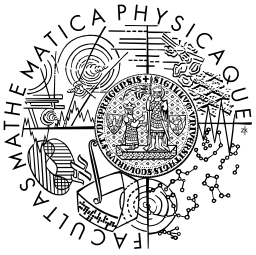


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Vojtěch Sejkora

Bojiště pro virtuální autonomní roboty

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce:	Mgr. Pavel Ježek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Obecná informatika

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Chtěl bych poděkovat vedoucímu své bakalářské práce Mgr. Pavlu Ježkovi, Ph.D. za skvělé vedení mé práce, trpělivost a za čas, který mi věnoval při konzultacích, jakož i při sepisování této práce. Velmi také děkuji Ondrovi a Martině, za kritické přečtení celé práce. Na závěr bych na tomto místě velmi rád poděkoval rodičům a sourozencům za to, že mne po celou dobu studia podporovali a dodávali sil v závěru studii.

Název práce: Bojiště pro virtuální autonomní roboty

Autor: Vojtěch Sejkora

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Programování se ve většině případů vyučuje na matematicky motivovaných příkladech, které nedokáží zaujmout všechny studenty. Proto v minulosti vzniklo mnoho projektů, které se snaží vyučovat programování pomocí her. Cílem této bakalářské práce bylo vyvinout hru, v které by mohli studenti mezi sebou soupeřit v programování virtuálních robotů, a tak si osvojit základy programování. Součástí bakalářské práce je několik typů her, zobrazování průběhu zápasu a mechanismus k přidání překážek na mapu. Dále tato bakalářská práce obsahuje také návod, jak pracovat s tímto projektem a programovat virtuální roboty.

Klíčová slova: programování autonomní roboti virtuální prostředí

Title: Battlefield for virtual autonomou's robots

Author: Vojtěch Sejkora

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: In most cases, programming is taught using mathematically motivated examples, which are not attractive for all students. As a result, there have been many projects in the past that aimed to teach programming through games. A goal of this Bachelor's thesis was to develop a game in which students can compete with each other in programming virtual robots, and thus gain basic programming skills. This Bachelor's thesis contains several types of games, a visualization of the game and a mechanism for adding obstacles to the map. Futhermore, it contains a manual describing how to use this project and develop new virtual robots.

Keywords: programming autonomous robots virtual enviroment

Obsah

1. Úvod

Základní znalost programování je velkou výhodou v mnoha oborech lidského počínání, a proto se dnes základy programování vyučují na většině středních škol. Avšak velkým problémem výuky programování zůstává, že není pro mnoho studentů atraktivní, neboť programování je vyučováno na matematicky motivovaných příkladech. Jedním z takových příkladů je vypsání počtu prvočísel do čísla 10 000. Řešení tohoto úkolu může vypadat například takto:

```
doCisla = nactiCislo();
cislo = 2;
dokud cislo <= doCisla {
  pokud jePrvocislo(cislo) {
    prvocisel = prvocisel + 1
  }
  cislo = cislo + 1;
}
vypis(prvocisel)
```

vstup	*výstyp*
10000	1229

Kód 1. Ukázka programu počítající počet prvočísel do zadaného čísla.

Mezi hlavní výhody takovýchto úloh patří snadná ověřitelnost správných výsledků nebo možnost vysvětlit mnoho programových struktur – podmínky, cykly, apod. A proto většina učebnic programování [1, 2, 3] obsahuje pouze takovýto druh příkladů. Avšak problémem těchto příkladů je, že nedokáží dostatečně oslovit všechny studenty, zejména ty, kteří nemají kladný vztah k matematice. Proto vznikla tato bakalářská práce, která se soustředí na vytvoření programu, pomocí kterého by bylo možné vyučovat programování atraktivnější formou.

1.1 Související práce

V minulosti vzniklo mnoho projektů, které si kladly za cíl vyučovat programování zábavnější formou. Některé z těchto projektů jsou krátce představeny v následující sekci.

Před samotným popisem projektů je nutné uvést rozdíly v přístupu k programování, které jsou u těchto projektů použité.

Procedurální programování je způsob, kdy jsou příkazy vykonávány v pořadí, ve kterém jsou napsány tj. zleva doprava a shora dolů. Tento způsob bývá první, se kterým se programátor setkává. Popsán je v mnoha učebnicích zabývajícími se základy programování [1, 3, 4].

Událostmi řízené programování je způsob, kdy jsou vybrány příkazy, které jsou následně vykonány podle události, která byla vyvolána. V rámci této události se příkazy vykonávají procedurálně. Programátor programuje reakci na danou událost (např. stisknutí konkrétního tlačítka v grafickém rozhraní).

1.1.1 Karel

Pravděpodobně nejznámějším projektem , který učí programovat formou hry je programovací jazyk Karel [5], který umožňuje programovat robota pohybujícího se ve městě sestaveném z navzájem na sebe kolmých ulic. Tento robot rozumí pěti příkazům: `move` (přesuň se na další křižovatku), `turnleft` (otoč se vlevo), `putbeeper` (polož bzučák), `pickbeeper` (zvedni bzučák) a `turnoff` (vypni se). Robot také umí zjistit, jestli je pozice před ním volná, jsou-li v blízkosti bzučáky a také, na kterou stranu je v dané chvíli natočen. Programovací jazyk Karel také obsahuje podmínky, cykly a metody. Výsledkem programu v tomto jazyce je robot, který se pohybuje po městě.

Příkladem programu v programovacím jazyce Karel (kód 2) je robot, který se třikrát otočí doleva, posune se o jedno políčko a nakonec se vypne. Pohyb robota, který se pohybuje na základě tohoto programu, je znázorněn na obr. 1.

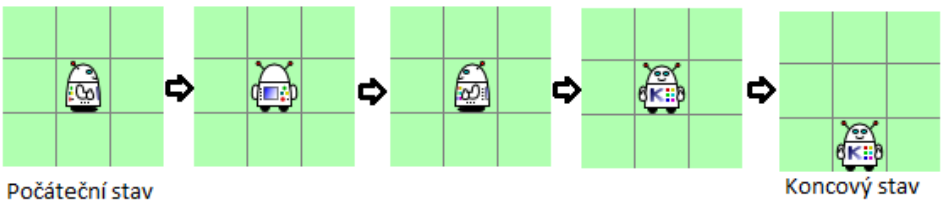
```
BEGINNING-OF-PROGRAM

 BEGINNING-OF-EXECUTION
   ITERATE 3 TIMES
     turnleft

   move
   turnoff
 END-OF-EXECUTION

END-OF-PROGRAM
```

Kód 2. Ukázka programu pro jazyk Karel.



Obrázek 1. Ukázka chování robota v programovacím jazyce Karel.

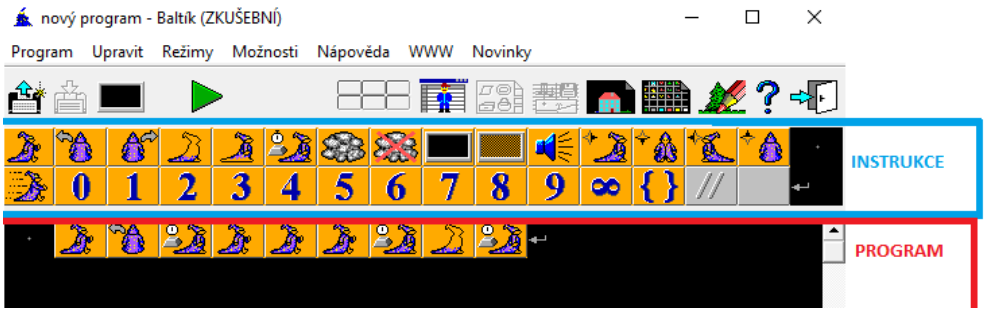
Tento úkol (otočení doprava) je typickým úkolem pro programovací jazyk Karel a může být snadno demonstrován při výuce přímo na studentech. Jeden student se otočí doprava a druhý student se bude moci otáčet jen doleva. Poté, kdy se druhý student otočí třikrát doleva, zjistí, že se dívá na stejným směrem jako první student, který se otočil jednou doprava. Výuka programování se tak stává přehlednou a zábavnou.

Tento programovací jazyk je procedurální.

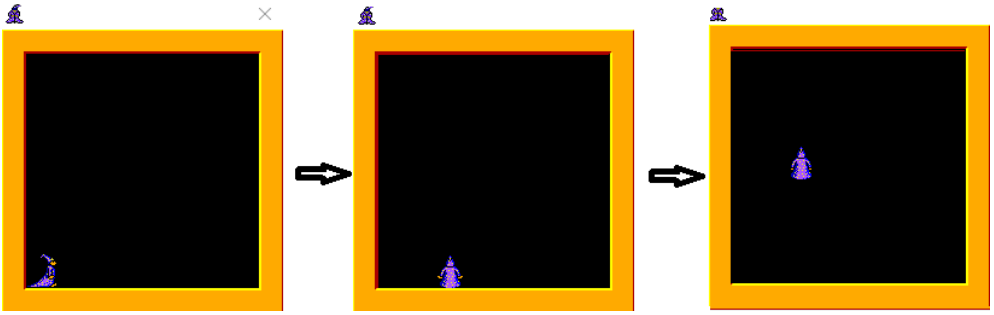
1.1.2 Baltík

Baltík [6] je další projekt (programovací jazyk), který učí programovat zábavnou formou. V tomto programovacím jazyce se programuje pomocí přetahování

instrukcí (na obr. 2 se jedná o modře ohraničenou část) do programu (na obr. 2 červeně ohraničená část). Toto může být v dnešní době výhodou, protože děti běžně používají tablety, tedy i když neumí ještě dobře psát, mohou programovat právě tímto přetahováním instrukcí. Výstup celého programu je poté zobrazen na nové obrazovce, která se spustí zelenou šipkou. Tento výstup (programu Baltík obr. 2) je zachycen na obr. 3. Po spuštění programu se objeví čaroděj, který postoupí o jedno políčko a otočí se doleva. Po stisku libovolné klávesy postoupí o 3 políčka. Po dalším stisku libovolné klávesy čaroděj zmizí a po ještě jednom stisku klávesy se celý program ukončí. Tento programovací jazyk je procedurální.



Obrázek 2. Ukázka programu v jazyce Baltík.

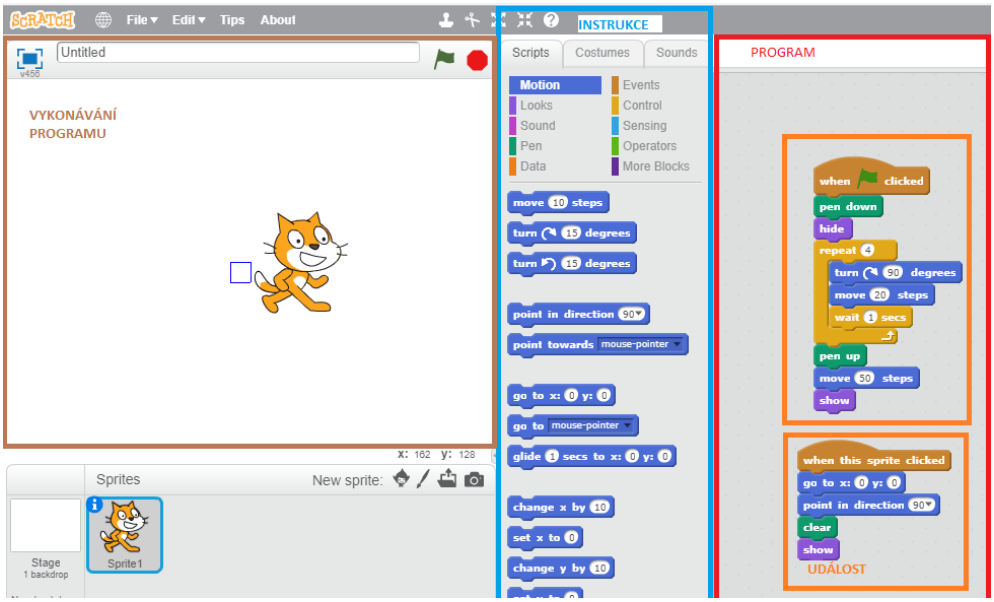


Obrázek 3. Ukázka výstupu programu v jazyce Baltík.

1.1.3 Scratch

Scratch [7] je dalším grafickým programovacím jazykem pro výuku programování. V tomto jazyce se programuje stejným způsobem jako v jazyce Baltík, tj. přetahováním instrukcí (modře zvýrazněná část) do programu (červeně zvýrazněná část) (obr. 4). Hlavní rozdíl oproti programovacímu jazyku Baltík je v systému vyhodnocování kódu – jazyk je událostmi řízený (události jsou na obrázku oranžově ohraničené). Vykonávání programu je vidět v levé části obrazovky (hnědě zvýrazněná část). Program, který je napsaný na obr. 4, při kliknutí na zelenou šipku vykreslí postupně čtverec (hranu po hraně) modrou čarou (událost na obrázku výše). Po vykreslení se vedle čtverce objeví kočka, na kterou lze poklepat (událost na obrázku níže). Po poklepání se kočka přesune do výchozí pozice

uprostřed plochy a vše nakreslené se smaže.



Obrázek 4. Ukázka programu v jazyce Scratch.

Scratch i Baltík jsou pro výuku velmi vhodné, protože je ihned vidět výsledek programu, např. pohyb postavičky. Také se ukazuje, že studenti základních i středních škol jsou schopni v těchto programovacích jazycích vytvářet poměrně složité programy. Například hry, kdy se čaroděj v podobě jezdce pohybuje podle určitých pravidel na stisk klávesy po hracím plánu a zároveň označuje místa, na kterých se už nacházel (což je 2. úloha z celostátního kola soutěže Mladý Programátor [8]).

I přes zmíněná pozitiva, nevýhodou výše zmíněných programovacích jazyků zůstává, že se v nich nedají programovat reálné aplikace, jako Word, e-mailový klient, Excel apod., protože zadávané instrukce ovládají pouze postavičku a její okolí a nepodporují komunikaci s operačním systémem.

1.1.4 Netrobots

Nástrojem, který učí programovat zábavnou formou a zároveň používá jazyk, ve kterém se píší reálné aplikace, je například hra Netrobots [9]. V této hře se programují virtuální autonomní roboti, kteří spolu bojují v aréně (průběh souboje robotů je možné sledovat v grafickém prostředí, viz obr. 5). Úkolem každého robota je zničit všechny protivníky. K tomu slouží příkazy, pomocí kterých je možné ovládat robotův kanón, skener nebo pohyb. Robot nemá jiné příkazy než ty, které ovládají tyto zmíněné části. Pomocí kanónu může robot střílet na další soupeřící roboty. Samotná střelba má i svá vlastní pravidla. Střílet je možné pouze na omezenou vzdálenost, a je také třeba brát v potaz, že střely z kanonu působí zranění pouze v omezeném rozsahu, a proto je nutné střílet velmi přesně. Pomocí skeneru může robot zjistit, jestli se ve sledované oblasti vyskytuje soupeř – skener je tak jediným prostředkem, pomocí kterého je možné určit polohu nepřítele. Pomocí motoru se robot může pohybovat. Zejména se může přiblížit k soupeři,

aby na něj mohl přesněji vystřelit, ale také ho může použít k úhybným manévrům. Programovaní robota v Netrobots se více podobá programování reálných aplikací, protože se programuje v programovacím jazyce C, a tak jediná změna při přechodu na programování reálné aplikace je výměna povelů robota za povely pro operační systém.

Obrázek 5. Ukázka zápasu v Netrobots.

Pomocí tohoto nástroje se programování vyučovalo například na Letním táboře programátorů 2013[1] (LTP)[2], kterého se účastní studenti základních a středních škol.

Všechny výše zmíněné projekty mají společné to, že programování vyučují zábavnou formou – formou hry. Jsou schopny studenta oslovit a zaujmout, a proto jsou vhodným nástrojem pro výuku programování na základních a středních školách.

1.2 Shrnutí

Bakalářská práce je zaměřena na vytvoření programu pro studenty středních a základních škol. U základních škol spíše pro vyšší ročníky. Proto v práci není kladen důraz na programování v grafickém programovacím jazyce (které je vhodné pro mladší studenty), ale spíše na programování psaním textu. Hlavní ideou práce je vyučovat programování tak, aby byl následný přechod k programování reálných aplikací co nejsnazší. Z tohoto důvodu byla také vybrána hra Netrobots. Budou odstraněny její hlavní nedostatky, které se objevily při použití v praxi, a také budou navržena další rozšíření hry. Proto se následující kapitola věnuje rozboru hry Netrobots a vysvětluje, které nedostatky byly ve hře objeveny. V závěru následující kapitoly jsou vypsány cíle této bakalářské práce.

[1] `http://protab.cz/`

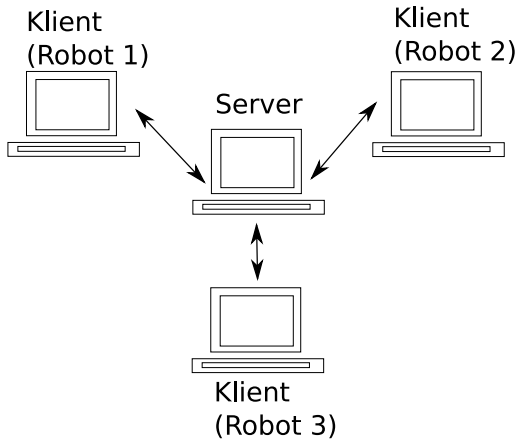
[2] Na LTP šlo programovat též v Pythonu, ale Netrobots neobsahuje pro Python knihovny.

2. Rozbor hry Netrobots a stanovení cílů práce

V této kapitole je nejprve popsán návrh hry Netrobots, zejména se zaměřením na architekturu aplikace, vyhodnocování akcí robotů a způsob programování robotů. Následně jsou popsána možná rozšíření původní verze hry.

