# Netzwerk-basiertes RobotGame mit zentralem Server

Name: Michael Domanek
Klasse: 5BHIF
Katalognummer: 4
Beispielnummer: 35

# Contents


**1 Aufgabenstellung** **2**

**2 Hintergrund** **2**

2.1 Geschichte . . . . . 2

2.2 Installation . . . . . 2

2.3 Spieleigenschaften . . . . . 2

2.4 Spielende und Gewinner . . . . . 4

2.5 Programmierung . . . . . 4

2.6 Spielmodi und Challenges . . . . . 4

**3 UML** **4**

**4 Umsetzung** **7**

4.1 Spiel Idee . . . . . 7

4.2 Serveroptionen . . . . . 9

4.2.1 width . . . . . 9

4.2.2 bullet-speed . . . . . 9

4.2.3 bullet-damage . . . . . 9

4.2.4 robot-speed . . . . . 9

4.2.5 health . . . . . 10

4.2.6 robot-rotation . . . . . 10

4.2.7 turret-rotation . . . . . 10

4.2.8 fire-countdown . . . . . 10

4.2.9 max-players . . . . . 10

4.2.10 port . . . . . 10

4.2.11 not-shoot-and-move . . . . . 10

4.2.12 json-config . . . . . 10

**5 Verwendung** **11**

**6 Code-Beispiele** **11**

**7 Quellen** **15**

# 1 Aufgabenstellung

Netzwerk-basiertes RobotGame wie robowiki mit zentralem Server.

# 2 Hintergrund

Robocode ist ein Spiel, bei dem das Ziel darin besteht, einen Roboter zu programmieren, der gegen andere Roboter antritt. Der Spieler hat keinen direkten Einfluss auf das Spielgeschehen, sondern programmiert die KI des Roboters und gibt vor, wie er sich verhalten soll und auf Ereignisse reagieren soll.

Ürsprünglich war Robocode gedacht Kindern bzw. Anfängern spielerisch die Programmiersprache Java beizubringen. In der Sprache wurde auch das Spiel geschrieben und unterstützt dadurch auch alle gängigen Betriebssystemen(Windows, macOS, Linux, ...)

## 2.1 Geschichte

Matthew A. Nelson hat im Jahr 2000 begonnen das Spiel zu entwickeln . Das Projekt hat zuerst als persönlicher Freizeitprojekt gestartet und 2001 hat er es professionell für IBM gemacht. 2005 wurde es dann Open Source, aber die Weiterentwicklung wurde gestoppt. Viele Programmierer haben zu dem Projekt RobocodeNG, das von Flemming N. Larsen geleitet wurde beigetragen und Erweiterungen hinzugefügt und Bugs gefixt. 2006 wurden das Projekt dann als Version 1.1 mit dem offiziellen Robocode Projekt zusammengefügt. Seit dem gibt es immer wieder neue Versionen.

## 2.2 Installation

Das Spiel ist ein Java-Programm das man sich herunterladen kann. Dann kann man seinen ersten Roboter programmieren mit Hilfe der Robowiki. Danach kann dann seinen Roboter gegen andere einfache Beispielroboter antreten lassen. Wenn man gegen alle Roboter gewonnen hat, kann man sich bessere Roboter aus dem Internet herunterladen. Es gibt auch Tuniere in dene die besten Roboter antreten und viele der besten Roboter stehen auch zum Download zur Verfügung.

## 2.3 Spieleigenschaften

Jeder Roboter besteht aus einem Untergestell, einer Waffe und einem Radar. Beim fahren bewegt sich alles. Beim Rotieren kann man das Untergestell, die Waffe und das Radar unabhängig voneinander bewegen. Die Waffe ist natürlich um andere Roboter abzuschießen. Das Radar um andere Roboter zu finden.

Das Spiel verwendet das Kartesische Koordinatensystem, das heißt (0, 0) ist unten links. Die Rotation des Roboters ist natürlich 360° und 0° ist im Norden.


Figure 1: Screenshot aus dem originalem Spiel

Die Beschleunigung des Roboters ist 1 Pixel/Runde und die Verlangsamung ist 2 Pixel/Runde. Die maximale Geschwindigkeit beträgt 8 Pixel/Runde.

Jeder Roboter startet am Anfang mit einer 100% Energie. Der Roboter verliert jedes Mal Energie, wenn er gegen eine Wand fährt, wenn er von einer gegnerischen Kugel getroffen wird, wenn er einen Gegner rammt oder gerammt wird oder wenn er sein Waffe abfeuert. Er bekommt Energie zurück wenn er einen anderen Roboter abschießt, zerstört oder rammt.

