auto resourceDifference = match.GetEmpire(0).GetResources() -
                          match.GetEmpire(1).GetResources();

// Actual fitness score
double fit = resourceDifference.metal + resourceDifference.crystal;
```

## 5.6. Rezultati strojnog učenja

Upotrijebljen je algoritam *steady-state tournament* kod kojeg je veličina turnira 3, a kao genotip je korištena neuronska mreža s 2 skrivena sloja od 32 neurona i linearnom aktivacijskom funkcijom. Algoritam je izveden za populacije od 50, 500 i 5000 jedinki. Trajanje izvođenja algoritma je 25 generacija. Algoritam je za svaku varijantu izveden 10 puta zbog stohastičke prirode evolucijskih algoritama, te je za prikaz u nastavku korišten prosjek svih izvedenih iteracija.

### 5.6.1. Ponašanje prosječne dobrote

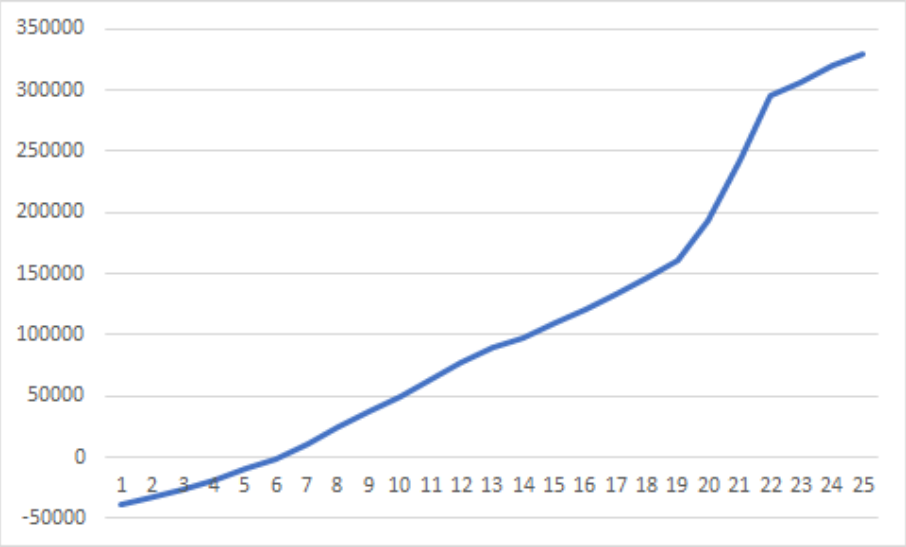
Ponašanje prosječne dobrote za veličinu populacije 50 je prikazano na slici 5.2.




**Slika 5.2:** Prosječna dobrota kroz 25 generacija za populaciju veličine 50

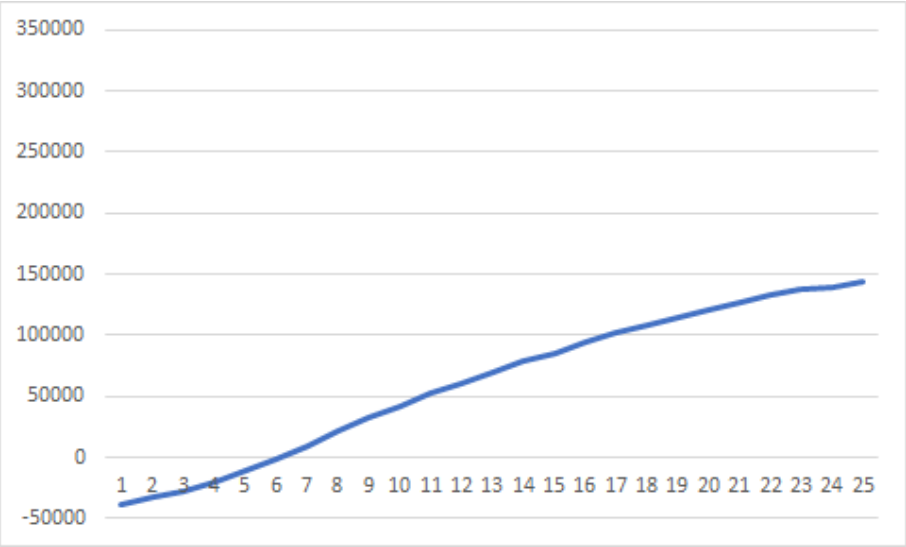
Iz priloženog je vidljivo da već u 6. generaciji dobivena umjetna inteligencija parira fiksnom računalnom protivniku, nakon čega kreće eksponencijalni rast prosječne dobrote koji zbog male veličine populacije brzo stagnira.