2.1 Architektura

Jak již bylo zmíněno výše, hra Netrobots slouží k výuce programování za pomoci programování virtuálních autonomních robotů, kteří mezi sebou soupeří na virtuálním bojišti - aréně. Aby mezi sebou mohli jednotliví roboti soupeřit, je nutné, aby aplikace uměla načíst a vyhodnotit programy pro tyto různé roboty. Variant, jak tuto situaci řešit, je více a jsou popsány při návrhu vlastní architektury (3.2). V rámci Netrobots je aplikace rozdělena na dvě části – na server a klient. Server se stará o aktualizaci stavu hry a zasílání klientovi odpovědí na jeho příkazy a klient se stará o ovládání robota zasíláním příkazů. Způsob, jakým spolu tyto části komunikují, je ukázán na obr. 6, kde jsou zobrazeni tři klienti (každý klient ovládá jednoho robota), kteří se připojují k serveru.



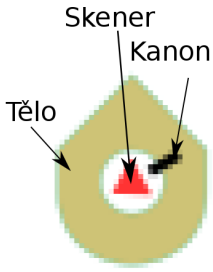
Obrázek 6. Ukázka propojení serveru s klienty.

2.2 Systém zobrazování

Důležitou součástí Netrobots je grafické prostředí, ve kterém probíhá animace zápasu a kde je také vidět, jaké příkazy naprogramovaný robot vykonává. V této části práce je ukázáno, co se zobrazuje během zápasu.

Nejprve si popíšeme z čeho se skládá robot. Ukázka, jak robot vypadá, je na obr. 7. Červený trojúhelník je skener, který se natáčí podle toho, kterým směrem se skenuje. Černá část je kanon a od středu vzdálenější konec určuje natočení kanonu. Zbytek robota je tělo, kde špička tohoto útvaru směřuje do směru, kterým

je robot natočen pro pohyb.



Obrázek 7. Popis, z čeho se skládá robot.

V ukázce zápasu(obr. 8) má *Robot 2* skener ve výchozí pozici natočený na 3. hodinu - 0°. *Robot 2* má kanon ve výchozí pozici a *Robot 1* ho má natočený přibližně na 330°. Zdraví robotů je vidět v pravém horním rohu - čím více je obdélník zelený, tím více životů roboti mají. *Střela* je černý kruh, který se pohybuje od robota, který jej vystřelil, do místa, kde zasahuje. *Výbuch* je vizualizace dopadu střely. Tato vizualizace neodpovídá pásu zranění robota, ten je o kousek menší. Také obrázek robota neodpovídá přesně jeho velikosti. Velikost robota je ve hře jeden bod (což na obrázku odpovídá středu skeneru). Robot je tedy zasažen až tehdy, když vizualizace výstřelu přesáhne celý skener.

Obrázek 8. Vizualizace zápasu.

Z výše zmíněných důvodů je zřejmé, že nelze dobře poznat, kdy je robot zasažen. Také není možné určit, který robot je který a také, které životy patří kterému robotu. Nikde nejsou nastavena nebo zobrazena jména robotů nebo hráčů.

Tyto dvě chyby je pravděpodobně možné opravit správnou konfigurací vykreslující knihovny, nicméně ani k této části není zpracována dokumentace, tedy si ji student těžko nastaví (samotná instalace této knihovny je poměrně složitá, protože je nutné instalovat poměrně mnoho balíčků a návod opět není k dispozici). Další nezjistitelná informace je, kterým směrem robot skenuje, protože trojúhelník zobrazující skener se zdá být rovnostranný a žádný vrchol, který by mohl ukazovat na směr skenování nevyčnívá. Poměrně často se sice stává, že směrem, kterým skener skenuje, se později natočí i kanon, ale není to přesné. Vizualizaci nelze v průběhu hry zastavit, protože probíhá na serveru a tedy bychom při ustavičném skenování stejně nepoznali, kam přesně robot skenuje. Jedním z možných řešení by byl vlastní program, který by vykresloval stav arény. Toto řešení bylo například použito při výuce programování na LTP.

2.3 Příkazy robotů

V původní verzi hry má robot pouze malé množství příkazů (podobně jako programovací jazyk Karel viz 1.1.1). Příkazy jsou rozděleny do dvou kategorii - zjišťující a bojové. Za jeden tah lze poslat pouze jeden příkaz.

2.3.1 Zjišťující příkazy

Pomocí zjišťujících příkazů je možné získat informace o stavu robota - o jeho poloze, zdraví a cyklu, ve kterém se nachází. Těmito příkazy jsou:

`loc_x` – zjišťuje horizontální pozici robota.

Vyvolává se zavoláním metody `int loc_x()` a vrací horizontální pozici robota (od 0 - zcela vlevo, do 1000 - zcela vpravo).

`loc_y` – zjišťuje vertikální pozici robota.

Vyvolává se zavoláním metody `int loc_y()` a vrací vertikální pozici robota (od 0 - zcela nahoře, do 1000 - zcela dole).

`cycle` – zjišťuje pořadové číslo aktuálního kola.

Vyvolává se zavoláním metody `int cycle()` a vrací pořadové číslo kola.

`damage` – zjišťuje poškození robota.

Vyvolává se zavoláním metody `int damage()` a vrací procentuální poškození (v rozmezí od 0 - nepoškozen, do 100 - zničen).

2.3.2 Bojové příkazy

Pomocí bojových příkazů je robot ovládán. Pomocí nich se pohybuje, zjišťuje polohu ostatních soupeřů a útočí na ně. Těmito příkazy jsou:

`drive` – je příkaz k ovládání pohybu.

Vyvolává se zavoláním metody `int drive(int degree, int speed)`, jejíž parametry mají význam:

`degree` – určení směru pohybu ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0° (přijímá hodnoty od 0 do 359). Směr je možné změnit pouze při dostatečně nízké rychlosti.

`speed` – požadovaná rychlost v procentech (při 100% ujede robot vzdálenost 4 jednotek za kolo)

Návratová hodnota je 1 pokud se povedlo změnit směr jízdy, jinak 0

`scan` – je příkaz k ovládání skeneru.

Vyvolává se zavoláním metody `int scan(int degree,int resolution)`, jejíž parametry mají význam:

`degree` – směr skenování ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0° (přijímá hodnoty od 0 do 359).

`resolution` – je parametr (rozšíření) výseče, ve které se skenuje. Udáván je ve °, (co to znamená je názorně zobrazeno na obrázku 9) (povolené hodnoty jsou od 0 do 10).

Návratová hodnota je 1, pokud se povedlo změnit směr jízdy, jinak 0.

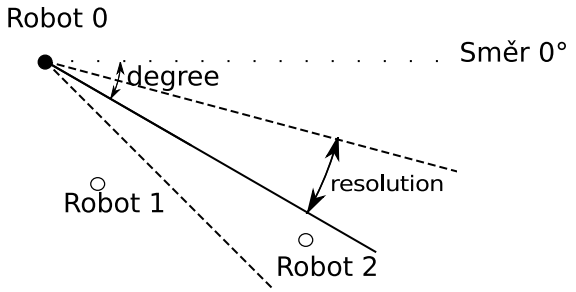
`cannon` – je příkaz k ovládání kanonu.

Vyvolává se zavoláním metody `int cannon (int degree,int range)`, jejíž parametry mají význam:

`degree` – směr natočení kanonu ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0° (přijímá hodnoty od 0 do 359).

`range` – je vzdálenost, do které se vystřelí (povolené hodnoty jsou od 0 do 700, nelze tedy střílet přes celou herní mapu, která je velká 1000×1000).

Návratová hodnota je 1, pokud se povedlo vystřelit (robot má dvě hlavně, ze kterých může střílet až po nabití, které trvá v kódu definovaný počet kol), jinak 0.



Obrázek 9. Ukázka fungování skeneru v Netrobots. Robot 0 skenuje. Robot 2 je vidět. Robot 1 není vidět. Aby Robot 0 viděl Robota 1, musel by změnit úhel skeneru, nebo zvýšit jeho rozlišení.

2.4 Způsob vyhodnocování příkazů

V Netrobots vyhodnocuje akce server a zasílá odpovědi zpět klientovi. V původní verzi hry má robot dva druhy příkazů (jak bylo popsáno v 2.3).

V každém tahu Netrobots je vyhodnocován:

- Pohyb a výbuch střel
- Aktualizace zobrazení
- Čekání 10ms (od startu aktualizování zobrazení)
- Příjem příkazů robotů
- Odeslání odpovědí na příkazy
- Pohyb robotů

Poslání polohy robotů na začátku tahu, i když je také uvedeno v dokumentaci u popisu průběh tahu, ve skutečnosti neprobíhá a je nutné si o ni zažádat příkazem `loc_x`, `loc_y`. Ale jak již bylo zmíněno, přijímá se pouze jeden příkaz za kolo, což znamená, že pokud se robot pohybuje pod jiným úhlem než $0°, 90°, 180°, 270°$, není možné zjistit jeho přesnou polohu (vždy jedna ze souřadnic bude neaktuální). Navíc, poloha robota je posílána se zpožděním jednoho kola (nejprve se odešle odpověď na příkaz a až poté se pohybují roboti). Důležité je též to, že tah trvá 10 ms. Proto na pomalé síti není možné zjistit aktuální stav robota. Z dokumentace by se však mohlo zdát, že se čeká na příjem příkazů od všech robotů. Motivací pravděpodobně bylo zobrazování, které probíhá během zápasu, a také pevný frame rate.

V praxi studenti používající Netrobots chtěli psát programy pro více robotů. To však komunikační knihovna nepodporuje. Pro programování více robotů je tedy nutné spouštět více procesů a programování komunikace mezi nimi je velmi obtížné (je nutné používat sdílenou paměť). Důvodem je, že knihovna po odeslání příkazu čeká na odpověď a pozastaví vykonávání programu, dokud tuto odpověď nepřijme. Více vláken nepomůže, protože to návrh knihovny nepodporuje. [1]

2.5 Způsob programování robotů

Robot je programován před spuštění hry v programovacím jazyce C s využitím dodané knihovny zajišťující komunikaci se serverem. Algoritmus, který bude popsán, je poměrně jednoduchý, ale přesto porážel první verze robotů naprogramované účastníky LTP. Pro snadnější pochopení je nejprve popsán algoritmus robota pomocí pseudokódu 3.

[1] Knihovna obsahuje hlavní metodu (metodu `main`), která je spuštěna při samotném spuštění programu. Metodu `main`, kterou uživatel naprogramuje, knihovna přejmenuje na `rmain` metodu a ta byla volána z původní `main` metody knihovny.

```
  smer = dolu;
  repeat forever {
    if ( stoji() ) {
      jet(smer, plny_plyn);
    }
    if (blizko_zdi_ve_smeru_jizdy()) {
       brzdi();
       smer = nasledujici_smer();
    } else {
       for (stupen od 0 do 360 s krokem 30) {
         if (je_robot_ve_smeru(stupen)) { vystrel_po_robotu(); }
       }
    }
  }
```

Kód 3. Algoritmus robota Spot v pseudokodu.

Robot se prvně vydá směrem dolů, dokud se neocitne blízko dolní zdi, pak pokračuje k levé zdi, dokud není blízko. Pak k horní a nakonec k pravé. A poté znovu dolů. Při jízdě průběžně skenuje s maximální `resolution` do dvanácti směrů. Pokud v daném směru uvidí soupeřícího robota, vystřelí na místo, které je vzdálené stejně jako byl tento robot, a ve směru hlavního skenovacího paprsku (`degree`). Algoritmus tohoto robota v Netrobots by se zapsal způsobem, jaký je vidět v ukázce kódu 4.

```
#include "robots.h"
main () {
  int range, direction = 90;
  while (1)  {
    if (speed () == 0)
      drive (direction, 100);
    if ((direction == 90 && loc_y () > 575)
          || (direction == 270 && loc_y () < 425)
          || (direction == 180 && loc_x () < 150)
          || (direction == 0 && loc_x () > 850)) {
      drive (direction, 0);
      direction = (direction + 90) % 360;
    } else {
      for (int degree = 0; degree < 360; degree +=30) {
        if (range = scan(degree, 10)) {
          cannon(degree, range);
        }
      }
    }
}
```

Kód 4. Algoritmus robota spot. zapsán v syntaxi Netrobots.

2.6 Možná rozšíření

V původní verzi Netrobots je pouze jedna varianta hry, v rámci níž mají roboti za úkol učinit co největší poškození všem ostatním robotům. Prostor, ve kterém se bojuje neobsahuje žádné překážky a všichni roboti se ovládají samostatně – není je možné seskupit do týmů. Na základě těchto poznatků a zpětné vazby od účastníků LTP, kteří chtěli rozšíření, aby se mohli dále zlepšovat v programování za pomocí projektu, který již poměrně dobře znají, jsme se rozhodli rozšířit původní verzi o tyto vlastnosti:

2.6.1 Překážky na mapě

Překážky na mapě je rozšíření, které umožňuje přidání stěn, antiradarových polí (jimiž neprojde skenovací signál), písku (kde robot jede pomaleji) a dalších. Pomocí tohoto módu lze vystavět terén, či celé místnosti a celá hra tím dostává nový rozměr. Zásadně se tak mění styl ovládání robota, tedy poměrně spolehlivě se mění obtížnost.

2.6.2 Herní varianty

Deadmatch (vybíjená) – je varianta hry, která je dostupná i v původní verzi Netrobots a spočívá v tom, že se hráč snaží vyhrát tím, že pomocí střelby vyřadí ostatní soupeře. Tento typ hry byl v naší implementaci uchován, protože je poměrně snadný na vymyšlení chování robota.

Capture the flag (zabírání vlajky) – je varianta hry, kde je zapotřebí donést soupeřovu vlajku do svého zázemí, což je následně ohodnoceno body.

Capture the base (zabírání strategických míst) – je varianta hry, kde jsou zabírány předem oznámené pozice. Zabírání probíhá tak, že se hráč nachází na dané pozici a za každou jednotku času, kterou tam je, získá určitou část pozice (tedy několik procent). Místo je hráčem zabráno ve chvíli, kdy hodnota dosáhne 100%. Čím více jednotek od stejného týmu je na dané pozici, tím rychleji je pozice zabírána. Soupeř může zabírání místa překazit tím, že se také nachází na dané pozici, což ovšem také zvyšuje možnost, že původního robota zničí střelbou.

Zde je důležité zmínit, že díky existenci různých herních variant, je student nucen přistupovat k programování robotů flexibilně a reagovat tak na danou herní variantu. Robot může být tedy naprogramován univerzálně, aby zvládal více druhů hry, nebo naopak velmi konkrétně jen pro danou variantu hry.

2.6.3 Vybavení

Velmi zajímavým aspektem hry může být možnost nakupování výkonnějšího vybavení:

- motoru – pomocí kterého se lze rychleji pohybovat po hracím poli a také zatáčet ve vyšší rychlosti,
- pancíře – který zvyšuje maximální počet životů,
- skeneru – pomocí kterého lze zvýšit jeho `resolution`, tedy skenovat ve větší výseči,
- vybavení pro útok – což je dělo, které může mít různý počet střel, různý dostřel a způsobit soupeři různě velká zranění.

2.6.4 Týmy

Komunikační knihovna Netrobots umí pracovat vždy pouze s jedním robotem (2.4). Ze zkušenosti s projektem ale víme, že uživatelé chtějí programovat týmy. V Netrobots je nutné k tomu využít více procesů a ke komunikaci mezi nimi sdílenou paměť, což je ale hodně pokročilý koncept. Proto jsme se rozhodli umožnit programovat týmy jednodušším způsobem.

2.6.5 Povolání

Po zavedení týmů byl logickým krokem mód, který podporuje více povolání robotů. Takovým povoláním může být například tank střílející střely, tank pokládající miny, tak opravující ostatní roboty a další. Každý robot v týmu může mít jiné povolání. Navíc určitá povolání jsou vhodná pro konkrétní druh mapy případně mód hry (např. minér není příliš užitečný v deadmatch variantě hry, ale naopak v capture the base bude mít tento robot velkou výhodu.)

Možných rozšíření, které lze použít a naprogramovat, je celá řada. V rámci této bakalářské práce jsme se soustředili na některé z nich, a to převážně na ty, které výrazně mění nároky na ovládání robota.

2.7 Proč reimplementace Netrobots

Při vývoji programu bylo nutné rozhodnout, jestli zvolíme reimplementaci celého projektu od začátku, nebo použijeme již existující Netrobots a ten pouze upravíme. Netrobots je pouze komunikační knihovna, zobrazování a server. Komunikační knihovnu by bylo nutné přeprogramovat, aby podporovala programování více robotů a aby umožňovala zjistit kompletní informace o stavu robota. Zobrazování by mohlo být zachováno, zde by bylo nutné jen naprogramovat lepší zobrazování skeneru. Ale server by měl být přeprogramován, aby podporoval více druhů hry, módy apod., což je velký zásah. A zobrazování je přímo závislé na serveru.

Z důvodů velkých zásahů do kódu Netrobots jsme se rozhodli celý projekt naprogramovat od začátku, protože z původního projektu Netrobots by toho příliš nezbylo. Navíc nám tato volba umožní použít jiný programovací jazyk než je C, například nějaký objektově orientovaný, kde je přístupný polymorfizmus a další techniky umožňující snadnější naprogramování.

2.8 Cíle práce

V původním návrhu Netrobots se objevily následující chyby:

- Nemožnost jednoduše programovat více robotů a nemožnost zjistit přesnou pozici (proč tomu tak v Netrobots je je rozebráno v kapitole 2.4)
- Nejednoznačnost zobrazení průběhu zápasu (konkrétně se tím zabývá kapitola 2.2)

Reimplementace Netrobots byla navržena s cílem opravit tyto chyby a navíc přidat rozšíření, která umožní dále s projektem pracovat. Zvláštní důraz je kladen na implementaci rozšíření, které mění požadavky na chování robota.

