Programmer documentation

Programmer documentation

Table of Contents

List of Figures

Chapter 1. Overview of Pogamut

Pogamut is a team software project conducted by a group of students on faculty of mathematics and physics, Charles University, Prague, Czech Republic. It emerges from the necessity of the platform suitable for a fast development of virtual human-like agents.

Its main concern is to create the connection between complex virtual environment and development tools. The chosen environment is Unreal Tournament 2004 (UT2004). The platform is connected to the environment through Gamebots 2004 (GB2004) which are a server built into UT2004. This connection is tended by the Parser which transforms string-based API of GB2004 to Java objects. Those objects are accepted by the Client or more specifically the Agent which includes the library of methods and structures to facilitate the development – e.g. action primitives, memory, navigation and inventory of the agent. Over all of this spreads the IDE – NetBeans plug-in – which implements log viewers, server control, projects, manual navigation, etc.

Structure of the programming documentation is therefore following. First there is an introduction to the platform architecture and short description of main modules. Then follows chapter devoted to Gamebots 2004 which are responsible for the export of information from UT2004. It contains the detailed description of functionality of all classes, mutators and game types.

Chapter 4 is about the Parser. It outlines its main responsibilities and the idea of a remote and local parser, that relate to the optimization of the network communication which is described there as well.

Chapter 5 is dedicated to the Client. It contains an overview of internal compounds of the top-level class Agent (memory, inventory, navigation, body), summary of usefull methods and description of the mechanisms that are under those methods (like how it handles items or navigation).

Chapter 6 presents the IDE. After the description of single packages follows characterization of introspection.

Chapter 7 is devoted to auxiliary module Mediator which is responsible for flawless communication between the Parser and the Agent.

Chapter 8 is about experiments.

The last chapter contains description of all the sample bots, with all relevant information.

Chapter 2. Modules overview of Pogamut

Pogamut consists of several modules. The main modules are IDE, Client, Parser, GameBots2004. There are also two supporting modules – Mediator, Experiments, which we will cover later on.

Main modules

Each module is named after it's usage or a role.

GameBots2004 is part of UT2004 and acts as a server offering clients the service of creation and control of the bot inside UT2004. It facilitate a text-based protocol which client must implement in order to be able to control the bot inside the game. To work over the text messages is hard – therefore the module Parser exists.

Parser translates text messages from GB2004 and creates Java objects out of them. This is used by a module Client.

Client module is really a client of GB2004 using Parser to translate text messages over which it works. It's purpose is to allow a Java programmer to create Java API for creating and controlling the bot inside UT2004. The main class of this module is Agent which wraps a few other classes representing a bot inside UT2004.

Finally there is an IDE. IDE serves for implementation, debugging and evaluation of user-created bots. It uses Client's API for creating and running new instances of created agents. It also using Experiments API allowing user to evaluate the bots and to experiment with them.

Support modules

Mediator. This module handles sending messages from one side to another. It's used for sending back and forth messages between Parser and Client (more specifically between Parser and Agent).

Tools and resources

Module Experiment serves for running experiments. That means setting the UT2004 environment, creating agent instances and observing them.

Introspection provides access to some of the agent's variables at the runtime.

Sample bots are example code to demonstrate features of the platform and provide a lead to the user.

Chapter 3. Gamebots 2004

Introduction

GameBots 2004 (GB, GameBots) are a modification (mod, more information in chapter 2) for the game Unreal Tournament 2004 (UT04). GameBots are written in UnrealScript (UT04 scripting language). GameBots provide network text protocol for connecting to UT04 and controlling in-game avatars (bots). With GameBots user can control bots with text commands and at the same time, he is receiving information about the game in defined format.

GameBots main purpose is to make available rich environment of UT04 for virtual agent development by allowing easy connection to UT04 through its text protocol. More information about GB text protocol can be found in GameBots 2004 user documentation. This documentation is organized as follows: First there will be brief introduction to UnrealScript, which is followed by general overview of GB architecture and most important mechanics and second we will describe every class of GB in detail.