Veličina populacije je potom povećana na 500. Kao i kod populacije veličine 50, već u 6. generaciji dobivena umjetna inteligencija počinje parirati računalnom protivniku, nakon čega kreće nešto sporiji, ali stabilniji rast prosječne dobrote (slika 5.3).



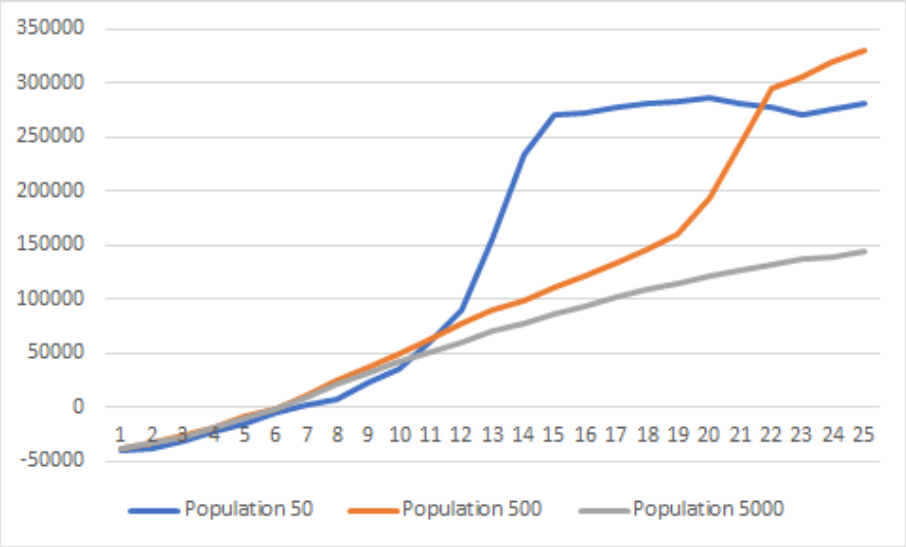

**Slika 5.3:** Prosječna dobrota kroz 25 generacija za populaciju veličine 500

Na kraju je veličina populacije povećana na 5000. Kao i u prethodnim konfiguracijama, nakon 6. generacije prosječna dobrota postaje pozitivna, ali dobiveni rast prosječne dobrote je gotovo duplo sporiji od prethodnog. Međutim, prosječna dobrota raste puno stabilnije.



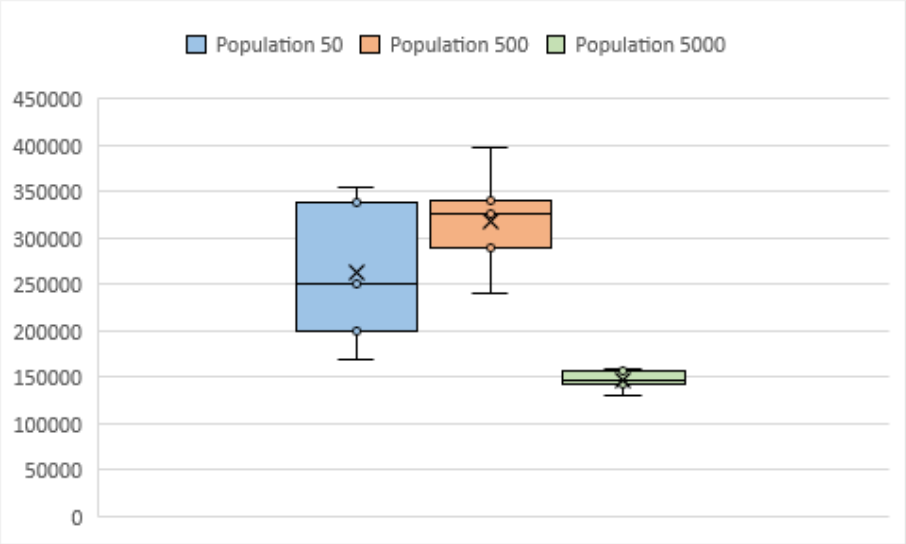

**Slika 5.4:** Prosječna dobrota kroz 25 generacija za populaciju veličine 5000

Na slici 5.5 prikazan je rast prosječne dobrote za sve navedene varijante.