1. Reimplementace Netrobots
   (a) Architektura
      i. Naprogramovat simulaci (to je to, co se děje v Netrobots na serveru).
      ii. Umožnit snadnou komunikaci s prostředím (to zajišťuje v Netrobots komunikační knihovna) a zamezit potenciálním hrozbám, kterými jsou v Netrobots:
         A. Umožnit zjištění přesné polohy robota.
         B. Umožnit programovat více robotů a snadnou komunikaci mezi nimi za pomocí dodané knihovny. (Kvůli rozšíření týmů).
   (b) Systém vyhodnocování akcí robotů
      i. Definovat jak jsou akce vyhodnocovány (mohou být vyhodnocovány tahově - jako v Netrobots, nebo v reálném čase).
      ii. Přesně definovat systém vyhodnocování akcí robotů (včetně pořadí).
   (c) Vizualizace
      i. Vyřešit problém s rozpoznáváním kterým směrem se skenuje.
      ii. Rozmyslet způsob zobrazování [během utkání vs. po utkání a kde se bude zobrazovat (např. u klienta, na serveru) - což závisí též na zvolené architektuře].
   (d) Způsob programování robotů
      i. Rozmyslet způsob programování robotů (procedurálně vs. událostmi řízené).
      ii. Popsat dostupné příkazy.
      iii. Popsat jak se programují týmy a jak jeden robot.
2. Implementace módů
   (a) Vybavení
   (b) Týmy
   (c) Překážky na mapě
   (d) Povolání
   (e) Typy her

3. Analýza

V této kapitole jsou podrobně rozepsány jednotlivé cíle a návrh jejich řešení a nakonec také vyhodnocení, které řešení se jeví jako nejlepší. Dále jsou zvoleny nástroje, které jsou pro implementaci těchto řešení nezbytné.

3.1 Způsob vyhodnocování příkazů robotů

Příkazy lze vyhodnocovat a simulovat tahově (postupně), nebo v reálném čase. V následujících kapitolách jsou obě tyto možnosti rozebrány.

3.1.1 V reálném čase

Při vyhodnocování v reálném čase je důraz kladen na komunikaci se serverem a na rychlé výpočty. Výpočty tedy musí být naprogramovány efektivně, aby mohly být provedeny dostatečně rychle. Což může být pro začínající programátory problém. Dalšími technickými problémy tohoto způsobu vyhodnocování příkazů jsou: latence sítě a garbage collector[1]

3.1.2 Tahově

Oproti vyhodnocování příkazů v reálném čase, tahové vyhodnocování, kdy se čeká až všichni roboti pošlou své akce, dává větší prostor pro výpočty, což nemusí být výhodné, protože ostatní roboti musí čekat na dokončení výpočtu jiného robota. Může se tedy stát, že jeden robot, jehož výpočet se ocitne v nekonečném cyklu, zablokuje celou hru.

3.1.3 Zhodnocení

Vyhodnocování v reálném čase s sebou nese vysokou náročnost na programování. Tahové vyhodnocování má nedostatek pouze v možném uvíznutí hry. Z těchto důvodů byl zvolen kompromis mezi předchozími dvěma přístupy (podobně jako je tato problematika zpracována v Netrobots). A to tak, že vyhodnocování probíhá v tazích, ale každý tah má určenou maximální dobu trvání. Tato varianta umožňuje programovat roboty i méně zkušeným programátorům, protože času na výpočty pro dané kolo je dostatek, ale také hra může pokračovat, pokud dojde k zacyklení některého z robotů, což lze u začínajících programátorů očekávat.

Protože je na tah omezený čas, není tato hra tahová, ale je vyhodnocována v reálném čase, avšak vhodnou volbou maximální doby na tah lze zamezit problémům s garbage collection a snížit dopad latence sítě. Domníváme se, že doba 100 ms na tah je dostatečná k snížení dopadu latence sítě na minimum. Robot oznamuje, že jeho tah skončil tehdy, když odesílá svou akci. Díky tomu server může zpracovat všechny akce robotů najednou. Druhou možností je, že robot dostane odpověď na svou akci ihned. V případě týmů by to ovšem znamenalo, že

[1] U klienta se spustí garbage collection. Ten pak nemůže se serverem komunikovat a následně může být stav robota u klienta velmi odlišný od stavu tohoto robota na serveru. Protože zatímco garbage collector u klienta uklízel, na serveru dál probíhaly výpočty.

ostatní roboti mohou ve stejném tahu reagovat podle toho, jakou odpověď dostal první robot a to nám nepřijde správné.

Přesně definované vyhodnocení systémů akcí je popsáno v kapitole 5.5.

3.2 Architektura

Možnými řešeními architektury, která je použita k načítání a vyhodnocování programů robotů, je lokální aplikace (načítá algoritmy robotů z lokálních souborů) nebo architektura server-klient (algoritmus běží odděleně na jiném stroji, než simulace). Obě tyto varianty jsou podrobně rozebrány v následujících kapitolách.

3.2.1 Lokální aplikace

Provedení implementace hry, jako lokální aplikace, která dokáže nahrát algoritmy robotů z lokálních souborů.

Zjištění přesného stavu robotem lze vyřešit tím, že knihovna po každém tahu aktualizuje všechna dostupná data (tj. pozici, životy, číslo tahu, apod.).

Programování více robotů. Roboti se dozvídají odpovědi ihned od systému provádějící simulaci. Tyto odpovědi tedy mohou běžet v jednom vlákně, protože není nutné čekat na proběhlou komunikaci. Knihovna případně implementuje synchronizaci robotů (knihovna může shromáždit informace o příkazech pro všechny roboty a počkat na jejich vyhodnocení pomocí běžných konstrukcí pro vícevláknové synchronizování).

I přes svou jednoduchost má tato varianta řešení několik nevýhod. Hlavní nevýhodou je, že hru lze hrát pouze lokálně, a proto není možné soupeření studentů mezi sebou navzájem. Bylo by to možné jedině pokud by studenti sdíleli svůj kód. Sdílení kódu považujeme za velký problém.

Pokud by byla autorita, které by studenti byli ochotni sdělit kód, pak by bylo poměrně bezpečně souboje provádět. Nicméně nejčastější způsob provádění soubojů předpokládáme mezi studenty. Při tomto způsobu řešení musí učitel souboje spustit, anebo je nutné mít server, k tomuto určený, do kterého se nahrají algoritmy a ten souboje následně spustí.

Další nevýhodou tohoto řešení je, že prostředí, které simuluje průběh hry, a algoritmy robotů jsou spouštěny ve stejném programu. Tedy pokud by byl některý ze studentů velmi nadaný a dokázal využít děr v systému, mohl by získávat informace o robotech, které jsou jinak nezjistitelné.

3.2.2 Server-klient

Následující kapitola se věnuje provedení implementace hry, kdy algoritmus robota běží na jiném zařízení (klient) než na kterém je vykonávána simulace (server).

Výhodou tohoto řešení je, že lze soupeřit aniž by bylo nutné kód sdílet. Další výhodou je, že program algoritmu je spuštěn odděleně od programu vykonávající

simulaci, a proto nelze podvádět, protože tyto dva programy spolu komunikují předem stanoveným protokolem a nelze z nich dostat více informací, než povoluje protokol.

Proto je nutné stanovit pravidla, jakým způsobem spolu budou server a klient komunikovat. Protože chceme umožnit studentům programovat co nejjednodušeji, je nutné dodat komunikační knihovnu, kterou může student při programování algoritmu využívat a tato knihovna pak odesílá příkazy na server a zpracovává získané odpovědi a předává je dál do algoritmu.

Další výhodou této architektury je snadná změna programovacího jazyka. Stačí pouze naprogramovat komunikační knihovnu pro tento jazyk a serverová část může zůstat stejná.

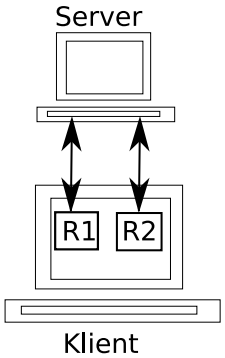
Zjištění aktuálního stavu robota lze vyřešit více způsoby:

- Prvním je povolit zjišťující příkazy posílat kdykoliv. Tento přístup má však nevýhodu v možném zasycení sítě, případně i zahlcení serveru.
- Druhým je zrušit zjišťující příkazy (stav robota se v rámci jednoho tahu nemění) a posílat informace o stavu robota na konci tahu.

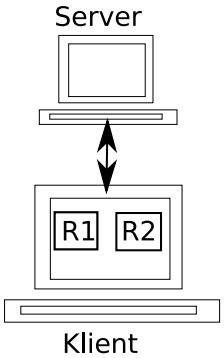
Programování více robotů lze provést dvěma způsoby.

- První možností je mít pro každého robota vlastní komunikační kanál a odesláním příkazu nezablokovat běh aplikace. Je také nutné mít dostupný příkaz, který počká na příjem zprávy pro všechny roboty. Tento způsob je nakreslen na obr. 10. Jak v takovém případě probíhá ovládání robotů je ukázáno na příkladu kódu 5. V tomto příkladě se každý robot připojí k serveru zvlášť, odešle příkaz a pak se čeká na odpověď. Čekání je nutné, aby aplikace nepokračovala bez aktualizovaných informací o stavu robotů, případně bez odpovědi na některou z provedených akcí.

  Tato varianta má tu výhodu, že server nemusí řešit, kteří roboti jsou v kterém týmu a to usnadňuje implementaci komunikace na straně serveru.



Obrázek 10. Ukázka architektury, kde má každý robot vlastní komunikační kanál.



Obrázek 11. Ukázka architektury, kde je jeden společný komunikační kanál pro všechny roboty.

- Druhou možností je použít jeden komunikační kanál, do nějž každý robot připraví příkaz. Příkazy jsou odeslány hromadně, až mají příkaz připravený všichni roboti. Teto způsob je nakreslen na obr. 11 a způsob ovládání robotů je ukázán na příkladu kódu 6. V tomto případě tedy proběhne pouze jedno připojení, poté si každý robot připraví příkaz a když jsou příkazy připraveny od všech robotů, tak jsou odeslány a čeká se na jejich vyhodnocení. O toto čekání se postará komunikační knihovna. Teprve poté kód pokračuje dál.

  Nevýhodou je, že server musí být schopen získat příkazy a přiřadit je správně k jednotlivým robotům. Další nevýhodou je, že nelze napsat ovládání robotů v jednom týmu z více pracovních stanic o což byl v praxi zájem.

```
odpoved1 = Robot1.Scan // robot jedna odeslal příkaz
odpoved2 = Robot2.Drive // robot dva odeslal příkaz
pockej(odpoved1, odpoved2) // počká až roboti dostanou odpověd

// zde lze pracovat s odpovědmi
```

Kód 5. Ukázka použití kódu pro komunikační kanál pro každého robota.

```
odpoved1 = Robot1.Scan // robot jedna připravil příkaz
odpoved2 = Robot2.Drive // robot dva připravil příkaz
// příkazy se odešlou - uživatel se o to nemusí starat

// zde lze pracovat s odpovědmi příkazů
```

Kód 6. Ukázka použití kódu pro jeden komunikační kanál.

3.2.3 Zhodnocení

Výhody architektury server-klient (popsané v 3.2.2) jsou významné, proto jsme se rozhodli zvolit tuto architekturu.

Protože jsme se rozhodli pro architekturu server-klient, musíme rozhodnout, kterým způsobem budou aktualizovány stavy robotů. Rozhodli jsme se pro druhý způsob (viz odstavec Zjištění aktuálního stavu robota v 3.2.2), protože ten nepřináší žádná nebezpečí (za předpokladu požadavků popsaných v závěru 3.1.3) a zamezuje nebezpečím, který přinášel první způsob..

Programování algoritmu robotů v obou variantách (viz odstavec Programování více robotů 3.2.2) je téměř totožné (v první variantě se musí explicitně uvést doba čekání, ve druhé se děje implicitně), ale pro programování serveru je jednodušší, když je pro každého robota vlastní komunikační kanál, a proto byla zvolena tato varianta.

3.3 Zobrazování průběhu zápasu

Důležitým prvkem projektu je vizualizace, pomocí které je možné sledovat, co se při zápasu děje. Způsobů zobrazování může být více:

3.3.1 Zobrazování během zápasu

Pokud se průběh zobrazuje rovnou během zápasu (on-line zobrazování), pak lze pozorovat změny ve vnitřním stavu robota pomocí zápisů, zároveň se sledováním průběhu hry. Avšak při rychlosti 1 tah za 100 ms se zápisy střídají tak rychle, že je lidské oko nedokáže zaznamenat, natož přečíst.

3.3.2 Zobrazování po zápase

Zobrazování po zápase (off-line zobrazování) má výhodu v tom, že lze shlédnout průběh zápasu kdykoliv. Navíc po zápase se může i klient dozvědět, kde byl který robot a v kterém tahu, tedy zobrazování může probíhat i u klienta. Pokud by on-line zobrazování probíhalo přímo u klienta, pak by se mohl zobrazovat pouze naprogramovaný robot (jinak by se dalo využít pozice cizích robotů získané ze zobrazování). Další výhodou off-line zobrazování je, že může jít krokovat tah od tahu, což může pomoci odhalit chybu v algoritmu robota.

3.3.3 Zhodnocení

Implementačně jsou obě varianty zobrazování (jak on-line tak off-line zobrazování) velmi podobné. Pro uživatele má více výhod off-line zobrazování a z tohoto důvodu bylo toto zobrazování implementováno.

3.4 Způsob programování robotů

Roboti v Netrobots jsou programováni procedurálně (popis, co znamená procedurální programování je na začátku kapitoly 1.1), což je způsob programování používaný také v jazyce Karel a Baltík (viz 1.1.1 a 1.1.2). Nicméně pro člověka je přirozené rozhodovat se na základě událostí, proto se domníváme, že pro studenty je mnohem zajímavější ovládat robota pomocí programování událostí (popis, co znamená procedurální programování je na začátku kapitoly 1.1), podobně jako se programuje v jazyce Scratch (viz 1.1.3).

V obou ukázkách je využita syntax příkazů z Netrobots (popsaná 2.3). Algoritmus robota je v obou případech shodný s algoritmem robota spot (ten byl popsán ve druhém odstavci kapitoly 2.5).

3.4.1 Událostmi řízené programování

Při událostmi řízeném programování jsou události předdefinované. Proto programátor přemýšlí nad chováním robota v konkrétní situaci (události). V nastavení knihovny je nutné nastavit parametry pro události (např. kdy už je blízko ke zdi) a to je vše. Ukázka programování robota při použití událostí je v ukázce kódu 7.

```
vidi_robota(uhelSkeneru, resolutionSkeneru, vzdalenost) {
    cannon(uhelSkeneru, vzdalenost)
}

moje metoda zmen_smer(){
```

```
     if (predchozi_smer != smer) {
         if(speed() == 0){
           drive(smer, 100);
          } else {
            drive(smer, 0);
         }
      predchozi_smer = smer
    }
}

blizko_ke_zdi_nahore() {
   smer = 0;
   zmen_smer()
}

blizko_ke_zdi_vpravo() {
   smer = 90;
   zmen_smer()
}

blizko_ke_zdi_dole() {
   smer = 180;
   zmen_smer()
}

blizko_ke_zdi_vlevo() {
  smer = 270;
  zmen_smer()
}

start() {
   drive(90, 100);
   while (1) { // stále dokola opakuj skenování
     for (int uhel = 0; uhel < 360; uhel += 30) {
       scan(uhel,10);
     }
   }
}
```

Kód 7. Ukázka kódu pro událostmi řízený způsob programování.

3.4.2 Procedurální programování

Při procedurálním programování je nutné řešit, jaké situace mohou nastat a také v jakém pořadí (podle důležitosti) budou vyhodnocovány. Také je nezbytné vyřešit jak rozpoznat, že robot se nachází v dané situaci. Pokud tento postup porovnáme s postupem událostmi řízeným programováním je tento způsob složitější. Je totiž nutné vymyslet situace, které mohou nastat a jak rozpoznat, že tyto situace nastaly. Ukázka, jak programovat roboty procedurálně, je v kódu 8.

```
moje metoda zmen_smer(){
  if (predchozi smer != smer) {
    if(speed() == 0){
      drive(smer, 100);
    } else {
      drive(smer, 0);
    }
    predchozi_smer = smer
  }
}

hlavni() {
  smer = 90;

  while(1) { // stále dokola opakuj
    if (speed() == 0){
      drive(direction, 100);
    }
    if (blizko_ke_zdi_dole()) {
      smer = 180;
      zmen_smer();
    }
    if (blizko_ke_zdi_vlevo()) {
      smer = 270;
      zmen_smer();
    }
    if (blizko_ke_zdi_nahore()) {
      smer = 0;
      zmen_smer();
    }
    if (blizko_ke_zdi_vpravo()) {
      smer = 90;
      zmen_smer();
    }
   for (uhel = 0; uhel < 360; uhel += 30) {
     if (vzdalenost =   scan(uhel, 30) {
       cannon(uhel, vzdalenost);
      }
    }
  }
}
```

Kód 8. Ukázka kódu pro procedurální způsob programování.

3.4.3 Zhodnocení

Obě výše zmíněné ukázky kódů jsou dosti podobné, ale způsob přemýšlení, který se za nimi skrývá, je velmi odlišný, jak je také patrné z předchozích dvou podsekcí. Důležité je také zmínit, že programovat jednoho robota je snazší u přístupu událostmi řízeného programování.

Situace je odlišná, pokud není programován jen jeden robot, ale například tři. V tomto případě by bylo nutné vytvořit události typu, *robot A* vidí *robota D* a *robot B* je k *robotu D* nejblíže a *robot C* chce kolem *robota D* projet pro vlajku. Tato situace je poměrně komplikovaná a je možné vymyslet komplikovanější situace, jako třeba *robot A* vidí *robota D*, *robot B* vidí *robota E*, ale *robot A* má málo životů a *robot D* je blízko k *robotu A*, proto je nutné zničit *robota D*. Kvůli této komplikovanosti nelze po autorovi knihovny chtít, aby vymyslel všechny možné situace.

Oproti tomu, pokud je použito procedurální programování, může programátor řešit, které události jsou pro něj zajímavé a ty si sám naprogramovat. (Přesně ty události, které jsou vhodné pro jeho roboty a pro jeho počet robotů a pro daný typ hry).

Je možné implementovat oba dva přístupy. Jeden, který by byl využit u programování jednoho robota a druhý pro programování týmů. V této práci je však použit způsob procedurální, stejně jako v Netrobots. Důvodem bylo, že programování týmu přináší mnoho nových možností ve volbě algoritmů, a proto nechceme, aby se studenti museli přeučovat způsob programování. Naopak chceme, aby studenti, než přejdou k programování týmů, uměli přemýšlet nad tím, které situace je vhodné řešit a jak tyto situace rozpoznat.