Das Spießen funktioniert so das man entscheiden kann wie viel Schaden die Kugel machen wird. Desto mehr Schaden desto langsamer fliegt die Kugel und sie kostet dann auch mehr Energie. Man muss auch länger warten bis man wieder Schießen kann. Vor allem wenn man wenig Energie hat sollte man eine niedrige Feuerkraft wählen.

Wenn man einen anderen Roboter rammt, bekommen beide Schaden, aber der Rammende bekommt Energie zurück. Wenn man gegen eine Wand fährt bekommt man auch Schaden.

## 2.4 Spielende und Gewinner

Der Gewinner des Spiels ist nicht der letzte Überlebende sonder der mit den meisten Punkten. Punkte kann man erhalten folgendermaßen erhalten:

- Survival Score - wenn jemand stirbt bekommen alle Lebenden Punkte
- Last Survivor Bonus - wenn man der letzte Überlebende ist
- Bullet Damage - wenn man Gegner abschießt
- Bullet Damage Bonus - wenn man einen Roboter zerstört beim Abschießen
- Ram Damage - wenn man eien Gegner rammt
- Ram Damage Bonus - wenn man einen Roboter zerstört beim Rammen
- 1sts, 2nds, 3rds - wenn man erste, zweiter oder dritter wird

## 2.5 Programmierung

Die Programmierung des Roboters ist grundsätzlich Eventbasiert. Am Anfang jeder Runde wird die Methode *run()* aufgerufen. Dort wird die Initialisierung des Roboters durchgeführt. Man kann sein Aussehen also die Farbe festlegen und das grundsätzliche Verhalten des Roboters. Man kann auch in einer Endlosschleife, das Verhalten des Roboters bestimmen. In dieser Schleife kann man mit get-Methoden den Zustand des Roboters abfragen und dann das Verhalten des Roboters verändern.

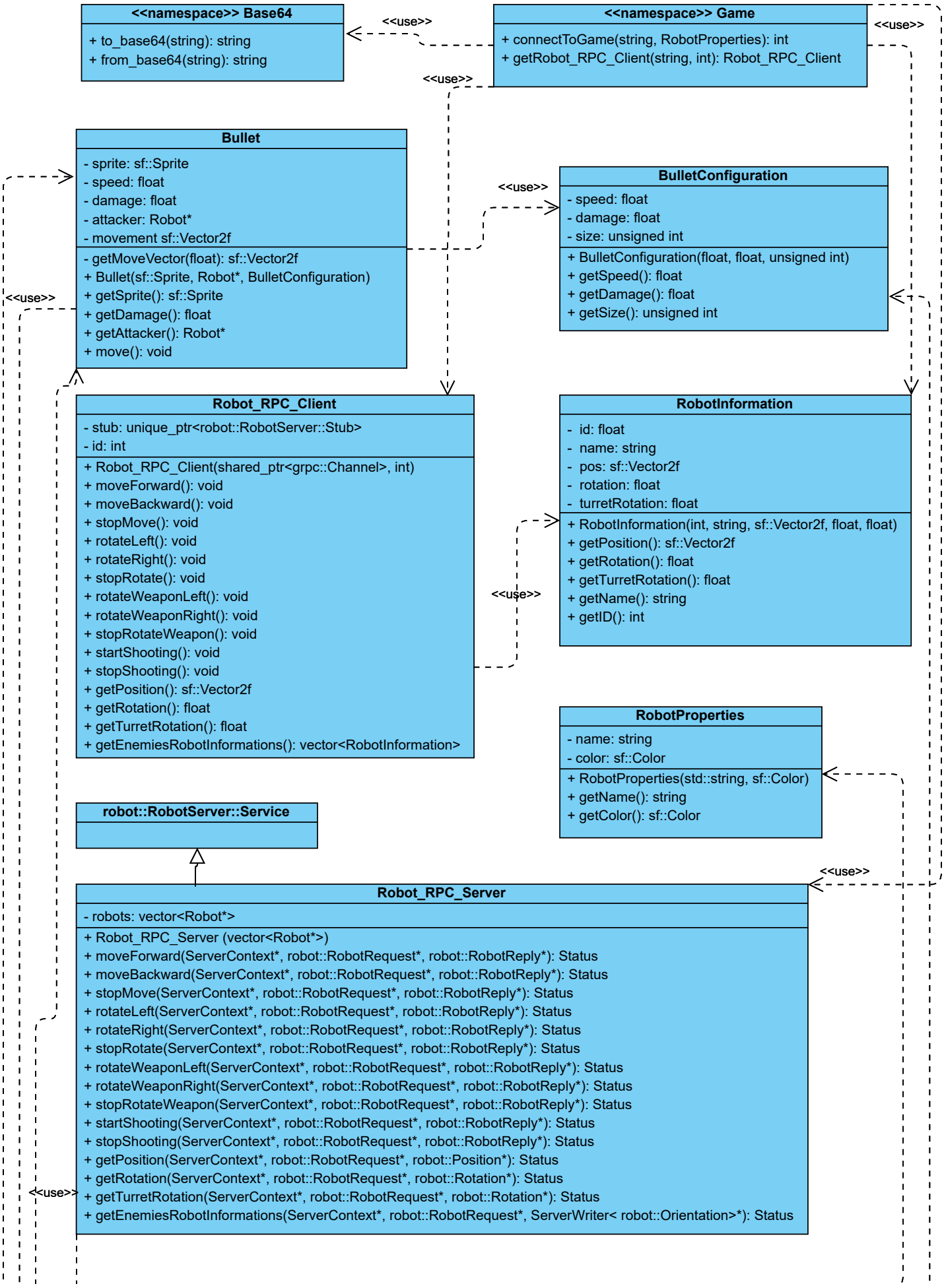
Man kann auch auf viele Events reagieren, die während des Spiels passieren. Events sind zum Beispiel onScannedRobot(event) und onHitByBullet(event) auf die man reagieren kann. Durch diese Events kann man auch auf die Endlosschleife in der Initialisierung verzichten.