UnrealScript basics

UnrealScript is a scripting language created especially for the game Unreal Tournament 2004. It is somehow similar to Java or C++. The game UT04 itself is written in UnrealScript except of the engine, which is written in C++. All classes written in UnrealScript can be inherited and modified or modified directly - although it is not recommended to do it that way. UT04 supports so called game modifications (mods). Mod is some package, that modifies the game in some way without touching the game internal classes.

In this chapter I want to discuss four features of UnrealScript. Namely special construct state, then event functions, native functions and spawning.

For more information about UnrealScript visit UnrealWiki site or http://unreal.epicgames.com/UnrealScript.htm for a quick reference and examples.

States

States are groupings of functions, variables, and code which are executed only when the actor is in that state. One state is usually divided in a few sub states by simple tags. For moving between states and sub states, UnrealScript has function gotoState. Their purpose is to support AI and to facilitate programming.

Event Functions

Event function is called by engine automatically, when particular event is triggered in the environment. UnrealScript features a lot of preprepared events, which can be used when programming the bots.

Native Functions

UnrealScript features large number of so called native functions. These functions are written in C++ and are part of the game engine (we cannot see their code, or modify them). Their headers are defined in UnrealScript with special word native (we can call these functions from UnrealScript). These functions handles movement, ray tracing, etc. But mostly important, they handle spawning of objects in the game.

Spawning

Spawning of objects is handled by native function Spawn. With this function we can spawn Actor class and all classes inherited from Actor class - that means for example the majority of classes we have in GameBots. Actor is anything that moves or is visible in the game, but Actors are not limited to this. Some actors can be invisible and can be used just for handling some special situation in the game.

Overview

GameBots are built on UT04 classes, which are inherited and then modified. Thanks to this, we don't need to change the code of original classes, so we leave the code of the game intact. In this section we will provide basic GameBots overview. We will start with most important mechanics - the servers and connections, this will be followed by bots, where will be said how is the controlling of the bot handled, then we will speak about mutators and last we say something about GB ini file.

Servers and connections

UT04 features a lot of different game types. Each game type can have different rules and different goals, which have to be completed, when our bots want to win. First thing that the GameBots do is, they inherit UT04 class DeathMatch. This class handles game type DeathMatch and all other game type classes, that are supported in GB, are inherited from it. GB class *BotDeathMatch* accepts connections to defined ports by creating two servers (*BotServer* and *ControlServer* more info below). It also modifies some mechanics for the purposes of GB. Game rules remains the same (in DeathMatch this means, that player who kills pre-selected number of opponents - fraglimit - wins).

As there are two types of connections to GameBots, there are two classes that handles this. First one is the class *BotServer*, which accepts connections used for spawning and controlling the bots (one connection can spawn and control one bot). Second one is the class *ControlServer*, which accepts connections on different port than *BotServer* and provides control of game mechanics (changing map, kicking players from the server, pausing the game, etc.). Usually there will be just one control connection, although GameBots supports multiple control connections.

For every accepted connection a class will be created that will handle the connection. Connections accepted by *BotServer* are handled by class *BotConnection*, connections accepted by *ControlServer* are handled by class *ControlConnection*. All the commands that can be sent to GameBots are processed in these two classes.

Because classes *BotServer* and *ControlServer* and classes *BotConnection* and *ControlConnection* does basically the same kind of things, they are inherited from a common ancestor. *BotServer* and *ControlServer* inherit *GBServerClass* and *BotConnection* and *ControlConnection* inherit *GBClientClass*. *GBServerClass* and *GBClientClass* are abstract and consists of basic client and server functionality and some common functions for handling certain commands. The hierarchy looks as follows:

```
Object->Actor->Info->GameInfo->UnrealMPGameInfo->DeathMatch-
>BotDeathMatch
Object->Actor->Info->InternetInfo->InternetLink->TcpLink-
>GBServerClass-> BotServer, ControlServer
Object->Actor->Info->InternetInfo->InternetLink->TcpLink-
>GBClientClass-> BotConnection, ControlConnection
```

As you see UT04 features classes that handles TCP/IP connections with standard functionality of sending and receiving text and/or binary data. As was already mentioned GameBots uses pure text protocol for inbound and outbound communication.