Důraz byl kladen na co nejmenší počet příkazů. Jednak proto, že čím méně příkazů, tím snáze se zapamatují. Dále proto, že je snažší napsat komunikační knihovnu pro méně příkazů a také proto, že pomocí těchto jednoduchých příkazů se dají snadno programovat příkazy složitější (například vystřelení dávkou). Příkazy ovládající pohyb, skener a kanon jsou pro základní verzi dostatečné.

3.5 Módy

V této kapitole budou zanalyzována rozšíření, která byla přidána pro možnost pokračovat v programování po té, kdy základní verzi již studenti zvládají.

3.5.1 Vybavení

V rámci tohoto módu bylo řešeno kdy bude možné nakupovat rozšíření vybavení robota, způsob získávaní peněz a zda budou mít roboti informace o vylepšení vybavení jiných robotů, nebo ne.

Hra je navržena na více kol (tím je v Netrobots celý zápas) a vybavení lze nakupovat pouze mezi koly. Peníze se přidělují za způsobené poškození ostatním robotům pouze po proběhnutém kole. Mezi jednotlivými koly musí být také robot opraven, a to se odečítá z celkové částky. Jsou dva možné způsoby, kterými lze řešit nákup vylepšeného vybavení - programátor zvolí (mezi koly se čeká na rozhodnutí programátora), nebo robot zvolí (robot je k tomu programátorem naprogramován).

- Programátorem zvolené vybavení – v této verzi nakupované vybavení volí programátor. Je tedy nutné čekat na vstup od programátora.

  Výhodou tohoto přístupu je, že programátor je obvykle chytřejší než jeho algoritmus a snáze se dokáže rozhodnout, které vybavení je pro něj v dané

situaci nejlepší. Nevýhodou je nutná interakce s programátorem a neúplně autonomní robot.

- Robotem zvolené vybavení – v této variantě nakupované vybavení volí robot. Program nemusí čekat na reakci programátora a probíhá kontinuálně bez zásahu člověka.

  Takto lze také program spustit kdykoliv bez nutnosti zajištění přímého dohledu všech zúčastněných (všech programátorů) a také je možné poslat algoritmy učiteli , který je může spustit sám, bez nutnosti interakce se studentem (studenty).

  K programování algoritmu pro nákup vybavení je ovšem nutné znát možnosti vybavení předem.[2]

I přes vyšší náročnost na programování algoritmu pro koupi vybavení jsme se rozhodli, že robot bude plně autonomní. Tak bude možné poslat práci učiteli, který může provést kontrolu včetně jejího spuštění.

Robot musí mít dostupné informace o druhu vybavení a jejich parametrech, což probíhá ihned po připojení k serveru a je to zajištěno komunikační knihovnou.

3.5.2 Týmy

V rámci programování týmů bylo nutné vyřešit, jestli se roboti z jednoho týmu budou moci navzájem zraňovat (tzv. friendly-fire). Jestli budou mít společné peníze, nebo nebudou, a jestli server bude vědět, kteří roboti jsou ze stejného týmu.

V původní aplikaci mohl být robot zraněn i vlastní střelou, toto se blíží realitě, proto není důvod měnit toto chování pro týmy. Peníze chceme přidělovat pouze za aktivitu. Proto budeme peníze přidělovat každému robotu zvlášť.

Na základě těchto požadavků by server nemusel vědět o tom, kteří roboti jsou v týmu. Nicméně pro týmové typy her (capture the flag, capture the base) je důležité toto rozpoznat.

Protože komunikační kanál má každý robot vlastní, je nutné poslat identifikátor týmu. Tímto identifikátorem je řetězec, který je podrobněji popsán v sekci 5.6.2. Je zde například zmíněno, jak vytvořit bezpečný řetězec, aby se do týmu nemohli přidat roboti, kteří do něj nepatří, nebo jak se roboti mezi sebou dorozumívají.

Protože server musí znát příslušnost robotů k týmu, budou přidělovány každému robotu peníze také za týmové aktivity (takovou aktivitou je např. přinesení vlajky).

3.5.3 Překážky na mapě

U tohoto módu je důležité brát v úvahu:

- Druhy překážek a co mohou ovlivnit.
- Jak se server dozví rozložení překážek.

[2]Pokud by programátor neznal vybavení, které je možné koupit, musel by implementovat umělou inteligenci, která by to dokázala rozhodnout, což je velice náročné.

- Jakým způsobem budou roboti informováni o překážkách.

Roboti mají příkazy ke střelbě, pohybu a skenování. Proto dává smysl, aby překážky dokázaly ovlivňovat činost robota právě při těchto povelech.

Jsou dva způsoby, jak vytvořit mapu:

- Náhodně generovaná mapa – sever by mohl mapu náhodně vygenerovat podle zvolených parametrů. Avšak napsat rozumný generátor terénu, když chceme povolit přidávání druhů překážek, je náročné. Náročnost se zvyšuje s povolením přidávání typů her, jako je například capture the flag, kde se roboti musí mít možnost dostat k vlajce.

- Mapa načtená ze souboru – druhým řešením je, že zadavatel (provozovatel serveru) vytvoří mapu a ta se při spuštění serveru načte ze souboru. V takovém případě lze snadno pracovat s novými typy překážek a upravovat mapu podle typu hry, která bude s touto mapou spuštěna, protože člověk dokáže zhodnotit, které podmínky taková mapa musí splňovat. Toto řešení jsme si vybrali. A pro snadnější vytváření mapy jsme připravili grafické rozhraní, ve kterém se bude mapa kreslit a až bude dokreslená, pak ji zadavatel uloží do souboru.

Způsobů jak informovat roboty o překážkách je několik.

- Po připojení k serveru, před startem zápasu – jedním ze způsobů je informovat o všech překážkách na mapě před startem zápasu (tedy ihned po připojení). Tato varianta má tu výhodu, že na začátku robot zná celou mapu a může podle toho začít fungovat. Ale i pokud tuto variantu nepoužijeme a zadavatel se rozhodne podat informace o překážkách hned při startu, pak je může předem zveřejnit a student si je může napsat do svého programu.

- Skenovat překážky – Další možností je, že robot musí překážky naskenovat. Skenování má tu výhodu, že robot se musí starat o překážky a musí si sám mapovat prostředí. Otázkou zůstává, jakým způsobem by toto skenování fungovalo

  - Jedna varianta je, že po skenování překážek robot dostane informaci o všech překážkách v okolí.
  - Druhá varianta je podobná funkci skenování robota. Robot dostává zpět informaci o tom, že v zadané výseči je překážka, ale přesnější informaci musí robot získat zpřesňováním zaměření. Nicméně, toto zpřesňování zaměření už musí robot použít na soupeřovy roboty, tedy další zaměřování nepřináší nic nového. Není také úplně jasné, jaká překážka by se měla nahlásit, kdyby jich v zadané výseči bylo více a případně by se nacházely i ve stejné vzdálenosti od robota.

- Překážky v okolí – poslední možností je, kombinovat dvě předchozí varianty. Robot bude dostávat informace o okolních překážkách v rámci přijímání informací o stavu robota, ale stále se musí pohybovat po mapě, aby ji mapoval. Toto je také varianta, která byla použita.

3.5.4 Povolání

Po zavedení týmů bylo logickým krokem vytvoření různých povolání. V rámci povolání byly řešeny následující body.

- Která povolání budou použita.
- Jak na ně budou reagovat ostatní módy - např. vybavení (5.8).
- Jak budou jejich akce vyhodnocovány.

Povolání které jsme se rozhodli zařadit jsou:

Tank (tank) – ten je známý z Netrobots. Je to robot, který střílí po jiných robotech.

Repairman (opravář) – je robot, který opravuje jiné roboty.

Opravování robotů funguje podobným způsobem jako střela. Pouze místo ubírání životů je přidává. Přidávat životy lze jen robotům v okolí - to aby se opravář vystavoval nebezpečí. Povolání opravář, dostává peníze za přidání životů jiným robotům. Opravář může opravovat pouze několikrát za kolo. Kolikrát za kolo může opravovat a kolik životů opravou vyléčí je uvedeno v jeho vybavení.

Miner (miner) – je robot, který pokládá miny.

Také miny fungují podobným způsobem jako střela. Rozdíl je v tom, že se nestřílí, ale pokládají. Min je omezený počet. Jsou dvě možnosti uvedení miny v činnost. Jeden způsob je při najetí robota na minu a druhý, který byl také použit, je při dálkovém odpálení miny. Druhý způsob s sebou nese výhodu, že lze odpálit minu, kterou již robot nechce používat. K poškození jiného robota je nutné pomocí skeneru lokalizovat soupeřova robota a minu odpálit až bude soupeřův robot v účinném dosahu miny. Z toho vyplývá,že se nejedná o pasivní čekání až na minu najede soupeřův robot.

3.5.5 Nové typy her

Na začátku bylo nutné vyřešit, jak roboti zjistí, která varianta hry se hraje.

- Zadavatel oznámí, že na jeho serveru se bude hrát tento typ hry.

  Tehdy je možné programovat algoritmy specializované pro daný typ hry. A tato varianta bude nejspíše běžnější, protože je snazší.

- Oznámení klientovi po připojení, jaký typ hry se hraje. To umožní programovat algoritmy universální, což je náročnější a mohlo by se to stát další výzvou pro studenty, kteří již zvládají programovat specializované algoritmy.

  Pokud zavedeme tento způsob, pak stále zadavatel může oznámit, že na jeho serveru se hraje tento typ hry, tedy stále bude možné programovat specializované algoritmy, ale bude možné programovat i universální algoritmy. Protože tato varianta přináší možnosti dalšího růstu, a zároveň nezabraňuje jednodušší variantě, rozhodli jsme se implementovat tuto variantu.

Nové typy her, které byly implementovány jsou Capture the base a Capture the flag.

Capture the base U tohoto módu je důležité oznámit polohu pozic, komu patří a z kolika procent (pozice nemusí patřit nikomu). Oznamování pozic probíhá na začátku hry a stav jednotlivých míst je oznamován po každém kole.

Capture the flag Podobný problém nastal u varianty Capture the flag, kde bylo nutné vyřešit, jak roboti získají informaci, kde jsou vlajková místa a jak získají informaci, že některý z robotů nese vlajku. Tento problém byl vyřešen oznámením vlajkových míst robotům na začátku hry a po každém kole informovat, který robot nese vlajku. Protože nebylo cílem oznámit polohu robota, který vlajku nese, bylo každému robotu přidáno id a u skeneru bylo též vraceno id robota, aby bylo možné najít robota, který nese vlajku.

3.6 Zvolení programovacího jazyka

Bylo důležité rozhodnout, v jakém jazyce bude psán server a který jazyk bude výchozí pro programování robotů.

3.6.1 Programovací jazyk pro server

Programovací jazyk pro server byl zvolen C#, protože ve verzi *.NET framework 4.5* jsou dostupné awaitery, díky kterým se asynchronní operace píší poměrně snadno. A tyto asynchronní operace budou potřeba pro správnou komunikaci s klienty, pokud chceme pracovat s co nejvíce klienty zároveň.

3.6.2 Programovací jazyk pro programování robotů

Programovací jazyk pro programování robotů by mohl být odlišný od programovacího jazyku pro server. Pak by studenti museli používat další programovací jazyk, pokud by chtěli přidávat nové módy. Protože předpokládáme, že studenti si nevystačí s rozšířeními námi dodanými a budou si chtít udělat také vlastní. Zvolili jsme stejný programovací jazyk pro server i pro programování robotů.

Stále však zbývá rozhodnout, jestli algoritmus, který napíší studenti bude generován přímo do spustitelného `exe` souboru, nebo bude dodán program, který jako parametr přijme cestu k studentem naprogramovanému algoritmu uloženém v souboru. Druhá varianta možňuje používat více abstrakce avšak je také vzdálenější od klasického programování v C#. Proto jsme se rozhodli, že student bude programovat přímo v C#, kde bude používat námi dodanou knihovnu, která zajistí odeslání příkazu na server a zpracování odpovědi a jeho kód bude kompilován do spustitelného `exe` souboru.

3.7 Zvolený nástroj pro ukládání dat

V aplikaci je zapotřebí mít možnost některá nastavení uložit (např. mapu viz 3.5.3). Od těchto nastavení chceme, aby je mohl uživatel v případě zájmu měnit a

nemusel k tomu spouštět další program. Proto jsme vybírali z textových formátů. Čím jednodušší formát pro uživatele, tím lépe, protože se mu bude snáze měnit. Nejvhodnější by bylo:

```
POCET_ROBOT : 2
POCET_TYMU: 2
```

S ohledem na tento požadavek jsme zkoumali formáty:

- XML – poměrně složitý (obasuhuje mnoho konstrukcí) a výsledný soubor je poměrně velký
- JSON - poměrně malý a přehledný formát
- Vlastní - není standardní, proto pro něj nejsou nástroje na transformace. A oproti JSON bude jen o málo lepší přehlednost.

Rozhodli jsme se ukládat data do formátu JSON. Požadavkem bylo, aby se s formátem dobře pracovalo, tedy aby serializovat objekt do řetězce a deserializovat z řetězce do objektu bylo co nejjednodušší. Také jsme chtěli využít nástroj, který používá mnoho programátorů. Zvažovali jsme:

- `DataContractJsonSerializer` – je nutné anotovat data. Anotace částečně znepřehledňují kód - pokud je programátor nezná, tak se jich zalekne.
- `JSON.NET` – pro tento typ jsme našli mnoho referencí od programátorů. Serializace i deserializace nepotřebuje žádné anotování dat.
- `JavaScriptSerializer` – pomalejší než `JSON.NET`, a nenašli jsme tolik referencí.

Protože `JSON.NET` se nám používá dobře, a umožňuje procházet JSON pomocí LINQ, rozhodli jsme se zvolit tento balíček.

3.8 Protokol

Jak bylo zmíněno v 3.2, rozhodli jsme se pro architekturu server-klient. A pro komunikaci je nutné domluvit pravidla – protokol.

Nejprve je nutné navrhnout formát protokolu. Nabízí se binární, nebo textový. Binární je obvykle těžší naprogramovat, než textový. Protože každý jazyk má jinak uložené typy a přechod mezi typy je náročnější naprogramovat, než serializaci těchto typů do řetězce. Rozhodli jsme se zvolit textový formát. Aby co nejméně docházelo k latencím, zvolili jsme vlastní formát, v němž jsou zprávy co nejkratší.

Dále jsme se rozhodovali, jestli se bude komunikovat po TCP, nebo UDP. Protože pro robota je důležité znát svou polohu, vadilo by, kdyby se nějaký packet ztratil, proto jsme se rozhodli pro komunikaci po TCP. Tento protokol zajišťuje, že ke ztrátě paketu nedojde. Ke komunikaci využíváme sockety.

Pro serializaci a deserializaci zpráv jsme se rozhodli použít vlastní protokol a nikoliv standardizovaný, protože jsme po protokolu chtěli, aby řetězce byli co nejkratší a aby se snížilo množství dat posílaných po síti. V praxi se nám stalo, že v místě, kde jsme používali Netrobots byla síť velmi slabá.

3.9 Zobrazování

Předpokládáme, že práce bude využívána spíše na počítači, než na tabletu. Ve školách obvykle mají operační systém Windows, proto cílíme hlavně na tento operační systém.

`WinForms` jsou podporovány na Windows, a měli by fungovat také na Mac OS i Linuxu. Zobrazování by se dalo programovat i v jiném jazyce, než C#, protože na serveru díky zobrazování off-line není závislé, ale s rozšířeními je nutné zobrazování upravovat, proto bylo použito znovu C#. Pokud budou studenti nebo zadávající programovat rozšíření, musí to provádět v C#, protože v něm je programován server. A také `WinForms` jsou oproti `WPF` technologicky jednodušší, protože nepoužívají další technologii krom C# (`WPF` používají `XML`), proto nám přišlo vhodnější volit `WinForms`.

4. Vývojová dokumentace

Aplikace byla vyvíjena ve Visual Studiu 2015 v rámci jednoho `solution` a jednotlivé knihovny jsou `project`ech. Celá práce je psána v programovacím jazyce C#.

Práce je rozdělena na serverovou a klientskou část (dále jen server a klient). Obě části používají základní komunikační knihovnu. Schéma je na obr. 12.

- Klient – u něj probíhá algoritmus ovládající roboty a odesílají se z něj příkazy robotů na server.
- Server – na něm probíhá simulace. Server také odesílá odpovědi na příkazy klientů.
- Komunikační knihovna (knihovna `BaseLibrary`) – stará se o posílání zpráv po síti. Serializuje objekty na zprávu a deserializuje zprávu do objektů.

Obrázek 12. Schéma rozdělení aplikace.

4.1 BaseLibrary

Knihovna `BaseLibrary` zajišťuje serializaci a deserializaci zpráv od serveru ke klientovi a od klienta k serveru. Skládá se z příkazů, protokolu a vybavení.

- Příkazy – popis příkazů s jejich atributy.
- Protokol – soubor pravidel, jak převést příkaz na řetězec a zpět.
- Vybavení – popis vybavení s jejich atributy.

4.1.1 Příkazy

Příkazy se v projektu nacházejí ve složce *BaseLibrary/command/*, jsou rozděleny podle typů do podsložek na common (společné všem robotům), equipment (zjišťující vybavení), tank (speciální příkazy pro typ `Tank`), miner (specializové příkazy pro typ `Miner`), repairman (specializované příkazy pro typ `Repairman`).

Protože se dalo očekávat, že příkazy se příliš rozrůstat nebudou, ale reakce na příkazy budou různé, byly příkazy navrhovány s ohledem na podporu návrhového vzoru *Visitor* (ten je popsán v knize Návrhové vzory [10] v kapitole 33), a proto mají metody `accept`.

V projektu jsou čtyři druhy *Visitorů* – `ICommonVisitor`, `IMinerVisitor`, `IRepairmanVisitor`, `ITankVisitor`. Všechna tato rozhraní jsou umístěna ve složce *BaseLibrary/visitors*.

- `ICommonVisitor` – obstarává běžné akce společné všem robotům a akce k zjištění vybavení.
- `IMinerVisitor` – obstarává akce pouze pro `Miner`.
- `IRepairmanVisitor` – obstarává akce pouze pro `Repairman`.
- `ITankVisitor` – obstarává akce pouze pro `Tank`.

Visitor se používá pro zpracování příkazu od robota na straně serveru, což je popsáno v 4.3.1

4.1.2 Protokol

Protokol je nutný pro určení pravidel, jak bude serializovaný objekt vypadat, aby se dal deserializovat. Protokol je textový a jednořádkový.