**Slika 5.5:** Prosječna dobrota kroz 25 generacija za različite populacije

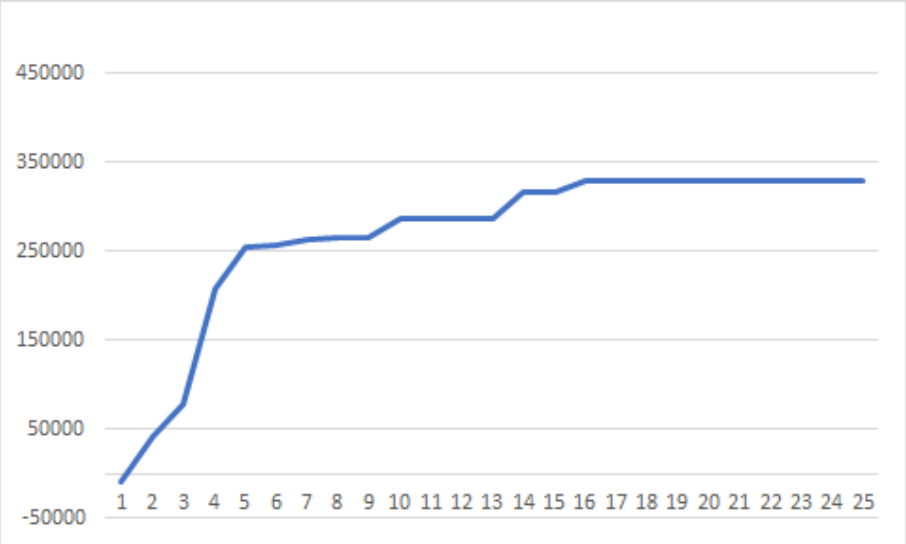
Na slici 5.6 je prikazan *box plot* krajnje prosječne dobrote za provedena mjerenja.



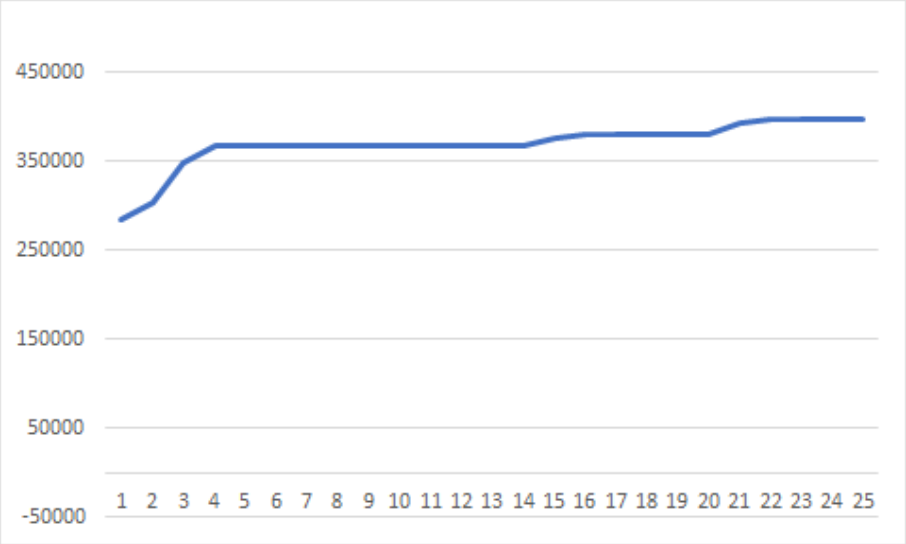

**Slika 5.6:** *Box plot* prosječne dobrote

### 5.6.2. Ponašanje najveće dobrote

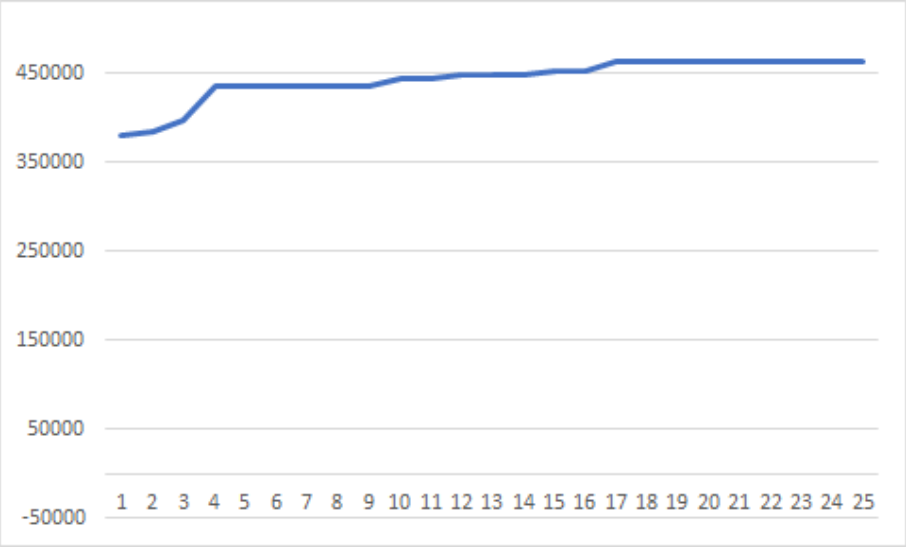
Prethodni testovi su ponovljeni uzimajući u obzir najveću dobrotu u generaciji umjesto prosječne dobrote. Rezultati mjerenja su prikazani na slikama 5.7, 5.8, 5.9. Kretanje najvećih dobrota za sve tri varijante je prikazano na slici 5.10.