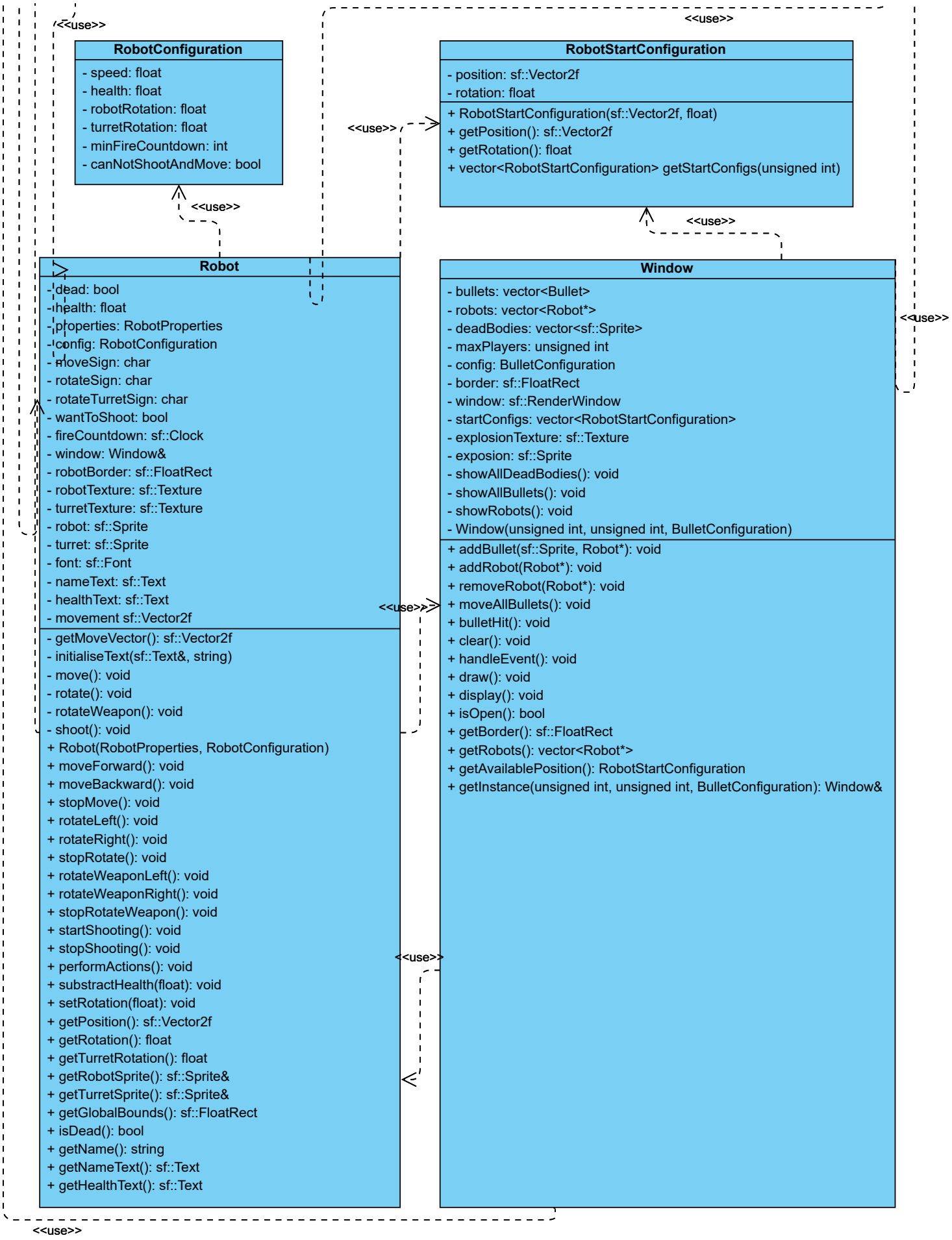
## 2.6 Spielmodi und Challenges

Es gab auch viele Robocode Wettbewerbe. In dieses sogenannten Challenges werden die besten Roboter bzw. Programmierer in einem bestimmten Spielmodus gesucht. Ein Bespeil ist die Movement Challenge 2K7.