Bots

UT04 features its own bots written in UnrealScript. The bots in UnrealScript (US) are finite-state machines (US has got special syntax that handles states). Every bot in US needs to have spawned two classes - the Controller class, which controls the bot behavior (in this class bots artificial intelligence is stored) and the Pawn class, which represents bot avatar in the game, handles animations, etc.

The bots in UT04 features nice behavior and their artificial intelligence is on higher level compared to other first person shooter games. However, in GameBots we don't want our bots to act autonomously. We want to control them by text commands. For this reason there is a class *RemoteBot*, that modifies standard UT04 bot controller class. It makes bots controllable, disables built-in AI and sends information about bot surroundings outside the game.

Class *RemoteBot* is spawned by class *BotConnection* (*BotConnection* requests spawning, the actual spawning is done in *BotDeathMatch* class). Text commands are received by class *BotConnection* and then the right methods and functions are called in the class *RemoteBot*. For the export of in-game information class *RemoteBot* uses socket maintained by class *BotConnection*. GameBots inherits xPawn class of UT04 and so created *GBxPawn* class modifies xPawn in a way we can set the bot skin (bot appearance in the game). So far just original UT04 skins are supported. List of them is in GameBots 2004 user documentation.

The hierarchy looks as follows:

```
Object->Actor->Controller->AIController->ScriptedController->     Bot-
>RemoteBot Object->Actor->Pawn->xPawn->GBxPawn
```

Mutators and the rest

GameBots features also functionality, that should facilitate development of behavior of bots by visualization of additional information in the game. For this purpose GameBots uses mutator classes and xEmitter classes. Mutators can modify (mutate) the game by adding or removing functionality, adding or removing game rules, modifying the map or even adding information to the environment. In GameBots there are two mutators - *PathMarkerMutator* and *GBHudMutator*. *PathMarkerMutator* visualize navigation points in the game by spawning a little cube over every navigation point in the map. Navigation points are spread across the map in UT04 and are used by bots for navigation in the environment.

GBHudMutator spawns a name of every navigation point in the map on the HUD in a way, that it appears the name is above the NavPoint (HUD is the screen with information about game status, we look on the game "through" this information screen). Also the player current location is displayed on the HUD.

GBHudMutator can be combined with *PathMarkerMutator* - then we can see all navigation points also with their names. Hierarchy looks like this:

```
Object->Actor->Info->Mutator->PathMarkerMutator, GBHudMutator
```

GB features one xEmitter class - it is *TraceLine* class. This class is used for the visualization of automatic ray tracing feature of GB. For every automatic ray, one green ray is spawned in the game. Hierarchy is:

```
Object->Actor->xEmitter->TraceLine
```

Rest of the classes in GameBots

The rest of the classes in GameBots are overall less important. They add three more game types *BotTeamGame* - players fight in a teams, *BotCTFGame* - players fight and teams and tries to steal a flag of opponent team and *BotDoubleDomination* - players fight in a teams and try to control two points in a map simultaneously. They override additional UT04 classes to make sure all required information from the environment are sent outside (*GBxBot*, *GBxPlayer*). They can set up our bots with some defaults (*RemoteBotInfo*) or they can help our bots to fulfill some commands (*FocusActorClass*, PauseFeed).

Ini file

GameBots have got one ini file BotAPI.ini. In this file a lot of features of GameBots can be configured. The game rules can be also modified (time limit, goal score, etc.). Each GameBots class can be configured separately here. More information can be found in GameBots 2004 user documentation.

Class description

In this chapter we will describe all GameBots classes in detail. First we will provide general overview of all used GB classes, then we will speak about game type classes, afterwards server and client classes, followed by bot and player classes, next mutator classes and last few words about the rest of the classes.

Class Overview

Here is a short description of every class in GameBots.