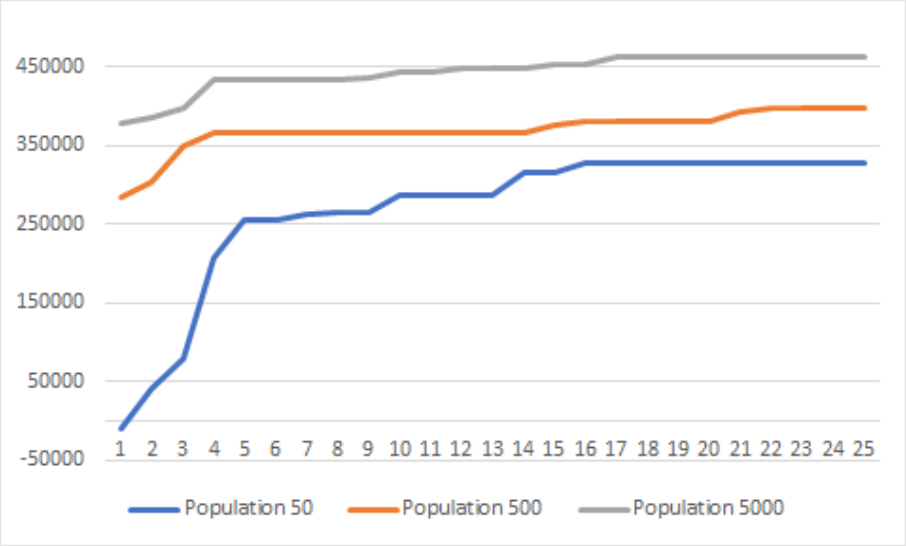

**Slika 5.7:** Najveća dobrota kroz 25 generacija za populaciju veličine 50




**Slika 5.8:** Najveća dobrota kroz 25 generacija za populaciju veličine 500

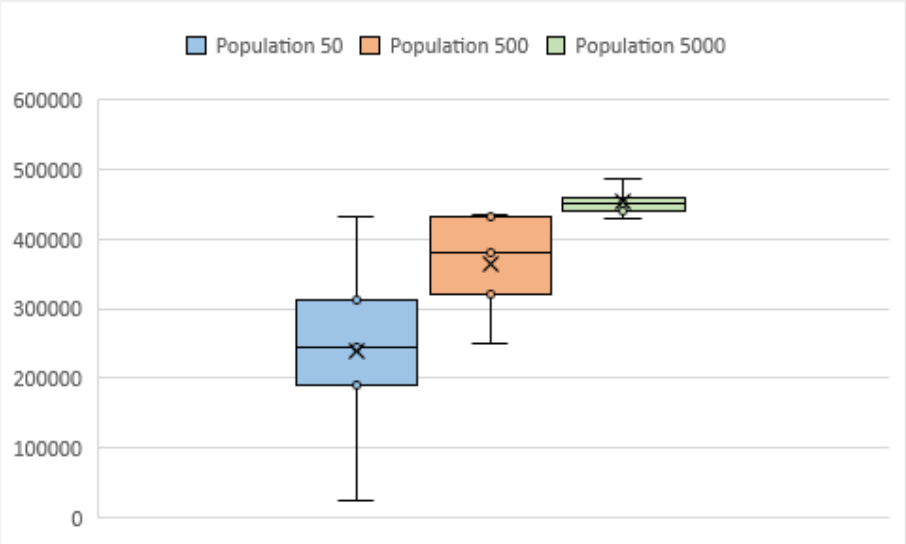



**Slika 5.9:** Najveća dobrota kroz 25 generacija za populaciju veličine 5000




**Slika 5.10:** Najveća dobrota kroz 25 generacija za različite populacije

Na slici 5.11 je prikazan *box plot* najvećih dobrota za provedena mjerenja.