Es gibt neben dem normalen Spielmodus auch noch viele weitere Spielmodi wie zum Beispiel 1v1, Melee, Teams Twin Duel Es gibt auch Wettbewerbe mit maximaler Codegröße namens MiniBots, MicroBots, and NanoBots.

# 3 UML


<<namespace>> Base64
+ to_base64(string): string
+ from_base64(string): string
<<use>>
<<namespace>> Game
+ connectToGame(string, RobotProperties): int
+ getRobot_RPC_Client(string, int): Robot_RPC_Client
<<use>>
<<use>>
Bullet
- sprite: sf::Sprite
- speed: float
- damage: float
- attacker: Robot*
- movement sf::Vector2f
- getMoveVector(float): sf::Vector2f
+ Bullet(sf::Sprite, Robot*, BulletConfiguration)
+ getSprite(): sf::Sprite
+ getDamage(): float
+ getAttacker(): Robot*
+ move(): void
<<use>>
BulletConfiguration
- speed: float
- damage: float
- size: unsigned int
+ BulletConfiguration(float, float, unsigned int)
+ getSpeed(): float
+ getDamage(): float
+ getSize(): unsigned int
<<use>>
<<use>>
Robot_RPC_Client
- stub: unique_ptr<robot::RobotServer::Stub>
- id: int
+ Robot_RPC_Client(shared_ptr<grpc::Channel>, int)
+ moveForward(): void
+ moveBackward(): void
+ stopMove(): void
+ rotateLeft(): void
+ rotateRight(): void
+ stopRotate(): void
+ rotateWeaponLeft(): void
+ rotateWeaponRight(): void
+ stopRotateWeapon(): void
+ startShooting(): void
+ stopShooting(): void
+ getPosition(): sf::Vector2f
+ getRotation(): float
+ getTurretRotation(): float
+ getEnemiesRobotInformations(): vector<RobotInformation>
<<use>>
RobotInformation
- id: float
- name: string
- pos: sf::Vector2f
- rotation: float
- turretRotation: float
+ RobotInformation(int, string, sf::Vector2f, float, float)
+ getPosition(): sf::Vector2f
+ getRotation(): float
+ getTurretRotation(): float
+ getName(): string
+ getID(): int
RobotProperties
- name: string
- color: sf::Color
+ RobotProperties(std::string, sf::Color)
+ getName(): string
+ getColor(): sf::Color
<<use>>
robot::RobotServer::Service
Robot_RPC_Server
- robots: vector<Robot*>
+ Robot_RPC_Server (vector<Robot*>)
+ moveForward(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ moveBackward(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ stopMove(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ rotateLeft(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ rotateRight(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ stopRotate(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ rotateWeaponLeft(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ rotateWeaponRight(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ stopRotateWeapon(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ startShooting(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ stopShooting(ServerContext*, robot::RobotRequest*, robot::RobotReply*): Status
+ getPosition(ServerContext*, robot::RobotRequest*, robot::Position*): Status
+ getRotation(ServerContext*, robot::RobotRequest*, robot::Rotation*): Status
+ getTurretRotation(ServerContext*, robot::RobotRequest*, robot::Rotation*): Status
+ getEnemiesRobotInformations(ServerContext*, robot::RobotRequest*, ServerWriter< robot::Orientation>*): Status
<<use>