- *GBServerClass* - abstract server class, list of all received connections is created here
- *GBClientClass* - abstract client connection class, universal code for receiving and storing GameBots commands is here. Some functionality that is common for all connection classes is here.
- *BotServer* - class where we listen on defined port and accept bot connections
- *ControlServer* - class where we listen on defined port and accept control connections
- *BotConnection* - class that handles bot connection, parsing of bots commands is here, from here we control our remote bots.
- *ControlConnection* - class that handles control connection, parsing of control server commands is here
- *RemoteBot* - main bot class, exporting information from the game, executing commands called by *BotConnection* class, overriding default UT04 bot AI
- *RemoteBotInfo* - sets some defaults for bot, got this class from old gamebots, now it is not used in GB (may be used in future)
- *BotDeathMatch* - main GameBots game type class, *BotServer* and *ControlServer* are spawned here, *RemoteBot* class and original UT04 bots are spawned here, features default DeathMatch rules (by inheritance)
- *BotTeamGame* - adding bots to teams handled here, features default TeamGame rules (by inheritance)
- *BotCTFGame* - features default CTFGame rules (by inheritance)
- *BotDoubleDomination* - features default DoubleDomination rules (by inheritance)
- *TeamGameCopy* - copy of an original UT04 class TeamGame - because of inheritance
- *CTFGameCopy* - copy of an original UT04 class CTFGame - because of inheritance

- *xDoubleDomCopy* - copy of an original UT04 class xDoubleDom - because of inheritance
- *FocusActorClass* - helper class, so our bots can focus on location in the game
- *PauserFeed* - helper class, so we can pause the game even if no player is in it.
- *PathMarker* - class that hold StaticMesh used for NavPoints visualization
- *PathMarkerMutator* - class that spawns *PathMarker* over NavPoints in the map
- *GBHudMutator* - class that adds *GBHudInteraction* class to all players in the game
- *GBHudInteraction* - class where *GBHud* is created and where we handle key inputs and draw NavPoints grid
- *GBHud* - class that extends default HUD without modifying class with additional information
- *TraceLine* - this xEmitter class lets us spawn the visualization of automatic ray tracing
- *GBxBot* - class that overrides standard UT04 bots, so they send information to our remote bots
- *GBxPlayer* - class that overrides standard UT04 player controller class, so players can now travel through walls when spectating and send information to our bots
- *GBxPawn* - we override standard xPawn class, so we can set the bot appearance in the game

Game type classes

These classes modify original UT04 game types classes and provide them with all the functionality needed to connect to the environment through TCP/IP and spawn and control remote bots. In GameBots there are four different types of game with somewhat different rules and different goals. They are *BotDeathMatch*, *BotTeamGame*, *BotCTFGame* and *BotDoubleDomination*. The names of UT04 original classes are DeathMatch, TeamGame, CTFGame and xDoubleDom. The hierarchy of original UT04 game type classes looks like this:

```
Object->Actor->Info->GameInfo->UnrealMPGameInfo->DeathMatch->
TeamGame-> CTFGame, xDoubleDom
```

As you see, TeamGame is a child of DeathMatch and CTFGame and xDoubleDom are children of TeamGame. In GameBots we wanted to preserve this hierarchy to avoid to have same code in multiple classes. Because of this we created GameBots hierarchy, which looks like this:

```
Object->Actor->Info->GameInfo->UnrealMPGameInfo->DeathMatch->
BotDeathMatch->TeamGameCopy->BotTeamGame->CTFGameCopy, xDoubleDomCopy
```

```
Object->Actor->Info->GameInfo->UnrealMPGameInfo->DeathMatch->
BotDeathMatch->TeamGameCopy->BotTeamGame->CTFGameCopy->BotCTFGame
```

```
Object->Actor->Info->GameInfo->UnrealMPGameInfo->DeathMatch->
BotDeathMatch->TeamGameCopy->BotTeamGame->xDoubleDomCopy->
BotDoubleDomination
```

The classes with suffix "Copy" are exact copies of original UT04 classes. Thanks to this we don't have any redundant code in our GB classes. Everything important for client connections and bot spawning is handled in class *BotDeathMatch* and then is inherited by other GameBots game types (*BotTeamGame*, *BotCTFGame*, *BotDoubleDomination*). Although this might not look well on the first glance, it helps greatly to maintain the code of GB.

BotDeathMatch

This is the main GameBots class inherited from UT04 DeathMatch class. All other GameBots game type classes are inherited from *BotDeathMatch*. The *BotServer* and the *ControlServer* are created here. In this class we handle also spawning of the *RemoteBot*s in the map, spawning of the original UT04 bots in the map and special events such as new player joining or leaving the server.