Návrh počítal s možným přidáním více protokolů, proto je nutné na začátku u klienta i u serveru provést *handshake*, kde se strany domluví jaký protokol budou používat (*handshake* obstarává třída `HandshakeProtocol` a je umístěna v knihovně `ServerLibrary` – pro *handshake* ze strany serveru a v knihovně `ClientLibrary` - pro *handshake* ze strany klienta). Proto musí být protokol popsaný atributem `ProtocolDescription`, kde je uvedeno jednoslovní jméno tohoto protokolu. Každý protokol musí být potomkem `AProtocol` a měl by se nacházet ve složce *BaseLibrary/protocol*. Námi implementovaný protokol (`ProcotolV1_0`) se jmenuje `"v1.0"`.

Příkazy pro `ProtocolV1_0` se nacházejí v *BaseLibrary/command/v1.0* (dále označovány V1.0 příkazy) a každý je potomkem klasického příkazu, který reprezentuje a implementuje `ACommand.Sandable`, který má metodu `Serialize` vracející serializovanou podobu příkazu. Každá třída V1.0 příkazu obsahuje *factory*, která slouží k transformaci z klasického příkazu na V1.0 příkaz a deserializaci příkazu. V konstruktoru `ProtocolV1_0` se registrují tyto *factory* do `comandsFactory`. `comandsFactory` je třída shromažďující *factory* a při požadavku na transofrmaci, nebo deserializaci, projde zaregistrované *factory* a najde, která daný požadavek umí zpracovat a vrátí její výsledek. Pokud žádná *factory* nevyhovuje, vrací výchozí hodnotu pro daný typ.

Pro snadnější implementaci serializace slouží statická třída `ProtocolV1_0Utils`, která má implementováno, jak se má který objekt serializovat a dokáže rozdělit takto serializovaný objekt na menší řetězcové části. Ty se musí dále deserializovat na základní typy. Důvodem proč nebyla použita reflexe byl požadavek na rychlost, protože lze očekávat, že server bude muset zpracovávat mnoho příkazů.

`AProtocol` obsahuje dvě metody `GetCommand(string)` vracející deserializovaný příkaz z řetězce. A metodu `GetSendableCommand(ACommand)` serializující příkaz. Obě tyto metody jsou virtuální, protože někdy může být vhodnější používat jinou implementaci, než přes `commandsFactory`.

4.1.3 Vybavení

Třídy popisující vybavení jsou nutnou součástí základní komunikační knihovny, protože je používá jak server, tak klient. Definice těchto tříd se nachází ve složce *BaseLibrary/equipment*. Těmito třídami jsou:

`Armor` – představující pancíř robota.

`Motor` – představující zařízení umožňující pohyb.

`RepairTool` – představující opravářské nástroje pro robota `Repairman`.

`MineGun` – představující minomet pro robota `Miner`.

`Gun` – představující kanon pro robota `Tank`.

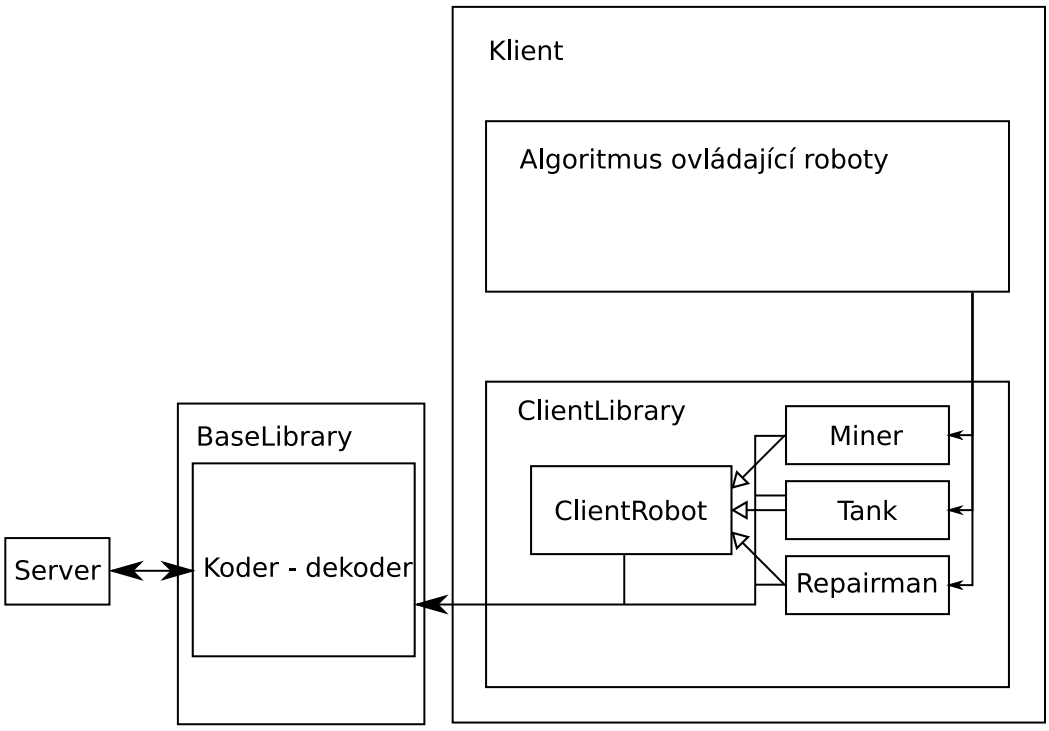
4.1.4 ModDescription

Atribut `ModDescription` se používá k rozeznání, u kterých tříd se má volat statický konstruktor při načítání rozšíření. Tyto módy (jejich *.dll*) musí být ve složce se spustitelným `exe` souborem. Všechny třídy, které se někam registrují mají mít tento atribut.

4.2 Klient

Klient (schéma na obr. 13) se dále dělí na algoritmus a komunikační knihovnu.

- Algoritmus - ten dodává student používající tento produkt.
- Komunikační knihovna (`ClientLibrary`) - zajišťuje komunikaci se serverem (odesílání příkazů a přijímání odpovědí), aby se student mohl soustředit pouze na programování algoritmu ovládající roboty.



Obrázek 13. Schéma rozdělení klienta.

4.2.1 ClientLibrary

Klientova komunikační knihovna se skládá z obecné třídy pro roboty a tříd konkrétních povolání.

- Obecná třída pro roboty (`ClientRobot`) - implementuje společný základ pro všechna povolání a běžné zpracování příkazů (pro aktualizaci stavu, pro opravu mezi koly, apod.).
- Konkrétní povolání (`Tank`, `Miner` a `Repairmen`) - rozšiřují `ClientRobot` a jsou v nich implementovány pouze příkazy příslušné danému povolání (např. pro `Miner` položení a odpálení miny).

Uživatel využívá knihovnu tím, že si vytvoří instanci třídy reprezentující konkrétní povolání a poté používá naprogramované metody, které se starají o komunikaci se serverem a přijmutí zprávy.

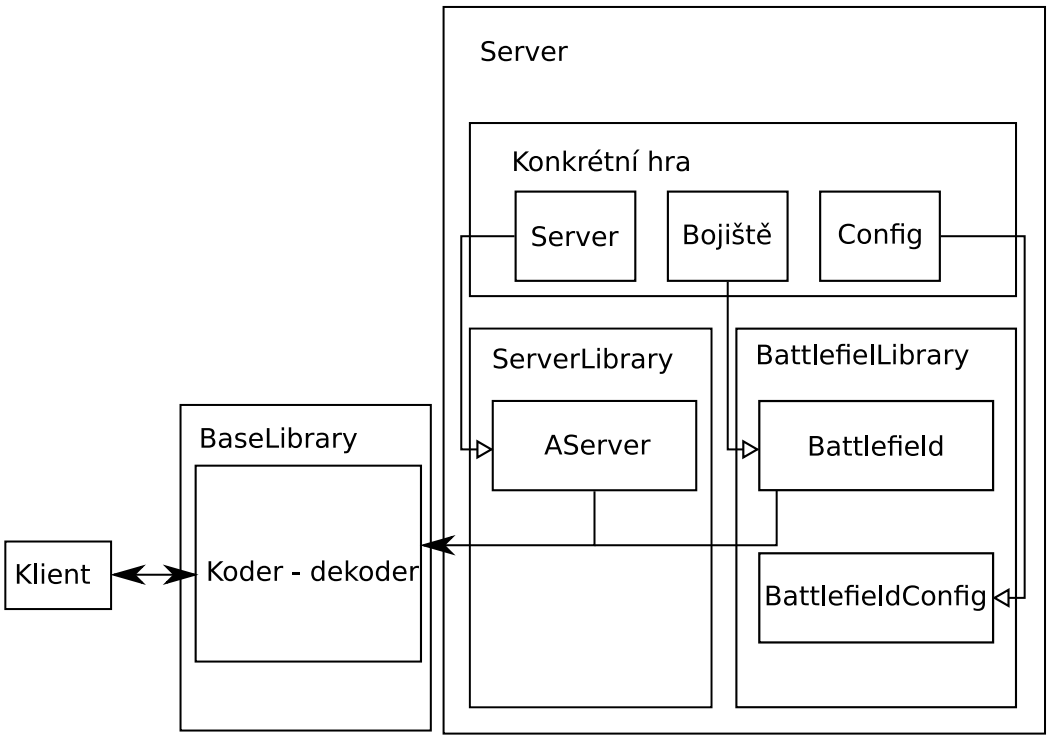
Ke každému příkazu pro robota existuje i jeho `Async` verze vracející `Task`. Očekávané zacházení s těmito `Task` je pro poslání příkazů za více robotů v jednom tahu a poté se čeká na jejich dokončení (viz 5.6.1).

Uvnitř zmíněných tříd `ClientRobot`, `Tank`, `Miner` a `Repairman` je k odesílání příkazu využívána základní komunikační knihovna společná jak pro klienta tak pro server `BaseLibrary` 4.1.

4.3 Server

Server (schéma na obr. 14) se dělí na knihovnu provádějící simulaci , knihovnu komunikační a algoritmus pro konkrétní typ hry.

- Simulační knihovna (`BattlefieldLibrary`) - simuluje průběh hry a zpracovává příkazy od klienta a odesílá odpovědi.
- Komunikační knihovna (`ServerLibrary`) - slouží k snadnějšímu vytvoření serveru.
- Algoritmus pro konkrétní typ hry - slouží k popsání zvláštností dané hry.



Obrázek 14. Schéma rozdělení klienta.

4.3.1 BattlefieldLibrary

Knihovna provádějící simulaci je zde nutná pro to, aby se implementace konkrétního typu hry mohla zaměřit pouze na její zvláštnosti. Také je pro programátora - klienta výhodnější, když bude simulace probíhat pro každou hru stejně.

V této knihovně je popsán herní cyklus. Hra probíhá jeden zápas, který se dělí na kola a ty dál na tahy.

`BattlefieldLibrary` se dále dělí na simulaci, zpracovatele příkazů, konfiguraci a záznam o tahu. Navíc je v této knihovně také serverová konfigurace.

- Simulace – je implementována v souboru `Battlefield.cs` obsahující třídu `Battlefield`.
- Zpracovatelé příkazů – pro zpracování příkazů od klienta pomocí návrhového vzoru *Visitor* je implementován v souboru `Battlefield.Visitors.cs`.
- Konfigurace (třída `BattlefieldConfig`) – pro konfiguraci spouštěného bojiště.
- Záznam o tahu (třída `BattlefieldTurn`) – pro pozdější zobrazení průběhu zápasu.
- Serverová konfigurace (třída `ServerConfig`) – konfigurace pro handshake a pro načtení vybavení ze souboru. (Nemůže být v `ServerLibrary` kvůli kruhové závislosti).

Všechny tyto části (krom třídy `ServerConfig`) se nacházejí v projektu ve složce *BattlefieldLibrary/battlefield*.

Simulace Po úspěšném *handshake* je předán socket do třídy `Battlefield`, kde se vytvoří instance `DefaultRobot`, dokud robot nevykoná příkaz `InitCommand`, kdy se nastaví konkrétní povolání robota. Třída `ClientRobot` na straně klienta odpovídá třídě `BattlefieldRobot` na straně serveru, `Miner`, `Repairman`, `Tank` odpovídají svým protějškům u klienta. Všechny tyto implementace robotů jsou ve složce *BattlefieldLibrary/battlefield/robot*.

Hra je rozdělena na kola. Kolo je vyhodnocováno:

1. Náhodné rozmístění robotů (s ohledem na případné překážky – může se stát, že generátor nenajde vhodnou polohu, pak je nutné polohu zadat ručně). Vynulování použitých vybavení (probíhá v metodě `newBattle`).
2. Vyhodnocování tahů.
3. Nákup.

Níže je popsán průběh simulace odehrávající se v metodě `singleCycleTurn` (kromě prvního bodu, ten se provádí v metodě `lisen`) (simulace se provádí v reálném čase viz kapitola 3.1.3).

1. Získání příkazů od robotů (čeká se maximálně `TIME_FOR_TURN_WAIT` ms).
2. Zpracování přijatých příkazů.
3. Proběhne `afterProcessCommand` v daném typu hry (více o tom v 4.3.2).
4. Pohyb robotů.
5. Pohyb a výbuch střel.
6. Výbuch min.
7. Proběhne `afterMovingAndDamaging` v daném typu hry (více o tom v 4.3.2) (např. v capture the base pokrok v zabírání bází).
8. Obnovení mrtvých robotů - pouze pokud je povoleno.
9. Zjištění, jestli je tímto tahem kolo dohrané.
10. Odeslání stavu robotů.
11. Odpojení dlouho nekomunikujících robotů.

Visitors Příkazy se zpracovávají za pomoci návrhového vzoru *Visitor* (ten je popsán v knize Návrhové vzory [10] v kapitole 33). Implementace *Visitorů* jsou v souboru `Battlefield.Visitors.cs`. Tento soubor je rozšířením třídy `Battlefield`, která je `partial`, tedy *Visitor* může přistoupit k privátním proměnným instance `Battlefield`. Jednotlivé implementace se poté používají pro konkrétní stav výpočtu simulace a pro daný typ robota. Stavy jsou – `GET_COMMAND` (před tím, než se připojí všichni roboti a pošlou Init), `FIGHT` (probíhá výpočet kola), `MERCHANT` (probíhá nákup mezi koly). Stavy slouží k rozpoznání, který visitor se má použít.

Konfigurace Konfigurace slouží k nastavení bojiště. Má tyto parametry.

`MAX_ROBOTS` – kolik robotů se připojí do bojiště (po připojení tohoto počtu robotů se spustí simulace).

`MAX_TURN` – na kolik tahů se maximálně hraje kolo.

`MAX_LAP` – na kolik kol se hraje zápas.

`ROBOTS_IN_TEAM` – kolik robotů je v jednom týmu (`MAX_ROBOTS` by mělo být tímto číslem dělitelné).

`RESPAWN_TIMEOUT` – kolik tahů trvá, než se znovu zrodí robot.

`RESPAWN_ALLOWED` – flag jestli je znovuzrození robotů povoleno.

`EQUIPMENT_CONFIG_FILE` – cesta k souboru s informacemi o vybavení.

`OBTACLE_CONFIG_FILE` – cesta k souboru s informacemi o překážkách.

`MORE` – pole obsahující další potřebné informace (například vlajky, nebo báze).

Záznam o tahu Záznam o tahu se používá při vykreslování. Záznamy se serializují do JSON a jsou ukládány na disk. Záznam o jednom tahu je na jednom řádku. `BattlefieldTurn` je použit pro dynamické přidávání jednotlivých parametrů do listu a před serializováním se konvertuje na `Turn` z `ViewerLibrary` (4.6.1). Metody které `BattlefieldTurn` obsahuje jsou:

`AddScan` – slouží k přidání informací o skenu.

`AddRobot` – slouží k přidání informací o robotu.

`AddBullet` – slouží k přidání informací o střele.

`AddMine` – slouží k přidání informací o mině.

`AddRepair` – slouží k přidání informací o opravě.

`AddMore` – slouží k přidání dalších informací (např. vlajek, bází apod.).

Pokud chce někdo přidávat do `More`, musí se nejprve zaregistrovat zavoláním `BattlefieldTurn.RegisterMore()`, kde se mu vrátí číslo zaregistrované pozice, do které může ukládat data zavoláním `AddMore(object, int)`.

4.3.2 Implementace konkrétních typů hry

Implementace konkrétního typu hry se skládá ze tří hlavní částí - bitevní pole, server a konfigurace.

- Bitevní pole (dále bojiště) – slouží ke specifikování zvláštností pro konkrétní typ hry.
- Server – slouží k vytvoření bojiště.
- Konfigurace – slouží ke konfigurování bojiště (počet robotů, počet kol, počet tahů, apod.).

Bitevní pole (dále bojiště) je potomkem třídy `Battlefield` z bitevní knihovny (`BattlefieldLibrary`). Třída reprezentující bojiště musí implementovat metody:

`AddToInitAnswereCommand` – která slouží k přidání informací pro robota na začátku hry. Vrací přijímaný `InitAnswerCommand`.

`AddToRobotStateCommand` – která slouží k přidání informací pro robota po každém tahu. Vrací přijímaný `RobotStateCommand`.

`afterProcessCommand` – co se provede ihned po zpracování příkazů (ještě před pohybem robotů).

`afterMovingAndDamaging` – co se provede před odesláním stavu robotů (po započítání zranění).

`NewLapState` – vyhodnocení kola (`LabState.WIN` - jestli je již vyhráno, `LabState.LAP_OUT` - nebo jestli vypršel počet tahů, `LabState.NONE` - nebo kolo pokračuje dalším tahem).

Třídní attribut `ModDescrioption` způsobí zavolání statického konstruktoru při načítání módů, toho je vhodné pro registrování místa k přidávání informací k příkazu. Registrování se provádí ve statickém konstruktoru. K tomu slouží metoda `RegisterSubCommandFactory` a jako parametr přijímá `ISubCommandFactory`. `ISubCommandFactory` slouží k deserializaci a serializaci informací, které jsou přidávány (např. informace o stavu bází). Jak se registrovat je ukázáno v ukázce kódu 9.

```
static BaseCapture() {
  POSITION_IN_BATTLE_TURN = BattlefieldTurn.RegisterMore();
  POSITION_IN_ROBOT_STATE_COMMAND =
     RobotStateCommand.RegisterSubCommandFactory(SUB_COMMAND_FACTORY);
}
```

Kód 9. Ukázka registrování do tahu a příkazu s informacemi o stavu.