RobotConfiguration
- speed: float
- health: float
- robotRotation: float
- turretRotation: float
- minFireCountdown: int
- canNotShootAndMove: bool
RobotStartConfiguration
- position: sf::Vector2f
- rotation: float
+ RobotStartConfiguration(sf::Vector2f, float)
+ getPosition(): sf::Vector2f
+ getRotation(): float
+ vector<RobotStartConfiguration> getStartConfigs(unsigned int)
Robot
- dead: bool
- health: float
- properties: RobotProperties
- config: RobotConfiguration
- moveSign: char
- rotateSign: char
- rotateTurretSign: char
- wantToShoot: bool
- fireCountdown: sf::Clock
- window: Window&
- robotBorder: sf::FloatRect
- robotTexture: sf::Texture
- turretTexture: sf::Texture
- robot: sf::Sprite
- turret: sf::Sprite
- font: sf::Font
- nameText: sf::Text
- healthText: sf::Text
- movement sf::Vector2f
- getMoveVector(): sf::Vector2f
- initialiseText(sf::Text&, string)
- move(): void
- rotate(): void
- rotateWeapon(): void
- shoot(): void
+ Robot(RobotProperties, RobotConfiguration)
+ moveForward(): void
+ moveBackward(): void
+ stopMove(): void
+ rotateLeft(): void
+ rotateRight(): void
+ stopRotate(): void
+ rotateWeaponLeft(): void
+ rotateWeaponRight(): void
+ stopRotateWeapon(): void
+ startShooting(): void
+ stopShooting(): void
+ performActions(): void
+ substractHealth(float): void
+ setRotation(float): void
+ getPosition(): sf::Vector2f
+ getRotation(): float
+ getTurretRotation(): float
+ getRobotSprite(): sf::Sprite&
+ getTurretSprite(): sf::Sprite&
+ getGlobalBounds(): sf::FloatRect
+ isDead(): bool
+ getName(): string
+ getNameText(): sf::Text
+ getHealthText(): sf::Text
Window
- bullets: vector<Bullet>
- robots: vector<Robot*>
- deadBodies: vector<sf::Sprite>
- maxPlayers: unsigned int
- config: BulletConfiguration
- border: sf::FloatRect
- window: sf::RenderWindow
- startConfigs: vector<RobotStartConfiguration>
- explosionTexture: sf::Texture
- exposion: sf::Sprite
- showAllDeadBodies(): void
- showAllBullets(): void
- showRobots(): void
- Window(unsigned int, unsigned int, BulletConfiguration)
+ addBullet(sf::Sprite, Robot*): void
+ addRobot(Robot*): void
+ removeRobot(Robot*): void
+ moveAllBullets(): void
+ bulletHit(): void
+ clear(): void
+ handleEvent(): void
+ draw(): void
+ display(): void
+ isOpen(): bool
+ getBorder(): sf::FloatRect
+ getRobots(): vector<Robot*>
+ getAvailablePosition(): RobotStartConfiguration
+ getInstance(unsigned int, unsigned int, BulletConfiguration): Window&
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>

# 4 Umsetzung

## 4.1 Spiel Idee

Die Idee meines Spieles ist es das man keine Events verwendt. Stattdessen man alle Information über den grpc Server / Cliet erhält und auch sendet. Man kann keine anderen Roboter rammen und bekommt auch keinen Schaden an Wänden. Man bekommt auch keine negative Effekt wenn man schießt und die Feuerkraft ist immer gleich. Außerdem gibt es keinen Radar. Jeder Roboter kann alles über andere Roboter abfragen. Der Rest ist sehr ähnlich zu dem original.

Die Idee meines Spiels ist das man zuerst den Server startet. Dieser hat viele Optionen die ich später noch beschreibe. Danach können sich die Clients verbinden und legen ihren Namen und die Farbe des Roboters fest. Nach dem sich alle Clients verbunden habe bekommen die Clients die Spieleinstellungen also die Roboterkonfiguration. Jetzt können sie sich mit dem Robot_RPC_Client verbinden. Ab jetzt können sie folgende Methoden aufrufen um ihren Roboter zu steuern oder Informationen über den eigenen oder die anderen Roboter zu bekommen.

```
void moveForward();
void moveBackward();
void stopMove();

void rotateLeft();
void rotateRight();
void stopRotate();

void rotateWeaponLeft();
void rotateWeaponRight();
void stopRotateWeapon();

void startShooting();
void stopShooting();

sf::Vector2f getPosition();
float getRotation();
float getTurretRotation();
std::vector<RobotInformation> getEnemiesRobotInformations();
```

Der Server funktioniert so, dass er zuerst gestartet werden muss, da er auf alle Clients wartet. Danach erstellt er die Roboter, der Clients die sich verbunden haben. Sobald sich alle verbunden haben erstellt er das Spiel mit allen Optionen und es öffnet sich auch das GUI des Spiels. Dann wird der Grpc-Server gestartet. Die Clients werden Informiert und das Spiel geht los. Die Roboter spawnen immer in einer Ecke und zeigen fast in die Mitte gerade so das sie

noch aneinander vorbeischießen. Während des Spiels werden folgende Aktionen durchgeführt:

- Roboter wird bewegt, gedreht und die Waffe gedreht
- Roboter schießt
- Bullets werden bewegt
- Es wird überprüft ob die Kugel getroffen hat
- Die GUI wird upgedated (alles wird gezeichnet)