The rules of *BotDeathMatch* are standard DeathMatch rules. That means the goal of the game is to survive and to kill as much opponents as possible. DeathMatch has time limit and goal score. If somebody reaches the goal score, the game ends and the player wins. Otherwise the game ends according to its time limit and the winner will be the player with highest score (with most killed opponents).

The main functions of this class are:

- function PostBeginPlay
- function *RemoteBot* Add RemoteBot
- function SpawnPawn
- function bool AddEpicBot
- function SpawnEpicBot

function PostBeginPlay	This function is automatically called after beginning of the game. Here we spawn classes *BotServer* and *ControlServer*, which are then used for accepting connections to defined ports.
function *RemoteBot* AddRemoteBot	This function spawns a class *RemoteBot*, which is a controller class for one remote bot. We set here the Id of the bot, correct number of players currently on the server and set variables affecting bot skills (accuracy, etc.). When we have our controller class ready we call SpawnPawn function, which spawns bots Pawn class, which results in the bots avatar appearing in the game.
function SpawnPawn	Function for spawning and respawning the bots Pawn (thats the visible avatar of the bot in the game). Here we set the bots Pawns peripheral vision.
function bool AddEpicBot and function SpawnEpicBot	This functions spawns Controller class and Pawn class for original UT04 bots. We needed to do this by ourselves because we wanted to set some bots variables (name, team, skill, etc.), which would be otherwise inaccessible.

BotDeathMatch has other functions - they handle sending of the game status to bots, special events as joining or leaving the server by player or bot, respawning of the bots and overriding functions that would otherwise cause the UT04 bots to join our server automatically.

In a special construct of UnrealScript - **defaultproperties** - we set what classes will be used for player controller (variable PlayerControllerClassName), if the game will be pauseable and other stuff needed for our game to start and run properly.

BotTeamGame

This class features UT04 TeamGame rules. It is inherited from class *TeamGameCopy* (here are the rules of UT04 TeamGame stored, as it is the exact copy of TeamGame class), which is inherited from *BotDeathMatch*.

TeamGame rules are: All players and bots in the game are divided into two teams and the goal is to beat the other team. The score is again the number of killed opponents. Game has its time limit and goal score.

The main function here is function bool Add*RemoteBot*ToTeam. It overrides function from *BotDeathMatch* and it assures our bot will be added to (a) desired team or (b) to some available team, when desired team is not set or cant be joined. Other functions here handle getting game status and player scores.

BotCTFGame

This class features UT04 CTFGame rules. It is inherited from *CTFGameCopy* (exact copy of UT04 CTFGame class). *CTFGameCopy* is inherited from *BotTeamGame*.

CTFGame rules are: All players and bots in the game are divided into two teams and the goal is to steal the flag owned by other team. Game has its time limit and flag limit (how many times can be ones team flag stoled until the other team wins).

Functions here handle getting game status and player scores.

BotDoubleDomination

This class features standard UT04 DoubleDomination rules. It is inherited from *xDoubleDomCopy* (exact copy of UT04 xDoubleDom class). *xDoubleDomCopy* is inherited from *BotTeamGame*.

DoubleDomination rules are: All players and bots in the game are divided into two teams. In the map there are two control points. The goal is to capture both of the control points and hold them for a few seconds. Then the team scores. Game has its time limit and goal score.

Functions here handle getting game status and player scores.

Server and client classes

Here we will describe in detail server and client classes.

GBServerClass

This is an abstract class and should never be instantiated Any connection accepting class in GameBots needs to be inherited from *GBServerClass* and spawned in the class *BotDeathMatch*.

In this class we are creating a list, where we have stored every accepted connection. US features two events - GainedChild and LostChild that are used for this.

BotServer

Class *BotServer* extends *GBServerClass*. This class is instantiated once in class *BotDeathMatch*. We have one server for the bots, but we can have multiple connections to it. The class *BotConnection* is spawned by *BotServer* for the connections.

In defaultproperties we set what class will be spawned for the connections in the variable AcceptClass. Port where we will wait for the connections is set in ListenPort variable and maximum connections in MaxConnections variable.