**Slika 5.11:** *Box plot* najveće dobrote

Iz priloženog je vidljivo da najveća dobrota raste i dobivena umjetna inteligencija uspješno pobjeđuje računalnog igrača. Također je vidljivo da dobrota stagnira oko vrijednosti 450,000, međutim, to potencijalno ukazuje na ograničenje samog modela.

Evoluirani virtualni igrač tipično pobjeđuje fiksnog računalnog protivnika za red veličine stotinu tisuća resursa.

Doduše, fiksni računalni protivnik je vrlo jednostavan i predvidiv, pa dobivena umjetna inteligencija ne treba biti veoma napredna da bi ga pobijedila. Da bi se proizvela naprednija umjetna inteligencija, potrebno je napraviti nekoliko računalnih protivnika koji koriste različite taktike, čime bi se postigao širi skup podataka za učenje, te upotrijebiti neku od naprednijih metoda strojnog učenja. Međutim, to je izvan opsega ovog rada.

# 6. Upute za korištenje

## 6.1. Kompilacija modela

Kompilacija modela moguća je korištenjem priloženog *Visual Studio 2017* projekta ili izgradnjom projekta kroz sustav *CMake* za razvojno okruženje po izboru.

Budući da model ovisi samo o standardnim bibliotekama, moguće ga je kompilirati bez dodatne konfiguracije. Krajnji rezultat izgradnje modela je statička biblioteka koja se uz skup C++ *header* datoteka može distribuirati aplikaciji koja će je koristiti.

### 6.1.1. Kompilacija modela kroz razvojno okruženje *Visual Studio*

Potrebno je pomoću razvojnog okruženja *Visual Studio 2017* otvoriti datoteku *fercraft.sln*. Otvara se okruženje s nekoliko projekata, od kojih je potrebno odabrati *fercraft-core* i odabrati opciju *Build*.

Izgradit će se statička biblioteka u direktoriju čije ime odgovara odabranoj konfiguraciji (*Debug* ili *Release*). Za korištenje modela u sklopu evolucijskih algoritama se preporuča izgradnja u *Release* konfiguraciji jer tada kompilator nad kodom izvodi optimizacije, što znači da će program raditi efikasnije.