Přidání informací v metodě probíhá přístupem k poli `MORE` na zaregistrovanou pozici a vložením objektů, které jsou posílány klientovi (jak je ukázáno v kódu 10).

```
protected override RobotStateCommand AddToRobotStateCommand(
      RobotStateCommand robotStateCommand, BattlefieldRobot r) {
  robotStateCommand.MORE[POSITION_IN_ROBOT_STATE_COMMAND] =
                                                    bases.ToArray();
  return robotStateCommand;
}
```

Kód 10. Ukázka přidání informací o stavu bází do `RobotStateCommand`.

Bojiště může číst od předka z těchto proměnných:

`robots` – seznam všech robotů.

`robotsById` – slovník robotů podle jejich id.

`robotsByTeamId` – slovník seznamu robotů podle týmové id.

`battlefieldTurn` – viz popis v kapitole 4.3.1. `battlefieldTurn` je `null` dokud nezačne první tah.

`turn` – aktuální číslo tahu.

Server rozšiřuje třídu `AServer` z knihovny `ServerLibrary` a zakládá nové bojiště a nastavuje port, ke kterému se klient připojuje. Musí implementovat metody:

`GetGameTypeCommand` – slouží k vytvoření odpovědi, ve které je zapsán typ hry a počet robotů v týmu.

`NewBattlefield` – vytvoří novou instanci bojiště a tuto instanci vrátí.

Konfigurace slouží ke konfigurování bojiště. Konfiguraci je vhodné mít pro konkrétní typ hry a tehdy musí být potomkem `BattlefieldConfig` z knihovny `BattlefieldLibrary`. Proměnné `BattlefieldConfig` jsou popsány v odstavci konfigurace v kapitole 4.3.1.

Instance tohoto objektu se používá při vytváření bojiště (zavoláním metody `NewBattlefield` u instance `AServer`). Také je možné konfiguraci načíst ze souboru pomocí statické metody `DeserializeFromFile` jíž se předá cesta k souboru ve třídě `Battlefield`. Třída `Battlefield` má též metodu k serializaci (`Serialize`) instance do JSON. Tato metoda přijímá jako svůj argument cestu k souboru, do kterého se má instance serializovat.

U všechny typů her, které jsou součástí této práce, lze měnit port a konfiguraci pomocí argumentů předávaných při spuštění konkrétního typu hry (jak je popsáno v manuálu v 5.2.1). Prvním argumentem je číslo portu a druhým je cesta ke konfiguračnímu souboru (přesněji k souboru, ve kterém je konfigurace serializována).

4.4 Překážky

Rozšíření implementující překážky je odděleno do dvou částí – knihovna s překážkami a grafické rozhraní pro vykreslení mapy.

- Knihovna s překázkami (`ObstacleMod`) – slouží k definování překážek, serializování a deserializování, apod.
- Grafické rozhraní (`ObstacleMap`) – slouží k umístění překážek.

4.5 ObstacleMod

Mód překážek je rozdělen do tříd:

`ObstacleManager` – shromažďuje informace o překážkách a kontroluje, jestli je pohyb, skenování nebo střela ovlivněna překážkami.

`ObstaclesAroundRobot` – zajišťuje serializaci a deserializaci překážek v okolí robota.

`IObstacle` – společné rozhraní pro všechny překážky.

`IMoveInfluence` – rozhraní pro překážky ovlivňující pohyb.

`IScanInfluence` – rozhraní pro překážky ovlivňující skenování.

`IShotInfluence` – rozhraní pro překážky ovlivňující střelbu.

Dále jsou třídy v *ObstacleMod/obtacle* reprezentující konkrétní překážku.

`Wall` – reprezentuje zeď. Přes zeď nelze střílet, ani se pohybovat.

`Sand` – reprezentuje písek. V písku se robot pohybuje pomaleji.

`Shielding` – reprezentuje stínění. Přes stínění neprojde skenovací paprsek.

4.5.1 ObstacleManager

`ObstacleManager` se stará o načtení překážek ze souboru. V třídě `Battlefield` během vypočítávání pohybů, zjišťování skenování a výpočtu dráhy střel se volají příslušné metody (`MoveChange`, `CanScan` a `ShotChange`). Tyto metody využívají metodu `getEnumerator`, která vrací `IEnumerable` vracející překážky, které jsou v dráze pohybu, v dráze skenovacího paprsku, nebo v dráze střely. Překážky jsou procházeny podle vzdálenosti od počátku dráhy. Během průchodu jsou překážky označovány, aby nebylo možné stejnou překážku vyhodnotit vícekrát. Na konci metody jsou všechny překážky odznačeny.

`IEnumerable` se vypočítává nově pokud se změní některá ze souřadnic. V metodě `MoveChange` lze měnit počátek i konec pohybu (aby bylo možné implementovat např. posuvné pásy), proto autor překážky musí správně spočítat všechny souřadnice. `ShotChange` může měnit pouze dopad střely, při změně je vypočítán nejbližší průsečík s překážkou z místa výstřelu a nový `IEnumerable` se vypočítává z tohoto místa (to se dále bere jako místo výstřelu) do změněného místa dopadu. Díky tomuto mechanismu lze naprogramovat vyvýšený terén, kde z jedné strany se dostřel bude prodlužovat a z druhé zkracovat.

4.5.2 ObstaclesAroundRobot

Je označená atributem `ModDescription`, a očekává, že překážky si registrují své *factory* do `ObstaclesAroundRobot.OBSTACLE_FACTORIES`. Tato třída serializuje `IObstacle` jako pole do `RobotStateCommand` a proto má též statickou metodu `GetFromState`, která vrátí pole `IObstacle` z `RobotStateCommand`.

4.6 Vizualizace

Vizualizace je rozdělena do dvou částí - knihovna a grafické rozhraní.

- Knihovna (`ViewerLibrary`) – obsahuje struktury k serializování a deserializování jednotlivých tahů.
- Grafické rozhraní (`Viewer`) – obsahuje algoritmus pro animovanou vizualizaci zápasu.

4.6.1 ViewerLibrary

Obsahuje pouze dva soubory - `JSONSerializer.cs` a `Turn.cs`.

- `JSonSerializer.cs` – obsahuje třídu `JSONSerializer`, ta slouží k serializování a deserializování tahu. Původně bylo v plánu mít více možných formátů serializace, ale později se od tohoto plánu upustilo.
- `Turn.cs` – obsahuje třídy potřebné k zobrazování:

  `Bullet` – představuje vystřelenou střelu. Obsahuje proměnné:
  - `X` – souřadnice střely na hrací ploše v tomto tahu.
  - `Y` – souřadnice střely na hrací ploše v tomto tahu.
  - `EXPLODED` – příznak, jestli střela v tomto tahu vybouchla.

  `Mine` – představuje položenou minu. Obsahuje proměnné:
  - `X` – souřadnice miny na hrací ploše.
  - `Y` – souřadnice miny na hrací ploše.
  - `EXPLODED` – příznak, jestli mina v tomto tahu vybuchla.

  `Repair` – představuje opravování. Obsahuje proměnné:
  - `X` – souřadnice místa opravy na hrací ploše.
  - `Y` – souřadnice místa opravy na hrací ploše.

  `Robot` – představuje robota. Obsahuje proměnné:
  - `TEAM_ID` – identifikátor týmu.
  - `SCORE` – skóre robota.
  - `GOLD` – peníze robota.
  - `HIT_POINTS` – zdraví robota.
  - `X` – souřadnice robota na hrací ploše.
  - `Y` – souřadnice robota na hrací ploše.
  - `ANGLE` – natočení určující směr pohybu robota.
  - `NAME` – jméno robota.

  `Scan` – představuje skenovací výseč. Obsahuje proměnné:
  - `ANGLE` – směr hlavního skenovacího paprsku.
  - `PRECISION` – parametr šířky výseče.
  - `DISTANCE` – vzdálenost naskenovaného objektu (může být větší než úhlopříčka hrací plochy, tehdy nenaskenoval nic).

`X` – souřadnice na hrací ploše odkud bylo skenováno.

`Y` – souřadnice na hrací ploše odkud bylo skenováno.

`Turn` – přestavuje tah. Obsahuje proměnné:

`TURN` – číslo kola

`BULLETS` – pole střel na ploše.

`MINES` – pole min na ploše.

`REPAIRS` – pole oprav na ploše.

`ROBOTS` – pole robotů na ploše.

`SCANS` – pole skenovacích výsečí.

`MORE` – pole polí objektů sloužící k dalším informací (např. informace o vlajkách).

4.6.2 Viewer – WinForms

`Viewer` je rozdělen na tři části - grafické rozhraní, nástroje a formulář.

- grafické rozhraní (složka *gui*) – v této složce je implementace základního vykreslení a rozhraní pro vykreslování.
- nástroje (složka *utils*) – nástroje pro ransformaci bodů (`DrawerUtils`) a pro transformaci barevných systému (`ColorUtils`).
- formulář (`AppForm`) – algoritmus načítání a vykreslování animace za použití `WinForms`.

4.6.3 Grafické rozhraní

Grafické rozhraní obsahuje rozhraní pro vykreslení `IDrawer`. `DefaultDrawer` implementuje vykreslování, je použito v `AppForm.DrawingBattle` pokud není nastaveno jiné zobrazování pomocí metody `SetDrawerFactory`. `IDrawerMore` je rozhraní pro vykreslování specifických věcí pro rozšíření, třídy implementující toto rozhraní musí mít bezparametrický konstruktor. Třídy implementující `IDrawerMore` jsou načteny a zaregistrovány při změně `IDrawer`.

5. Manuál pro uživatele

Úkolem hry je naprogramovat autonomního virtuálního robota, který bojuje v aréně s ostatními roboty. V této kapitole je popsáno vše potřebné k tomu, aby mohl být robot naprogramován. Návody uvedené v této kapitole jsou psané pro operační systém Windows 10 a pro programovací jazyk C#.

5.1 Jak získat aplikaci

Před vlastním programováním autonomního robota, je nutné aplikaci získat. Postup získání aplikace je následující:

1. V prohlížeči otevřete stránky `https://github.com/aahriman/RobotBattlefield`
2. Kliknete na tlačítko na stránce *Clone or download*.
3. V okně, které se objeví, kliknete na tlačítko *Download ZIP*.
4. Rozbalíte stažený soubor
   (a) Otevřete složku, do které se soubor stáhl.
   (b) Kliknete pravým tlačítkem na soubor.
   (c) Vyberete možnost *Rozbalit soubor zde* případně *Extract file here*.
5. Vytvoří se složka *RobotBattlefield-master*.

Před dalším pokračováním se doporučuje přenést složku *RobotBattlefield-master* do vlastní složky na ploše (např. do složky *moji roboti*).

5.2 Spuštění hry

Nejprve musí být spuštěna aréna a až poté jsou spuštěni roboti.

5.2.1 Spuštění arény

Návod ke spuštění spuštění arény:

1. Dostaňte se do složky *RobotBattlefield-master\DeadmatchBattlefield\bin\Release*.
2. Dvojím klepnutím na *DeadmatchBattlefield.exe* spustíte arénu.

Tato aréna je spuštěna pro dva roboty, každého v jednom týmu. Pokud budete chtít připojit více robotů, musíte provést následující.

1. Upravte soubor *defaultconfig.json*
   (a) Ve složce *RobotBattlefield-master\DeadmatchBattlefield\bin\Release*.
   (b) Otevřete soubor *defaultConfig.json* pomocí Poznámkového bloku.

(c) Pokud soubor *defaultConfig.json* nevidíte, ale vidíte pouze soubor *defaultConfig*.

i. V horní liště Průzkumníku klikněte na *Zobrazení*

ii. Zaškrtněte možnost *Přípony názvů a souborů*

(d) Po otevření změňte číslo za textem *"MAX_ROBOTS":* na vámi požadovaný počet robotů.

2. Vytvořte zástupce pro program *DeadmatchBattlefield.exe*.

(a) Kliknete pravým tlačítkem na soubor *DeadmatchBattlefield.exe*.

(b) Vyberete možnost *Vytvořit zástupce*.

(c) Vytvořil se soubor *DeadmatchBattlefield.exe - zástupce*.

3. Nastavte parametry při spuštění u *DeadmatchBattlefield.exe - zástupce*

(a) Klikněte pravím tlačítkem na soubor *DeadmatchBattlefield.exe - zástupce*.

(b) Vyberete možnost *Vlastnosti*.

(c) V záložce zástupce do kolonky *Cíl:* za cestu k *DeadmatchBattlefield.exe* zapište `2000` a `defaultConfig.json` oddělené mezerou. (První je číslo portu ke kterému se chcete připojit a druhé je cesta ke konfiguračnímu souboru).

(d) Konec cíle vypadá takto:

```
\DeadmatchBattlefield.exe" 2000 defaultConfig.json
```

(znak `"` (uvozovka) musí být též na začátku cíle)

(e) Potvrdíte zmáčknutím na tlačítko *OK*.

Nyní můžete *DeadmatchBattlefield.exe - zástupce* umístit přímo do vaší složky s *RobotBattlefield-master* (např. *moji roboti*) a arénu budete spouštět dvojím poklepáním na *DeadmatchBattlefield.exe - zástupce*

5.2.2 Spuštění robotů

V projektu je již naprogramován robot spot. Ten lze přidat do zápasu tímto způsobem:

1. Spustíte arénu (pokud jste ji ještě nespustili).

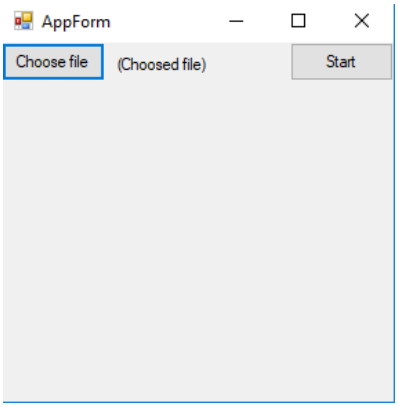
2. Ve složce *RobotBattlefield-master\Spot\bin\Release* poklepete na *Spot.exe* a tím je robot přidán do arény.

5.3 Zobrazení průběhu zápasu

Průběh zápasu je možné sledovat až po skončení tohoto zápasu. Provedete to následovně:

1. Ve složce *RobotBattlefield-master\Viewer\bin\Release* poklepáte na *RobotViewer.exe* .
2. V okně, které se objeví (obr. 15), kliknete na tlačítko *Choose file* a projdete do složky *RobotBattlefield-master\DeadmatchBattlefield\bin\Release* a vyberete soubor *arena_match<cislo portu>.txt*. (Název souboru lze změnit změněním atribut `MATCH_SAVE_FILE` v *defaultConfig.json*).
3. Potvrdíte tlačítkem *Otevřít*.
4. Pak kliknete na tlačítko *Start* a začne animace průběhu zápasu

.



Obrázek 15. Ukázka otevřeného okna zobrazení.

.

5.4 Jak začít programovat

Jak začít programovat záleží na používaném vývojovém prostředí (tzv. IDE), postup, který je uveden dále, je popsán pro Visual Studio a pro SharpDevelop.[1]

5.4.1 Pro Visual studio 2015

Spustíte Visual studio a dále pokračujete podle postupu:

1. Vytvoříte projekt.
   (a) V nabídce horní lišty vyberete *File* → *New* → *Project*.
   (b) Z nabídky *Templates* vyberete *Visual C#*.
   (c) Uprostřed nahoře vyberte v seznamu *.Net Framework 4.5* (nebo vyšší) vedle *Templates:*
   (d) Do kolonky vedle *Name* napíšete jméno robota (např. Robot).

[1]SharpDevelop je možné stáhnout z adresy `https://sourceforge.net/projects/sharpdevelop/` kliknutím na tlačítko *Download* se stáhne instalační soubor. SharpDevelop je volně dostupný – za jeho používání se platit nemusí. Pro instalaci stačí poklepat na stažený soubor a dále pokračovat podle pokynů v průvodci instalací.

(e) [Volitelné - je možné přeskočit] Zvolíte cestu ke složce v níž se nachází složka *RobotBattlefield-master* (např. složka *moji roboti*). V *Location* kliknete na tlačítko se třemi tečkami a pak se objeví okno průzkumníka, kde tuto složku vyberete (tam se bude ukládat váš projekt).

(f) V okně které se objeví vyberete *Console Aplication* (musí být v modrém rámečku).

(g) Zmáčknete tlačítko *OK*.

2. Přidáte knihovnu *ClientLibrary.dll* a *BaseLibrary.dll* k vašemu řešení (Tyto knihovny jsou v *RobotBattlefield-master**ClientLibrary**bin**Relase*)

(a) Vpravo kliknete na *Solution Explorer* (aby se nabídka rozbalila).

(b) Pravým tlačítkem kliknete na *References* → *Add references*.

(c) Kliknete na tlačítko *Browse ...* .

(d) Dostaňte se do složky *RobotBattlefield-master*, dále *ClientLibrary**bin**Relase* a poklepejte na *ClientLibrary.dll*.

(e) Ještě jednou klikněte na tlačítko *Browse ...* a přidejte *BaseLibrary.dll*.

(f) Zmáčkněte tlačítko *OK*.

Spustitelný `exe` soubor se vytvoří v adresáři *bin**Debug*, který se nachází v adresáři podle *Location*.

1. Kliknete pravým tlačítkem na jméno vašeho projektu.
2. Kliknete na *Open Folder in File Explorer*.
3. Otevře se složka, do které se ukládá projekt.

Nyní již můžete začít psát algoritmus pro svého robota.

5.4.2 Pro SharpDevelop 4.4

Spustíte dříve nainstalovaný program SharpDevelop, a dále pokračujete podle níže popsaného postupu:

1. Vytvoření projektu.

(a) V nabídce horní lišty vyberete *File* → *New* → *Solution*.

(b) Vpravo nahoře vyberte v seznamu *.Net Framework 4.5* (nebo vyšší) vedle *Templates:*.

(c) Do kolonky vedle *Name* napíšete jméno robota (např. Robot).