The only function here - **BeginPlay**, which is called automatically when the game starts, binds our connections to desired port and starts listening (waiting for new connections).

ControlServer

Class *ControlServer* extends *GBServerClass*. It is very same as the *BotServer* class, except the listening port and the classes it spawns for the connections. Here it is *ControlConnection* class.

GBClientClass

This is an abstract class and should never be instantiated From this class classes *BotConnection* and *ControlConnection* are inherited. In these two classes we process GameBots commands.

In *GBClientClass* we have some universal code for client connections. It is mainly some configure variables, variables and functions for storing and parsing incoming messages and function for sending messages outside the UT04. *GBClientClass* features also some functions used by both child classes - *BotConnection* and *ControlConnection* (exporting lists of objects in a map for instance).

For receiving there are functions **ReceivedText** and **ReceivedLine** and variables **ReceivedArgs**, **ReceivedVals** and **ReceivedData**. For parsing messages we have functions **ParseVector**, **ParseRot** and **GetArgVal**. And for sending messages we have function **SendLine**. Main function for handling text commands is **ProcessAction** function. It is called by ReceivedLine function and is not implemented in *GBClientClass*. It should be implemented by children of *GBClientClass*. In **ProcessAction** we define, what should be done when we receive certain command.

BotConnection

Class *BotConnection* extends *GBClientClass*. This class is spawned by *BotServer*. Each connection to *BotServer* has spawned its own *BotConnection* class (one *BotConnection* class handles one bot connection). In this class GameBots commands for bots (*RemoteBot*s) are processed. One *BotConnection* class can spawn one *RemoteBot* class and control one *RemoteBot* (actual spawning is handled in *BotDeathMatch* class, but functions are called from here).

This class has two states - waiting and monitoring. In waiting state, the class waits for INIT command - by this command the bot will be created in the game (bot controller and afterwards the bot pawn). In state monitoring we are periodically calling functions on *RemoteBot* that exports synchronous messages with information about the game (viz. GameBots 2004 user documentation).

In this class there are functions and structures for automatic ray tracing of the bot (LaunchRay, AutoTrace, AddDefaultRays, AddCustomRay and RemoveCustomRay). In AutoTrace function the class for ray visualization is created.

The main function here is function **ProcessAction**, where commands for *RemoteBot* are parsed and executed. For complete list of GameBots command see GameBots 2004 user documentation. More details about how the GameBots bots commands are executed can be found in this documentation in chapter *RemoteBot* class.

ControlConnection

Class *ControlConnection* extends *GBClientClass*. This class is spawned by *ControlServer*. Each connection to *ControlServer* has spawned its own *ControlConnection* class (one *ControlConnection* class handles one control connection). In this class GameBots commands for control server are processed. (in fact each instance of *ControlConnection* is independent of each other).

This class has one state running - with two sub states waiting and running. In sub state running we are periodically exporting info about location of all players in the game. This can be used for visualization of players position in mini map.

Functions in this class handle exporting of information about the map and about the game. Main function here is function **ProcessAction**, where commands are parsed and executed. For more information about *ControlServer* command see GameBots 2004 user documentation.

Bot and player classes

Here we will speak about classes for our bots and players. We will start with most important class *RemoteBot*, which will be followed by *GBxPlayer* class, *GBxBot* class and *GBxPawn* class.

RemoteBot

Class *RemoteBot* extends UT04 class Bot. The bots in UnrealScript are finite-state machines, that react to events in the game (events are special types of functions called by engine, when particular situation appears in the game). In *RemoteBot* class we override original bots states, functions and events, so no autonomous behavior will be executed.

Class *RemoteBot* has got three states. It is StartUp state, Dead state and GameEnded state (we override also two more states - MoveToGoal and TakeHit, but it seems they are never called in our class). StartUp state is the main state here. In StartUp state bot movement and turning is executed. Every time the bot is killed, we end up in the Dead state. From Dead state we respawn the bot. State GameEnded is active, when the game ends - because of time limit or reached goal score. There are few seconds, when the winner is showed and then the map is changed.