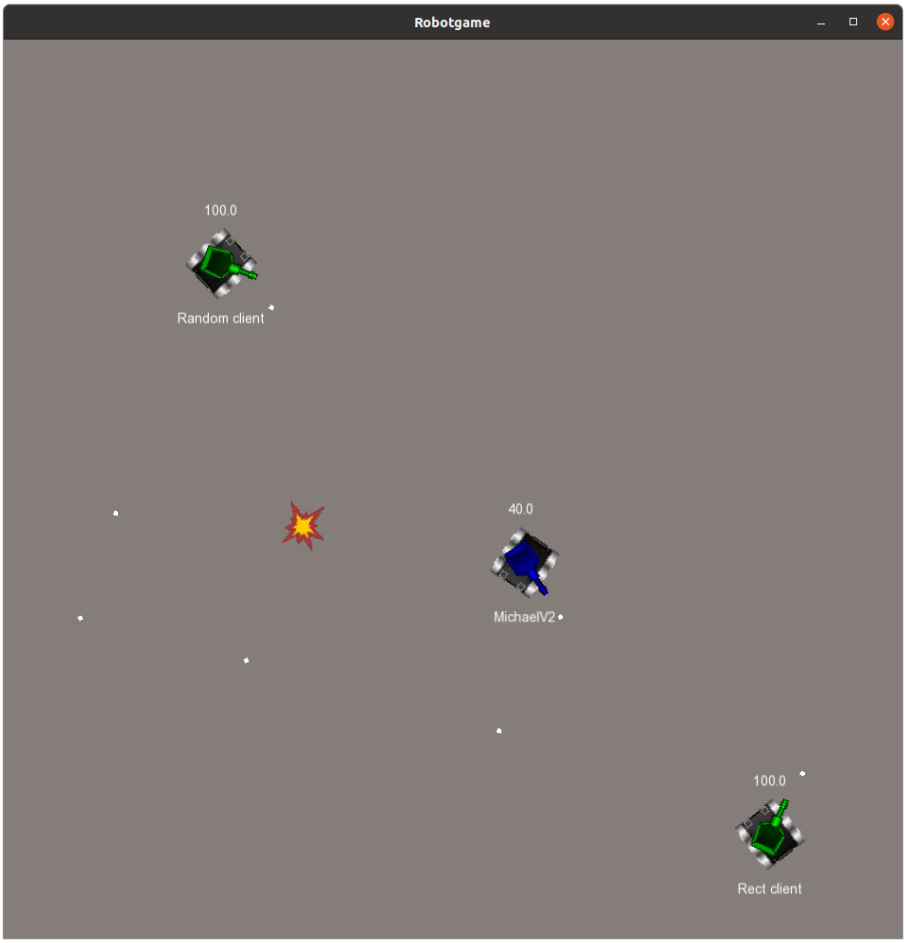



Figure 2: Screenshot aus meinem Spiel

| Option | Typ [Bereich]=Standartwert | Beschreibung |
|---|---|---|
| -h,--help | FLAG | Print this help message and exit |
| -w,--width | UINT:INT in [500 - 1500]=950 | width and height of the window |
| --bullet-speed | FLOAT:INT in [1 - 50]=6 | the speed of the bullets |
| --bullet-damage | FLOAT=20 | the damage of the bullets |
| --bullet-size | UINT:INT in [1 - 10]=3 | the size of the bullets in pixel |
| --robot-speed | FLOAT:FLOAT in [0.5 - 10]=2 | the speed of the robots |
| --health | FLOAT:POSITIVE=100 | the health of the robots |
| --robot-rotation | FLOAT:FLOAT in [0.5 - 3.5]=1 | the roation speed of the robots |
| --turret-rotation | FLOAT:FLOAT in [0.5 - 5]=1.5 | the turret roation speed of the robots |
| -f,--fire-countdown | INT:POSITIVE=500 | the time after a robot can shoot again in ms |
| -m,--max-players | UINT:INT in [2 - 4]=4 | the maximum players of the game |
| -p,--port | UINT=1113 | port to connect to |
| -s,--not-shoot-and-move | FLAG | Robot can not shoot while it is moving |
| -j,--json-config | TEXT:FILE | JSON Configuration for port, max-player, ... |

Figure 3: Serveroptionen

## 4.2 Serveroptionen

In diesem Kapitel geht es um die Einstellungsmöglichkeiten des Servers.

### 4.2.1 width

Die Breite und Höhe des Fensters der GUI kann verändert werden. Das Fenster ist immer ein Quadrat. Die Größe muss zwischen 500-1500 pixel sein.

### 4.2.2 bullet-speed

Die Geschwindigkeit des Geschosses der Waffe kann verändert werden. Die Kugel kann sehr langsam oder sehr schnell fliegen, also sozusagen direkt treffen wenn die Waffe auf einen Roboter gerichtet ist.

### 4.2.3 bullet-damage

Der Schaden der Waffe kann beliebig gewählt werden.