### 6.1.2. Kompilacija modela kroz sustav *CMake*

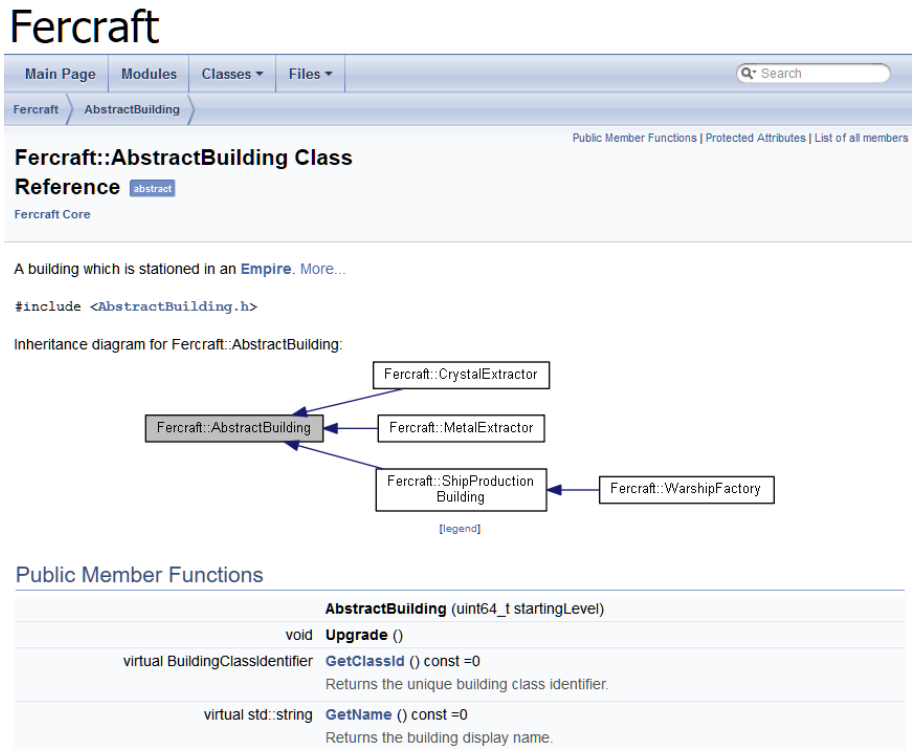
Najprije je potrebno u vršnom direktoriju projekta stvoriti direktorij "build", te se pozicionirati unutar njega. Nakon toga naredbom "cmake .." pokrenuti *CMake*, koji će iz direktorija iznad generirati određeno okruženje za kompilaciju projekta.

Ovisno o generiranom okruženju, potrebno je pokrenuti postupak kompilacije. Primjerice, na *Linux* sustavu se podrazumijevano generira *makefile*, pa pokretanjem naredbe *make* počinje postupak kompilacije, nakon kojeg se u direktoriju *build* stvaraju dodatni direktoriji s izvršnim datotekama i ostalim rezultatima kompilacije.

## 6.2. Izgradnja dokumentacije

Programski kod je dokumentiran sintaksom podržanom generatorom dokumentacije *Doxygen*. Za izgradnju dokumentacije s grafovima potreban je *Doxygen* i *Graphwiz*, a put do njihovih izvršnih datoteka mora biti dodan u sistemsku varijablu *PATH*.

Nakon instalacije preduvjeta, potrebno je pozicionirati se u vršni direktorij projekta te pokrenuti *Doxygen*. Generirana dokumentacija (slika 6.1) se nalazi u direktoriju *docs*.




**Slika 6.1:** Prikaz generirane dokumentacije

## 6.3. Ugrađivanje modela

Ugrađivanje modela u aplikacije je vrlo jednostavna - potrebno je klijentskoj aplikaciji distribuirati samo *header* datoteke iz direktorija *fercraft-core/include* te binarnu *lib* datoteku dobivenu izgradnjom modela iz izvornog koda.

Nakon što su datoteke smještene na odgovarajuća mjesta, potrebno je dati uputu kompilatoru da izvede *linking* klijentske aplikacije i dobivene biblioteke.

## 6.4. Implementacija vlastitog računalnog igrača

Implementacija vlastitog igrača će biti opisana na temelju primjera *DummyEmpireController*. Za implementaciju je potrebno stvoriti klasu naslijeđenu od klase *AbstractEmpireController*:

```
class DummyEmpireController : public Fercraft::AbstractEmpireController
```

Potrebno je naslijediti konstruktor klase te u inicijalizacijskoj listi pozvati konstruktor bazne klase:

```
DummyEmpireController(Fercraft::Empire& empire) : AbstractEmpireController(empire)
{
}
```

Na kraju je potrebno napraviti *override* metode *FetchCommand* koja se izvršava svaki potez. Primjerice, ovakva implementacija će svaki potez izvesti slučajno odabranu naredbu.