Functions and events in this class controls the bot (shooting, aiming) and exports information through *BotConnection* class (exporting events in the game, checking surroundings periodically, etc.). Now we will speak about bot variables and control basics.

Variables

Important inherited variable in this class is Pawn. Pawn represents our bots avatar in the game. When we want to get location or rotation of our bot, we need to look at our bots Pawn variables. Other important variables are FocalPoint (vector), Focus (actor) and Target (actor). These variables have got influence on bots turning and shooting. See below.

Bot movement

For moving the bot we call native engine latent functions **MoveTo** and **MoveToward**. We support them with location or object where to go (vector in case of MoveTo, Actor in case of MoveToward), what actor we want to focus on and if we should use walking speed. Functions MoveTo and MoveToward resets FocalPoint variable to location we supported them with (if we leave the focus input of these functions unspecified).

Bot turning

Bot turning is done automatically by the engine. Only thing we need to do is to set FocalPoint variable. We can also turn to Actors, for this we set Focus variable. If we call function FinishRotation afterwards, the function will end when we will be facing the spot. If we have some Actor targeted, we will continue to turn to face him if he moves (if no other commands will be received by bot).

Bot shooting

For shooting these functions are important: **RemoteFireWeapon**, **WeaponFireAgain**, **StopFiring** and **AdjustAim**. With RemoteFireWeapon our weapon starts to fire. BUT! It would fire just one shot if we wont have function WeaponFireAgain overridden to return true. Now our weapon will continue to fire until we call StopFiring function (called by STOPSHOOT), or run out of ammo, or our bot dies. Function AdjustAim does aiming for us. It is called by the firing routines. This function provides aiming correction.

In AdjustAim function: We are firing on Target (Actor class) - that is inherited variable. If Target is not set, it is get from Enemy variable - also inherited, also Actor class. We made a slight change to code in AdjustAim function, so it is now possible to fire even on location (we set FireSpot variable in AdjustAim to our location target).

So when we want to fire on someone we set Target variable. If we want to shoot on the location, we set FocalPoint variable (and myFocalPoint variable, as the FocalPoint is changed by MoveTo and MoveToward functions) and set Target and Enemy variables to None.

Reachability and paths

For reachability information and path finding we use native engine functions. Namely **actorReachable**, **pointReachable** and **FindPathTo**. Functions are self-explanatory, FindPathTo function fills array RouteCache with ordered list of NavPoints we should follow, when we want to get to our goal.

CanSee and LineOfSightTo

These two functions are preprepared functions for getting information if we can see some actor and if some line of sight exists to desired location or point from our bot. CanSee is influenced by Pawns peripheral vision.

GBxPlayer

This class extends standard UT04 player controller class - in this class commands from player are processed and are executed in a game. These commands are keyboard and mouse inputs. In this class we have code for player moving with his avatar, for player moving as the spectator and so on.

We changed a code for spectating a bit, so now it is possible for spectators to go through walls in the game. Also we modified a bit functions for sending messages to other players in a game so now also *RemoteBot*s can receive this messages (functions ServerSay, TeamSay). Moreover HIT message is now sent to bots properly also from players (function NotifyTakeHit).

In our GameBots game types we spawn *GBxPlayer* class instead of standard xPlayer class for UT04 players. It is set in game types variable PlayerControllerClassName.

GBxBot

This class extends standard UT04 bot class. We override here one function, so our bots can receive HIT message also from UT04 bots properly (function NotifyTakeHit). In our GameBots game types we spawn *GBxBot* class instead of standard xBot class for original UT04 bots.

GBxPawn

This class overrides standard UT2004 xPawn class. This class represents bot virtual body. That means - bots appearance, bots animations and the way of handling movement etc. Bot skins are loaded here.

In GameBots we override **Setup** function of xPawn class, so we make possible to set bot skin (bot appearance) in the game.

RemoteBotInfo

This class is taken from the old GameBots for UT2000 and is spawned in *BotDeathMatch* at the beginning of the game. In UT2000 it was probably needed for configuring some variables needed by engine for spawning the bot correctly. In GB04 we don't use this class at this time.

In this class the skins, bot difficulty, bot names, accuracy and so on are configured.