### 4.2.4 robot-speed

Die Geschwindigkeit mit der sich der Roboter bewegt.

#### 4.2.5 health

Die Start-Leben des Roboters.

#### 4.2.6 robot-rotation

Die Geschwindigkeit mit der sich der Roboter drehen kann.

#### 4.2.7 turret-rotation

Die Geschwindigkeit mit der sich die Waffe des Roboters drehen kann.

#### 4.2.8 fire-countdown

Die Zeit nach dem der Roboter erneut schießen kann. Eine niedrige Abklingzeit bedeutet das der Roboter sehr schnell schießt, fasst wie ein Maschinengewehr. Hohe Abklingzeit heißt das er sehr selten schießen kann und daher die Treffergenauigkeit wichtiger ist.

#### 4.2.9 max-players

Die maximale Anzahl der Spieler bzw. der Clients, die sich verbinden könnnen. Es müssen mindestens 2 Spieler sein also ein 1 gegen 1 und es können maximal 4 Spieler sein.

#### 4.2.10 port

Der Port der asio-Verbindung

#### 4.2.11 not-shoot-and-move

Eine Flag, ob der Roboter schießen kann während er sich bewegt. Das heißt er darf nicht fahren und sich nicht drehen bzw. die Waffe darf sich auch nicht drehen.

#### 4.2.12 json-config

Port, max-players und not-shoot-and-move kann

# 5 Verwendung

Die normale Verwendung ist wie folgt. Es wird zuerst der Server gestartet. Es können auch eine Vielzahl an Optionen übergeben werden wie gerade beschrieben.

Danach startet man 4 Clients jeweils in einem eigenen Terminalfenster. Ich hab 4 Beispielclients implementiert. Der Fortgeschrittenste ist domanekV2. Diese berechnet sich die kürzeste Distanz zum nähesten Roboter und schießt diesen dann ab und verfolgt ihn mit seiner Waffe (er ziehlt immer auf ihn). Der Rect-client fährt in einem großen Rechteck im Feld immer wieder im Kreis und dreht die Waffe dauerhaft im Kreis und schießt. Der Random client fährt semi-random im durch das Feld und schießt irgendwohin.

```
./robo-server
```

```
./rect-client
./random-client
./domanek
./domanekV2
```

Der einfachste Weg das Programm zu testen ist den Server mit 2 Spielern zu starten. Der test-client führt einfach jede mit einem Abstand von einer Sekunde aus und gibt dann alle Informationen über den Roboter und die Gegner, die er über den Server erhalten kann, aus. Am Anfang werden noch die Spieleinstellungen (RobotConfiguration) ausgegeben.

```
./robo-server -m 2
```

```
./test-client
./test-client
```

# 6 Code-Beispiele

Der Grpc-Server wird folgendermaßen gestartet. Es wird mit dem Robot_RPC_Server Service erstellt, der die Requests der Clients bearbeitet. Der Server läuft bis er am Ende des Spiels heruntergefahren wird.

```
Robot_RPC_Server service{robots};
grpc::ServerBuilder builder;
builder.AddListeningPort("0.0.0.0:50051", grpc::InsecureServerCredentials());
builder.RegisterService(&service);
std::unique_ptr<grpc::Server> server(builder.BuildAndStart());

std::thread grpcServerThread{[&server]{
    server->Wait();
```

```
}};
grpcServerThread.detach();
```

Der Grpc-Service von moveForward sieht folgendermaßen aus. Es wird die ID des Roboters, der sich bewegen will an den Server geschickt. Der Client speichert sich beim Initialisieren die Client-ID und sendet sie dem Server. Der Server führt die Methode beim Roboter aus, da er eine Referenz des vectors mit allen Robotern hat. Der Client bekommt keinen response, lieft aber einen Fehler wenn die Kommunikation nicht funktioniert hat. Der Server erbt von dem RobotServer::Service, dieses wird anhand des service aus dem proto file erstellt und enhält die virtuellen Methoden, die implementiert werden müssen.

Bei anderen Methoden die etwas zurückliefern wird statt der leeren RobotReply, einen Nachricht mit Inhalt als reply zurückgeliefert. z.B. reply.set_x(). Die Client-Implementierung liest das dann aus und gibt dem Client das richtig Objekt zurück.

```
//Protobuf
rpc moveForward (RobotRequest) returns (RobotReply) {}
message RobotRequest {
    int32 id = 1;
}

message RobotReply {}
```

```
#include <grpcpp/grpcpp.h>
#include "robotServer.grpc.pb.h"

//Client
std::unique_ptr<robot::RobotServer::Stub> stub;
Robot_RPC_Client(std::shared_ptr<grpc::Channel> channel, int id) :
    stub(robot::RobotServer::NewStub(channel)), id(id) {}

void Robot_RPC_Client::moveForward() {
    ClientContext context;
    RobotReply reply;
    RobotRequest request;
    request.set_id(id);

    Status status = stub->moveForward(&context, request, &reply);

    if (!status.ok()) {
        throw std::runtime_error(status.error_code() + ": " + status.error_message());
```

```
    }
}


//Server
std::vector<Robot*> robots;
Robot_RPC_Server (std::vector<Robot*>& robots) : Service(), robots(robots) {}

Status Robot_RPC_Server::moveForward(ServerContext* context, const RobotRequest* request,
                                     RobotReply* reply) {
    (void) context;
    (void) reply;

    robots[request->id()]->moveForward();

    spdlog::debug("Robot {}: moveForward", request->id());

    return Status::OK;
}
```