(d) [Volitelné - je možné přeskočit] Zvolíte cestu ke složce v níž se nachází složka *RobotBattlefield-master* (např. složka *moji roboti*). V *Location* kliknete na tlačítko se třemi tečkami a pak se objeví okno průzkumníka, kde tuto složku vyberete (tam se bude ukládat váš projekt).

(e) V okně, které se objeví, vyberete *Console Aplication* (musí být v modrém rámečku).

(f) Zmáčknete tlačítko *Create*.

2. Přidáte knihovnu *ClientLibrary.dll* a *BaseLibrary.dll* (Tyto knihovny jsou v *RobotBattlefield-master**ClientLibrary**bin**Relase*).

   (a) Vlevo rozkliknete tlačítkem + váš projekt.

   (b) Pravým tlačítkem klikněte na *References* → *Add references*.

   (c) Klikněte na záložku *.Net Assembly Browser*, a pak na tlačítko *Browse. . . .*

   (d) Dostaňte se do složky *RobotBattlefield-master*, dále *ClientLibrary**bin**Relase* a poklepejte na *ClientLibrary.dll*.

   (e) Ještě jednou klikněte na tlačítko *Browse . . .* a přidejte *BaseLibrary.dll*.

   (f) Zmáčkněte tlačítko *OK*.

Spustitelný `exe` soubor se vytvoří v adresáři *bin**Debug*, který se nachází v adresáři podle *Location*.

1. Kliknete pravým tlačítkem na jméno vašeho projektu vlevo.
2. Kliknete na *Properties*.
3. V otevřeném okně přečtete dole umístění vedle *Project folder*.

Nyní již můžete začít psát algoritmus pro svého robota.

5.5 Vyhodnocování a fungování

Hra (zápas) je rozdělená na kola a ty se dále dělí na tahy. V každém tahu je možné poslat pouze jeden příkaz. Server čeká, než dostane příkaz od všech robotů, nebo než vyprší doba čekání (nastavená na 100 ms). Pokud robot dlouho nekomunikuje(800 ms), pak je odpojen, aby hra mohla dále plynule pokračovat.

Tah se vyhodnocuje takto:

1. Získání příkazu.
2. Zpracování příkazů a odeslání odpovědí.
3. Pohyb robotů.
4. Výbuchy střel.
5. Výbuchy min.
6. Respawn robotů (pokud je povolen).
7. Odeslání informací o stavu robotů `RobotStateCommand` (o aktualizaci stavu robota se nemusíte starat, více o `RobotStateCommand` na 5.11.

Pohyb robotů se provádí takto:

1. Nastavení požadovaného výkonu (`WantedPower`).

2. Změna směru (pokud je dostatečně nízký výkon).

3. Změna rychlosti:

Zrychlení	(a) Výkon nastaven na nižší z `WantedPower` a `Motor.SPEED_UP_TO` (pokud výkon již není vyšší). (b) Výkon se zvýší o `Motor.SPEED_UP` (pokud nebylo ještě dosáhnuto `WantedPower`).
Zpomalení	(a) Výkon snížen o `Motor.SPEED_DOWN` (nejníže však na `WantedPower`).

4. Změna pozice podle výkonu případně překážek.

5.6 Jak psát algoritmus

Psaní algoritmu pro jednoho a pro více robotů se liší, proto jsou oba popsány zvlášť.

5.6.1 Jak psát algoritmus pro jednoho robota

Při programování jednoho robota, je nutné nejprve zvolit jeho povolání, které musí být bojové (nemůže to být `Repairmen`) (jak se jednotlivá povolání liší je popsáno v 5.9). Následující postup je zpracován pro psaní algoritmu pro `Tank`, pro `Miner` je postup shodný, jen místo `Tank` se použije výraz `Miner`.

Nejprve je nutné robota vytvořit. Poté robota připojíte k serveru a nastavíte mu jméno a přiřadíte k týmu. A dále píšete algoritmus a používáte metody z 5.9. Jak se toto zahájení v kódu napíše je popsáno v ukázce kodu 11.

```
Tank nasRobot = new Tank(); // vytvoříme robota
nasRobot.Connect(); // připojíme k serveru
nasRobot.ProcessInit(nasRobot.Init("Nas prvni tank", "Muj team 1"));
/* pojmenování  a přiřazení k teamu a zpracování odpovědi */
```

Kód 11. Ukázka programu pro připojení robota do arény.

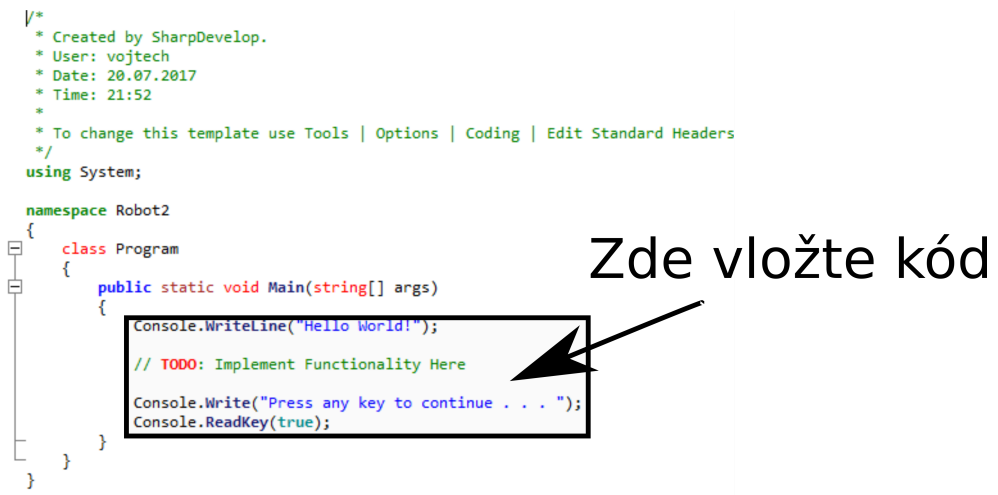
Po této sekci následuje váš algoritmus. Algoritmus pro robota, který kolem skenuje a pokud něco uvidí tak tam vystřelí je napsán v ukázce kódu 12.

```
Tank nasRobot = new Tank(); // vytvoříme robota
nasRobot.Connect(); // připojíme k serveru
nasRobot.ProcessInit(nasRobot.Init("Nas prvni tank", "Muj team 1"));
while(true){
    for(int angle = 0; angle < 360; angle +=30) {
       ScanAsnwerCommand odpoved = nasRobot.Scan(angle, 10);
       if (odpoved.ENEMY_ID != nasRobot.ID) {
            nasRobot.Shot(angle, odpoved.RANGE);
```

```
        }
    }
}
```

Kód 12. Ukázka programu robota, který kolem skenuje a pokud něco nalezne, tak vystřelí.

Tento kód se píše do metody `Main` (jak je ukázáno na obr. 16). A robot se spustí zelenou šipkou v IDE, nebo poklepáním na vygenerovaný `exe` soubor.



Obrázek 16. Ukázka, kam se má psát kód.

5.6.2 Jak psát algoritmus pro více robotů

Připojení více robotů probíhá podobným způsobem jako připojení jednoho robota. Ukázka jak toto připojení provést pro dva roboty je v kódu 13.

```
Tank nasRobot1 = new Tank(); // vytvoříme prvního robota
Tank nasRobot2 = new Tank(); // vytvoříme druhého robota
nasRobot1.Connect(); // připojíme k serveru
nasRobot1.ProcessInit(nasRobot.Init("Nas prvni tank", "Muj team 1"));
nasRobot2.Connect(); // připojíme k serveru
nasRobot2.ProcessInit(nasRobot.Init("Nas druhy tank", "Muj team 1"));
/* pojmenování  a přiřazení k teamu a zpracování odpovědi */
```

Kód 13. Ukázka programu pro připojení více robotů do arény.

Změnou při psaní algoritmu pro tým robotů je používání metod končící `Async` a po zavolání těchto metod čekání na jejich vyhodnocení pomocí `Task.WaitAll`.

Po této sekci následuje algoritmus. V následujícím kódu (14) je ukázáno jak naprogramovat roboty, kteří kolem sebe skenují a vystřelí jen pokud oba dva vidí stejného robota. (Tento příklad je pouze ilustrativní).

Doporučujeme vygenerovat názvy týmů unikátně, aby se nestalo, že se budou některá jmnéna shodovat. Vygenerování unikátního jména a uložení do proměnné `TEAM_NAME` se napíše `String TEAM_NAME = Guid.NewGuid().ToString();` . A poté místo naříklad `"Muj team 1"` se použije `TEAM_NAME`. V ukázce 14 je použit už tento přístup s unikátním ID.

```
String TEAM_NAME = Guid.NewGuid().ToString();
Tank nasRobot1 = new Tank(); // vytvoříme prvního robota
Tank nasRobot2 = new Tank(); // vytvoříme druhého robota
nasRobot1.Connect(); // připojíme k serveru
nasRobot1.ProcessInit(nasRobot.Init("Nas prvni tank",  TEAM_NAME));
nasRobot2.Connect(); // připojíme k serveru
nasRobot2.ProcessInit(nasRobot.Init("Nas druhy tank",  TEAM_NAME));
/* pojmenování  a přiřazení k teamu a zpracování odpovědi */
while(true){
  for(int angle = 0; angle < 360; angle +=30) {
    Task<ScanAsnwerCommand> task1 = nasRobot1.ScanAsync(angle, 10);
    Task<ScanAsnwerCommand> task2 = nasRobot2.ScanAsync(angle, 10);
    Task.WaitAll(task1, task2);
    ScanAnswerCommand odpoved1 = task1.Result;
    ScanAnswerCommand odpoved2 = task2.Result;
    if (odpoved1.ENEMY_ID == odpoved2.ENEMY_ID &&
      odpoved2.ENEMY_ID != nasRobot2.ID &&
      odpoved1.ENEMY_ID != nasRobot1.ID) {
      Task.WaitAll(
        nasRobot1.ShotAsync(angle, odpoved1.RANGE),
        nasRobot2.ShotAsync(angle, odpoved2.RANGE)
      );
    }
  }
}
```

Kód 14. Ukázka kódu pro tým robotů, kteří skenují a pokud vidí stejného soupeře, tak vystřelí.

5.7 Jak soupeřit s někým jiným

V této kapitole je popsáno, jak spolu mohou dva cizí roboti soupeřit. Je zapotřebí znát *IP adresu* stroje, na který se chce programátor připojit (to je stroj, na kterém bude spuštěna aréna). Zjištění ip adresy hráče (tento postup provádí ten, kdo bude spouštět arénu):

1. Spustíte příkazový řádek
   (a) Zmáčknete klávesu `WINDOWS + R`.
   (b) Napíšete cmd a potvrdíte klávesou `ENTER`.
2. Napíšete `ipconfig` a potvrdíte klávesou `ENTER`.
3. Přečtete si řetězec za `Link-local IPv6 Address` a to je vaše adresa (řetězec vypadá např. takto `fe80::cc16:8c4:7384:5c17%9`).

Aby se vaši roboti uměli připojovat i k jiným robotům, musíte vyměnit metodu `Connect()` za `Connect(args)`. Díky této změně při spuštění můžete zvolit kam se má robot pomocí spouštěcích parametrů připojit. Postup je stejný jako při přidávání parametrů při spuštění arény (5.2.1 bod 2 a 3). Parametry při spuštění jsou v pořadí *ip adressa* a *port*. Tedy konec cíle při spuštění přes zástupe pro robota Spot by vypadal takto:

```
\Spot.exe" fe80::cc16:8c4:7384:5c17%9 2000
```

5.8 Vybavení

K pochopení dalšího fungování hry je nutné popsat používané vybavení. Existuje pět druhů vybavení: pancíř, motor, kanon, opravářské nástroje a minomet.

- Pancíř (`Armor`) - pancíře, podle druhu se odvíjí maximální počet životů robota.
- Motor (`Motor`) - slouží k pohybu robota.
- Kanon (`Gun`) - slouží k střílení střel - je speciálním vybavením pro povolání `Tank`.
- Opravářské nástroje (`RepairTool`) - slouží k opravování - jsou speciálním vybavením pro povolání `Repairman`.
- Minomet (`MineGun`) - slouží k pokládání min - je speciálním vybavením pro povolání `Miner`.

Atributy vybavení shrnuje tabulka:

Společné všech	`COST` – cena za příslušný exemplář vybavení. `ID` – identifikační číslo tohoto exempláře.
`ARMOR`	`MAX_HP` – maximální počet životů robota s tímto pancířem.
`Motor`	`MAX_SPEED` – rychlost při maximálním výkonu motoru. `ROTATE_IN` – v kolika % výkonu motoru lze změnit směr jízdy. `SPEED_UP` – zvednutí výkonu za tah při zrychlování. `SPEED_DOWN` – snížení výkonu za tah při zpomalování. `SPEED_UP_TO` – maximální zvýšení výkonu z 0.

Gun	`MAX_BULLETS` – počet hlavní (na začátku jsou všechny nabity, po výstřelu se nabijí 20 tahů). `MAX_RANGE` – dostřel - maximální vzdálenost, na kterou lze vystřelit. `SHOT_SPEED` – rychlost letu střely. `ZONES` – popis soustředných kruhů, seřazený podle vzdálenosti (obvykle platí, čím blíže, tím větší zranění). `EFFECT` – udělené zranění. `DISTANCE` – vzdálenost do které se udílí `EFFECT`.
RepairTool	`MAX_USAGES` – maximální počet oprav za kolo. `ZONES` – popis soustředných kruhů, seřazený podle vzdálenosti (obvykle platí, čím blíže, tím větší oprava). `EFFECT` – přidané životy. `DISTANCE` – vzdálenost, do které se udílí `EFFECT`.
MineGun	`MAX_MINES` – maximální počet min na mapě. `ZONES` – popis soustředných kruhů, seřazený podle vzdálenosti (obvykle platí, čím blíže, tím větší zranění). `EFFECT` – přidané zranění. `DISTANCE` – vzdálenost, do které se udílí `EFFECT`.

5.9 Popis příkazů

Níže jsou popsány příkazy, které mohou roboti používat. Příkazy jsou rozděleny do skupin - společné pro všechny roboty a pro konkrétní povolání.

5.9.1 Společné příkazy

Některé akce mají všichni roboti. Těmito akcemi jsou:

`Wait` - slouží k čekání. Používá se, pokud nemá být v tahu nic provedeno. Nebo pokud má být čekáno, než bude robot obnoven (při respawn, nebo na konci kola).

Vyvolá se zavoláním metody `void Wait()`. Tento příkaz nemá žádnou odpověd. Po tomto příkazu přímo následuje informace o stavu.

`Scan` - slouží k ovládání skeneru.

Vyvolává se zavoláním metody `Scan(double angle, double precision)` vracející `ScanAnswerCommand`, její parametry mají význam:

`angle` - směr skenování ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0°.

`precision` - je parametr (rozšíření) výseče, ve které se skenuje. Udáván je ve °, co to znamená je uvedeno na obr. 17.

.

Návratový typ `ScanAnswerCommand` má parametry:

`RANGE` - vzdálenost k nepříteli.

`ENEMY_ID` - ID naskenovaného robota. Pokud se skenování nepovedlo, pak je `ENEMY_ID` rovno `ID` robota, který skenoval.

`Drive` - slouží k ovládání pohybu.

Vyvolává se zavoláním metody `Drive(double angle, double power)`, která vrací `DriveAnswerCommand`, její parametry mají význam:

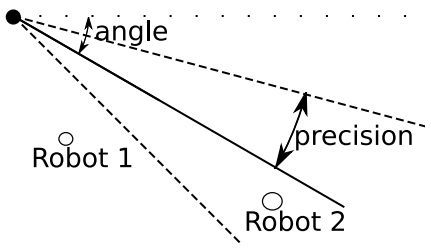
`angle` - směr jízdy ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0°.

`power` - je výkon (stlačení plynu) kolik chcete jet z `Motor.MAX_SPEED`.

Návratový typ `DriveAnswerCommand` má parametry:

`SUCESS` - `true` pokud se povedlo zatočit, `false` jinak. Zatočit lze, jen pokud je `Power` robota nižší jak `Motor.ROTATE_IN`.

Všechny metody mají i svou `Async` verzi, která je určená pro psaní týmů a její parametry i odpovědi získaná z nich jsou stejné. Používejte postup popsaný v kapitole 5.6.2.



Obrázek 17. Ukázka skenování. Robot 2 je vidět. Robot 1 není vidět, na to aby byl Robot 1 viditelný je nutné zvýšit `precision`*, nebo změnit úhel skenování (*`angle`*)*

5.9.2 Příkazy pro Tank

Speciální příkaz pro povolání Tank je pouze jeden. Níže je detailně popsán.

`Shot` - slouží k výstřelu z děla.

Vyvolává se zavoláním metody `Shot(double angle, double range)` vracející `ShotAnswerCommand`, její parametry mají význam:

`angle` - směr výstřelu ve stupních, měřeno po směru hodinových ručiček, kde 3. hodina je 0°.

`range` - je vzdálenost, do které se vystřelí. Lze vystřelit nejdále do vzdálenosti `Gun.MAX_RANGE`.

Návratový typ `ShotAnswerCommand` má parametry:

`SUCESS` - `true` pokud se povedlo vystřelit, `false` kanon nebyl k výstřelu nabit. Počet kanonu je `Gun.MAX_BULLET` a doba nabití je 20 tahů. Na začátku hry jsou všechny kanony nabity.

5.9.3 Příkazy pro Miner

Speciální příkazy pro povolání Miner jsou:

`PutMine` - slouží k položení miny na místo, kde se nachází robot.

Vyvolává se zavoláním metody `PutMine()`, jejímž návratovým typem je `PutMineAnswerCommand`.

Návratový typ `PutMineAnswerCommand` má parametry:

`SUCESS` - `true` pokud se povedlo položit minu, `false` pokud `Miner` již má položeno `MineGun.MAX_MINES` min.

`MINE_ID` - vrací číslo položené miny. Je nutné pro odpálení viz níže.

`DetonateMine` - slouží k opálení miny.

Vyvolává se zavoláním metody `DetonateMineint(int mineId)` vracející `DetonateMineAnswerCommand`, její parametr má význam:

`mineId` - číslo miny, kterou chcete odpálit. Toto číslo dostanete při příkazu `PutMine` viz výše.