```
std::unique_ptr<Fercraft::AbstractCommand> FetchCommand(const Fercraft::Empire& empire) override
{
    const int commandIndex = rand() % 10;
    switch (commandIndex)
    {
    case 0: return std::make_unique<Fercraft::CommandBuild>(0);
    case 1: return std::make_unique<Fercraft::CommandBuild>(1);
    case 2: return std::make_unique<Fercraft::CommandBuild>(2);
    case 3: return std::make_unique<Fercraft::CommandBuild>(3);
    case 4: return std::make_unique<Fercraft::CommandBuild>(4);
    case 5: return std::make_unique<Fercraft::CommandBuild>(5);
    case 6: return std::make_unique<Fercraft::CommandResearch>(0);
    case 7: return std::make_unique<Fercraft::CommandResearch>(1);
    case 8: return std::make_unique<Fercraft::CommandResearch>(2);
    case 9: return std::make_unique<Fercraft::CommandAttack>();
    default: return std::make_unique<Fercraft::CommandNone>();
    }
}
```

Uvid u stanje carstva moguće je dobiti pozivom metoda razreda *Empire* čija instanca je proslijeđena kao parametar. Također, moguće je dobiti i uvid u stanje cijele igre pozivom metode *GetMatch* iz razreda *Empire*, preko čega je moguće doći i do stanja protivničkog carstva. Vrijedi napomenuti da je uvid u stanje protivničkog carstva sukladan pravilima igre, tj. ne postoji koncept magle rata (engl. *fog of war*) kao u tipičnoj računalnoj strateškoj igri.

# 7. Zaključak

Ostvareni model je vrlo fleksibilan i moguće je znatno proširiti njegovu funkcionalnost. Također, zahvaljujući korištenju sustava za izgradnju projekata *CMake*, model je vrlo lako prenosiv i moguće ga je izgraditi na bilo kojem sustavu za koji postoji kompilator za jezik C++ verzije 14.

Iako se simulacija modela izvodi dovoljno brzo za potrebe prosječnog korisnika, očito usko grlo je sustav borbe ostvaren kao primjer. Naime, sustav borbe iterira kroz svaki brod da bi odredio njegov efekt na ishod, pa je složenost borbe najmanje O(N), što je loše ako u borbi sudjeluje mnogo brodova. Zbog toga bi bilo bolje pronaći način da se računanje ishoda borbe ostvari analitički, čime bi složenost postala O(1).

Zahvaljujući pažljivom planiranju dijeljenja resursa između instanci igre, moguće je u paraleliziranom okruženju pokrenuti više instanci modela koji dijele skup pravila bez pojave greške ili nedefiniranog ponašanja.

Ovakav model strateške igre može imati široku uporabu u raznim područjima. Primjerice, može biti korišten kao *benchmark* raznih metoda strojnog učenja, biti predmet natjecanja iz izrade umjetne inteligencije ili biti predmet laboratorijske vježbe u sklopu nekog kolegija čija tema je vezana uz evolucijske algoritme ili strojno učenje.

# LITERATURA

[1] Nepoznat autor. *History of Chess: The Basics*. 2009. URL https://www.chess.com/article/view/the-history-of-chess.

[2] OpenAI. *OpenAI: Dota 2*. 2017. URL https://blog.openai.com/dota-2/.

[3] Umut Riza. *Artificial intelligence in real-time strategy games*. 2009. URL http://hevi.info/tag/artificial-intelligence-in-real-time-strategy-games/.

## Implementacija modela strateške igre


### Sažetak

Opisana problematika modeliranja strateških igara za više igrača s obzirom na uvjete igranja i izvedbu poteza. Istražene mogućnosti uporabe strojnog učenja u modeliranju računalnih igrača strateških igara i navedena postojeća rješenja. Ostvaren modularni programski sustav za simulaciju i prikaz strateške igre za više igrača. U sustav dodana mogućnost prikaza te proizvoljni model računalnog igrača. Ispitana učinkovitost postupaka strojnog učenja za oblikovanje računalnog igrača na primjeru strateške igre. Radu priloženi izvorni tekstovi programa, dobiveni rezultati uz potrebna objašnjenja i korištena literaturu.



**Ključne riječi:** implementacija, model, strateška igra, igra, umjetna inteligencija, strojno učenje


## Strategy game model implementation


### Abstract

Described multiplayer strategy game model with specific ruleset. Researched machine learning application in producing artificial players and mentioned existing solutions. Implemented a modular strategy game simulation model with interfaces for multiple players. Added match progress tracking and implemented an artificial intelligence opponent. Researched efficiency of machine learning concepts for developing artificial players. Attached source code, machine learning results with commentary and references.



**Keywords:** implementation, model, strategy game, game, artificial intelligence, machine learning