Der Client erstellt den Client folgendermaßen. Er übergibt den Port und die ID und ruft dann die Methode aus. moveForward() lieft nichts zurück, getPosition() liefert einen sf::Vector2f.

```
#include "robotClient.h"

Robot_RPC_Client robotClient = Game::getRobot_RPC_Client(grpcPort, id);

robotClient.moveForward();

sf::Vector2f pos{robotClient.getPosition()};
fmt::print("x: {} | y: {}\n", pos.x, pos.y);
```

Die Implementierung von move(), rotate() und rotateWeapon() ist sehr ähnlich. Ich beschreibe sie anhand von rotate(). Die Funktion wird einmal pro Durchgang aufgerufen und dreht den Roboter abhängig von rotateSign. Rotatesign kann -1, 0 und 1 sein. -1 bedeutet nach links drehen, 0 bedeutet nicht drehen und 1 bedeutet. Die Methoden rotateLeft(), rotateRight() und stopRotate() ändern lediglich den Wert des chars (char da es der kleinste Datentyp für -1, 0 und 1 ist). Diese Methoden werden auch von dem grpc-Server verwendet. Die Besonderheit von rotate() ist noch, dass sie auch die Waffe mitdreht und das überprüft werden muss ob sie die Robter berührennach dem drehen und dann kann sich nicht gedreht werden.

Am Ende der Drehung muss noch der Movement-vector neu berechnet werden, der entscheidet in welche Richtung der Roboter fährt.

shoot() hat nur 2 optionen schießen und nicht schießen und ist daher ein boolean.

```
char rotateSign{};

void Robot::rotate() {
    const float robotRotation{config.getRobotRotation()};
    robot.setRotation(robot.getRotation() + (robotRotation * rotateSign));
    turret.setRotation(turret.getRotation() + (robotRotation * rotateSign));

    if (rotateSign) {
        for(Robot* robo: window.getRobots()) {
            if (this != robo && robot.getGlobalBounds().intersects(robo->getGlobalBounds())) {
                robot.setRotation(robot.getRotation() + (robotRotation * -rotateSign));
                turret.setRotation(turret.getRotation() + (robotRotation * -rotateSign));
                break;
            }
        }

        movement = getMoveVector();
    }
}

void Robot::rotateLeft() {
    rotateSign = -1;
}

void Robot::rotateRight() {
    rotateSign = 1;
}

void Robot::stopRotate() {
    rotateSign = false;
}
```

Der movment-vector, also wohin der Roboter beim vorwärtsfahren fährt, wird einfach anhand der Rotation des Roboters und der Fahrgeschwindigkeit berechnet.

```
sf::Vector2f Robot::getMoveVector() {
    const float angle{robot.getRotation() * (float)M_PI / 180.0f};
    const float speed{config.getSpeed()};
    return sf::Vector2f{sin(angle) * speed, cos(angle) * -speed};
}
```

Etwas anzuzeigen ist in SFML einfach. Am Anfang wird die ganze Fläche gecleart. Dann werden die erstellten Sprite bzw. Text einfach an die Methode draw() übergeben. Am Ende werden alle diese Objekte dann durch die Methode display() gezeichnet.

```
window.clear();

for(Robot* robot: robots) {
    window.draw(robot->getRobotSprite());
    window.draw(robot->getTurretSprite());
    window.draw(robot->getNameText());
    window.draw(robot->getHealthText());
}

window.display();
```

# 7 Quellen

Hintergrund - Robowiki

- https://robowiki.net/wiki/Main_Page
- https://robowiki.net/wiki/Robocode/Getting_Started
- https://robowiki.net/wiki/Robocode/FAQ
- https://robowiki.net/wiki/Robocode/My_First_Robot
- https://robowiki.net/wiki/Robocode/Game_Physics
- https://robowiki.net/wiki/Robocode/Robot_Anatomy
- https://robowiki.net/wiki/Robocode/Scoring

stream-basierte Kommunikation mit asio:

- 26_tcpip_programming1 Seite 16-17