Návratový typ `DetonateMineAnswerCommand` má parametry:

`SUCESS` - `true` pokud se povedlo odpálit minu, `false` pokud nebyla položena mina s `mineId`.

5.9.4 Příkazy pro Repairman

`Repair` - slouží k opravě okolí kolem robota.

Vyvolává se zavoláním metody `Repair()` nebo `Repair(int maxDistance)` vracející `RepairAnswerCommand`, parametry mají význam:

`maxDistance` - je maximální vzdálenost, do které chce robot opravovat. (V případě zavolání `Repair()` je opravováno do maximální vzdálenosti).

Návratový typ má parametry `RepairAnswerCommand`

`SUCESS` - `true` pokud bylo opraveno, `false` pokud se již opravovat nedá. Opravovat je možné jen `REPAIT_TOOL.MAX_USAGES` krát za kolo.

5.10 Jak nakupovat vybavení mezi koly

Mezi koly lze nakupovat vybavení. Nákupem lze získat lepší pancíř, výkonější motor nebo účinější zbraň. Mezi koly je též nutné se opravit, o to se však nemusíte starat, pokud nechcete též nakupovat (o to se postará knihovna).

Dostupné předměty, které lze koupit jsou k nalezení v `ClientRobot`. Všechny jsou uloženy do slovníků:

`MOTORS_BY_ID` – dostupné motory roztříděny podle jejich ID

`ARMORS_BY_ID` – dostupné pancíře roztříděny podle jejich ID

`GUNS_BY_ID` – dostupné kanony roztříděny podle jejich ID

`REPAIR_TOOLS_BY_ID` – dostupné opravářské nástroje roztříděny podle jejich ID

`MINE_GUNS_BY_ID` – dostupné minomety roztříděny podle jejich ID

Pokud chcete zjistit parametry vybavení, uděláte to tak, jak je ukázáno v ukázce 15 (případně za `MOTORS_BY_ID` dáte jiný slovník, který chcete projít). Procházet vybavení můžete až po zavolání `Connect` s alespoň jedním robotem.

```
foreach(var item in ClientRobot.MOTORS_BY_ID.Values) {
    Console.WriteLine(item);
}
```

Kód 15. Ukázka, jak lze vypsat parametry všech dostupných motorů

Nakupuje se pomocí příkazu `Merchant`, kde jako argumenty jsou předány identifikátory jednotlivých vybavení (jejich `ID`). To umožňuje šetřit a koupit si až bude dostatek peněz na nejlepší vybavení.

5.10.1 Vyhodnocení nákupu

Po příkazu k nákupu se provádí následující:

1. Přidá se $\frac{1}{10}$ životů z robotových `Armor.MAX_HP`.
2. Koupí se pancíř (pokud jsou na něj peníze) a robot dostane životů `Armor.MAX_HP`.
3. Chce-li se vyléčit do více životů, než robot má. Pak se vyléčí na kolik je požádáno (pokud je dostatek peněz, jinak se vyléčí do výše kreditu). Oprava stojí 1 peníz za každé započaté 4 životy.
4. Koupí se motor (pokud jsou na něj peníze)
5. Koupí se třídní vybavení (`Gun`, nebo `MineGun`, nebo `RepairTool`) (pokud jsou na něj peníze)

5.11 Jak zprovoznit rozšíření

Rozšíření je nástroj, jak pokračovat dále s tímto projektem, pokud už předchozí zvládáte. Pokud chcete použít některé z rozšíření, pak provedete následující:

1. Stáhnete rozšíření.
2. Knihovnu (soubor s příponou *.dll*) vložíte do stejné, ve které se vytváří spustitelný `exe` soubor.
3. Přidejte tuto knihovnu do svého programu stejným způsobem, jako jste přidávali *ClientLibrary.dll*.

Rozšíření pracují s příkazy převážně s příkazem s informacemi na začátku hry `InitAnswerCommand` a s příkazem s informacemi po tahu `RobotStateCommand`. Proto je nutné tyto příkazy zpracovávat alespoň částečně vlastním algoritmem.

Zpracovávat `RobotStateCommand` vlastním algoritmem lze dvěma způsoby. První je vytvořit si vlastní třídu robota a implementovat metodu `ProcessState` (jak je ukázáno v ukázce kódu 16). Třída bude rozšířením třídy `Miner`, nebo `RepairMan`, nebo `Tank`.

```
class MujTank : Tank {
  pulic overide ProcessState(RobotStateCommand state) {
    base.ProcessState(state); // nastavi pocet zivotu, pozici apod.
    // vlastni zpracovani state
  }
}
```

Kód 16. Ukázka jak implementovat vlastní třídu a v ní metodu `ProcessState`

Nebo po zavolání každého příkazu volat metodu `State`, která vrátí příkaz `RobotStateCommand` (v případě týmů tuto metodu volat až po čekání). Pokud se rozhodnete pro tento způsob, pak vytvářejte robota přes konstruktor s prvním parametrem `false`. Druhý parametr, je pro automatické nakupování. Např. tank by se vytvářel

```
Tank tank = new Tank(false, false);
```

Informace, které jsou navíc, jsou uloženy v `MORE`, což je pole objektů. Jak z tohoto pole získat přesně ty objekty, které jsou použity pro dané rozšíření je popsáno u každého rozšíření.

Stejný způsob funguje také pro `InitAnswerCommand`, jediné co je jiné, že `Init` se posílá pouze na začátku hry.

Níže jsou popsána dostupná rozšíření. Tyto rozšíření stahovat není potřebné, protože se stáhla společně s celou aplikací.

5.11.1 Capture the flag

Toto rozšíření spočívá ve změně smyslu hry. U tohoto rozšíření není cílem vystřílet všechny soupeře, ale donést soupeřovu vlajku do svého domečku vícekrát než soupeř. Za každé donesení získává celý tým peníze (na konci kola) a zvyšuje si bodové ohodnocení.

Aby roboti věděli, kam mají donést vlajku a kde se vlajky nachází posílají se v `InitAnswerCommand` (po zavolání `Init`, nebo `InitAsync`) informace o poloze vlajek. V `RobotStateCommand` se posílají informace o vlajce. Jak tyto informace vypadají, shrnuje následující tabulka:

FlagPlace (v `InitAnswerCommand`)	`TEAM_ID` - id týmu ke kterému patří toto místo `ID` - id tohoto místa `X` - horizontální pozice na hrací ploše `Y` - vertikální pozice na hrací ploše
Flag (v `RobotStateCommand`)	`FROM_FLAGPLACE_ID` - z kterého místa tato vlajka pochází `RobotId` - který robot ji nese

Sebrání nepřátelské nebo odevzdání nepřátelské vlajky probíhá po výpočtu všech zranění. Po odevzdání se ihned objeví nová vlajka na místě, ze kterého byla vzata.

5.11.2 Capture the base

Toto rozšíření spočívá ve změně smyslu hry. U tohoto rozšíření není cílem vystřílet všechny soupeře, ale obsadit všechny base.Prostředkem k obsazení všech bází je zraňování soupeřů a opravování vlastních robotů.

Aby roboti věděli, kde base jsou, jak moc jsou zabrány a kterým týmem, posílá se tato informace v `RobotStateCommand`. Popis jak tyto informace vypadají:

Base (v `RobotStateCommand`)	`Progress`	– jak moc je base zabrána
	`MAX_PROGRESS`	– kdy je plně zabrána (čím nižší číslo, tím dříve lze zabrat, ale také ztratit)
	`Team_Id`	– ktery tym ma zabrany `Progress`
	`X`	– horizontální pozice středu na hrací ploše
	`Y`	– vertikální pozice středu na hrací ploše

Roboti, kteří jsou blízko báze, která jim patří, jsou opravovány za $\frac{5}{100} *$ `Progress` životů. Protože hodnota `Progress` nemůže přesáhnout `MAX_PROGRESS` tak báze, které lze dříve zabrat méně léčí.

Zabírající roboti jsou ti, kteří jsou do vzdálenosti 25 od středu báze. Výpočet `Progress`u lze popsat takto:

1. Určení počtu zabírajících robotů pro každý tým. Tento počet se zapíše do `ProgressTeam`.
2. K `ProgressTeam` pro vlastníka báze se přičte `Progress`.
3. Určí se tým s nejvyšší `ProgressTeam` (uložené do `MaxProgressTeam`).
4. Sečte se `ProgressTeam` od ostatních týmů (uložené do `ProgressOtherTeam`).
5. `Progress` se nastaví na rozdíl `MaxProgressTeam` − `ProgressOtherTeam`.

Nejnižší `Progress` je však 0.

K získání informací o bázi slouží metoda `BaseCapture.GetBases` vracející pole `Base` a přijímající v argumentu `RobotStateCommand`.

5.11.3 ObstacleMod

Toto rozšíření spočívá v přidání překážek do mapy. Překážky mohou ovlivňovat pohyb, střelbu a skenování. Součástí aplikace jsou tyto překážky:

`Wall` (zeď) - nelze projet. Nelze prostřelit. Skenování neovlivňuje.

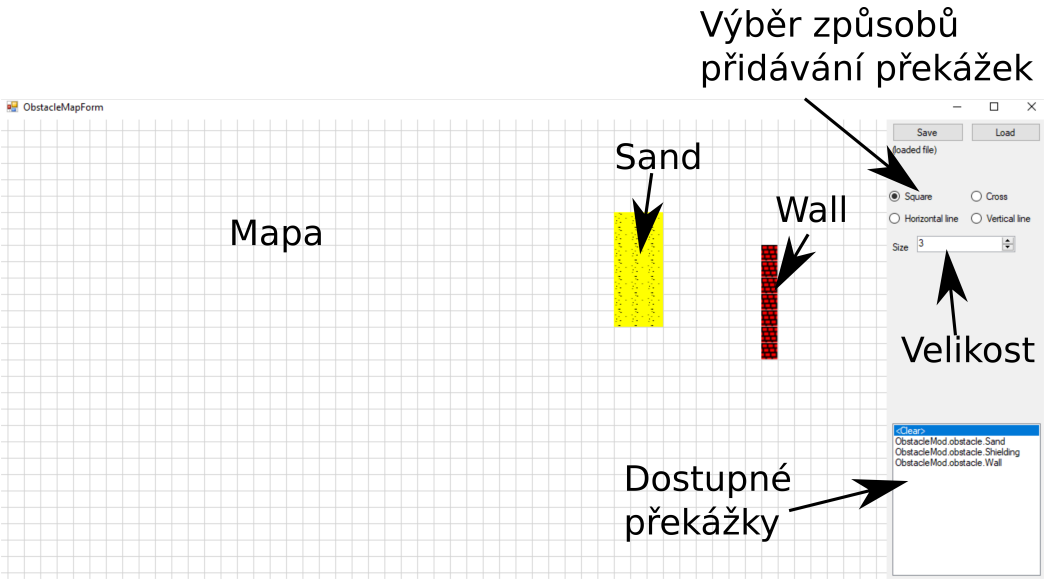
`Sand` (písek) - zpomaluje pohyb na polovinu.

`Shilding` (stínění) - není prostupný pro skenování.

Robot zjišťuje překážky v jeho okolí. Jaké jsou v okolí překážky lze zjistit zavoláním `ObstacleAroundRobot.GetFromRobotStateCommand`, kde jako argument je předán `RobotStateCommand`. Tato metoda vrací pole `IObstacle`. Zjistit

druh lze pomocí metody `GetType()` volanou na instanci `IObstacle` a porovnává se s `typeof(Wall)`, případně s jiným typem překážky.

Vygenerovat mapu lze pomocí spuštění programu *ObtacleMap.exe* ve složce *RobotBattlefield-master/ObtacleMap/bin/Release*. Spuštěná aplikace je vidět na obrázku obr. 18.Tlačítko *Save* slouží k uložení (musí se ukládat do nového souboru, protože jinak aplikace zamrzne). Tlačítko *Load* slouží k nahrání dříve vytvořené mapy a umožňuje editaci. Zvětšení se provádí pomocí kolečka ny myši a pohybovat se v rámci mapy lze po stisknutí a držení pravého tlačítka a posunu myši.



Obrázek 18. Ukázka programu pro generování mapy překážek s popisky.

Závěr

Hlavním cílem bylo vytvořit prostředí pro souboj virtuálních robotů s důrazem na odstranění nedostatků, zjištěných při použití projektu Netrobots v praxi.

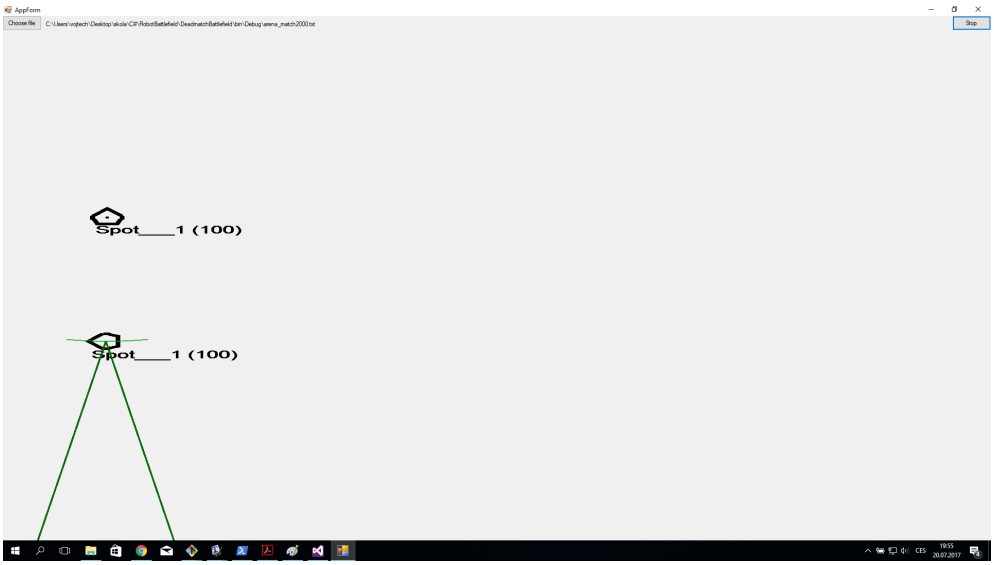
1.a.i Simulace byla naprogramována ve třídě `Battlefield` a průběh simulace byl v této práci podrobně popsán.

1.a.ii Byla implementována snadná komunikace mezi serverem a klientem z pohledu uživatele. Student je od komunikace odstíněn dodanou komunikační knihovnou a je dodán podrobný návod, jak tuto knihovnu používat.

Nejobtížnější mechanismus je asynchronní provádění akcí, avšak v kapitole 5.6.2 je popsáno, jak se tyto akce mají používat, kde mimo odeslání příkazu na server vše probíhá synchronně. Studenti by neměli mít s tímto přístupem problém, protože jsou zvyklí hrát týmové hry např. v tělocviku, kde také každý provádí akce samostatně ve prospěch celku.

1.b V kapitole 5.5 je přesně definované vyhodnocování akcí.

1.c Podařilo se také vylepšit vizualizaci, z které je již poznat, kam robot skenuje, který robot je který i kolik má který robot životů obr. 19.



Obrázek 19. Ukázka průběhu zápasu.

Ve vizualizaci řešeno nebylo, který robot má kolik bodů, kde se nachází centrum robota, pokud robot neskenuje a aby výbuchy odpovídaly skutečné ploše zranění. (Ikdyž, plocha výbuchu je poměrně malá a proto není problém rozeznat zda byl robot zasažen). Informace o bodech robotů však v serializovaném souboru obsažená je, proto jedním z nápadů na vylepšení aplikace je například přidání tabulky s výpisem robotů podle bodů. Možné by bylo také rozšířit vizualizaci o krokování tah po tahu, včetně funkce o tah zpět.

1.d Procedurální způsob programování robotů se ukázal jako velmi vhodný, protože usnadňuje přechod od jednoho robota k týmům. S vhodným pohledem na procedurální programování je programování podobné událostmi řízenému programování.

Tento způsob programování podporuje nutnou dovednost, kterou je rozdělení celku na menší části a přemýšlení o těchto částí odděleně.

2. Byla implementována zmíněná rozšíření hry a navíc je hru možné dále rozšiřovat za použití protokolu V1.0. Použití jiného protokolu je možné řešit změnou architektury.

Simulace lze snadno předělat na plně tahovou variantu a to změnou jednoho řádku

```
Task.WhenAny(
    Task.Delay(TIME_FOR_TURN_WAIT), allSendCommand.Task
).Wait();
```

na

```
Task.WhenAll(allSendCommand.Task).Wait();
```

v metodě `running`. Simulaci je také možné předělat na simulaci s okamžitým vyhodnocením akcí. Je pouze nutné kontrolovat odesílání stavu robota na konci tahu.

Dalším možným vylepšením je přidat stažení průběhu zápasu ze serveru ke klientovi (momentálně se průběh zápasu ukládá do souboru na serveru).

V budoucnu plánujeme napsat manuál pro další operační systémy a nabídnout tento projekt školám.

Seznam použité literatury

[1] Pavel Herout. *Učebnice jazyka Java.* České Budějovice: Kopp, 2010.

[2] Pavel Töpfer. *Algoritmy a programovací techniky.* Prometheus, 1995.

[3] Pavel Herout. *Učebnice jazyka C.* České Budějovice: Kopp, 2010.

[4] Marek Summerfield. *Python 3.* Computer Press a.s., 2010.

[5] Dokumentace k programovacímu jazyku karel. `http://mormegil.wz.cz/prog/karel/prog_doc.htm`. Přístup 2017-07-05.

[6] SGP Systems. Baltík. `http://sgp.cz/cz/`, 2001. Přístup 2017-07-05.

[7] Scratch. `https://scratch.mit.edu/`, 2005. Přístup 2017-07-05.

[8] Celostátní kolo soutěže mladý programátor 2017, kategorie a, b, c. `http://soutez.tib.cz/subdom/soutez/downloads/MP2017_CK_zadani.pdf`, 2017. Přístup 2017-07-04.

[9] Netrobots. `https://github.com/bonzini/netrobots`, 2011. Přístup 2017-07-04.

[10] Rudolf Pecinovský. *Návrhové vzory.* 2007.