Mutator classes

Here we discuss mutator classes in detail.

PathMarkerMutator

This Mutator spawns at the beginning of the game on every navigation point without inventory item PathMaker class, which will visualize the otherwise invisible NavPoint.

PathMarker

This class is a part of NavPoint visualization. StaticMesh used for the visualization is loaded by exec command in this class. After spawning of this class in the map on the desired location, the object specified in the variables of *PathMarker* class appears in the game.

GBHudMutator

This Mutator causes, that for every player in the game class *GBHudInteraction* will be spawned. *GBHudInteraction* class is necessary for adding additional functionality for player HUD and for the GameBots key commands.

GBHudInteraction

GBHudInteraction extends Interaction class, which is special type of UT04 class for the purposes of adding additional functionality to key events, player HUDs (and so on) without modifying the classes, that are normally responsible for this.

In this class we create special GameBots HUD for players (*GBHud* class, we call its function **PostRender** from here - without this we wouldn't be able to draw on the HUD) and we catch GameBots key commands here. Also we draw NavPoints grid (NavPoints reachability graph) in this class (function DrawNavPointsGrid). *GBHud* and NavPoints grid is controlled by key events (function KeyEvent).

GBHud

GBHud extends HudBase (original UT04 HUD base class). We draw here NavPoints names above the NavPoints in the game and current location of the player in UT units. This class is controlled by key events (they are processed by *GBHudInteraction* class).

TraceLine

This class is based on xEmitter class. xEmitter classes are used in UnrealScript for creating various visual effects, that can be later seen in the game. *TraceLine* class spawns beam effect, that is used for the visualization of automatic ray tracing. Each automatic ray has got one beam associated with it. Colour of the beams is green (in the future we plan to implement changing colours of the beams according to whether the ray hits something or not).

Most important function here is **Tick**. This function is called regularly many times per second by the engine. In this function we are creating the desired beam and here we also change its location according to bot movement. Function Tick is simulated that means, that it is called not only on game server, but also on all the clients. If it were not simulated, we couldn't see the rays on the clients - that means spectators connected to our game would be unable to see the rays. Also all the other functions here are simulated.

The replication construct defines some additional variables that are replicated to the client. Otherwise, the variables wouldn't change their value on the clients - just on the server.

Other classes

The rest of the classes that did not fall into preceding categories.

FocusActorClass

This class extends UT04 Actor class (Actor is an abstract class and cannot be spawned, but we want to spawn it for special purpose). Normally bots in UT04 cannot focus on a location1, when they are running toward location2, which is different. They can focus on some other Actor, but not on location. For GameBots purposes we wanted to make our bots able to focus also on the location. For this we use *FocusActorClass*. We set the location of *FocusActorClass* to desired location and then we set our focus to *FocusActorClass*. *FocusActorClass* is normally invisible, but can be made visible, so we can see the spot, the bot is currently heading or looking. Position of *FocusActorClass* is updated in Tick function of *RemoteBot* class, if we have set it to be visible (otherwise it is used just for setting the focus on location).

PauserFeed

This class is used to pause the game even without no players in it. The problem is that UnrealScript requires us to support some PlayerReplicationInfo class, when we want to pause the game. *PauserFeed* class inherits PlayerReplicationInfo class, so it can be used for this purpose. Class is created at the start of the game.

Chapter 4. Parser module

Overview

Parser is a module of Pogamut, it is middleware between GameBots2004 and Client. It is used by the Client to communicate with GameBots2004. Its purpose is to simplify handling messages from GameBots and to lower network bandwidth. Simplification is done by translating messages from text messages (ASCII format, sent by GameBots) to Java objects (MessageObject class). Objects are then sent to the Client (another module of Pogamut2, where AI is). Parser is lowering data bandwidth by transmitting only informations that has changed like position of the bot, visibility etc., not the position of items that can't move - *delta messages*.

There are two different kinds of view on a Parser. The first is Parser as a Java class. Strictly speaking Parser as a class just translates text messages to Java objects. It does not contain any mechanisms to receive or to send messages via TCP/IP for instance. Sending messages are done by Mediator (see chapter MEDIATOR).

The second view is Parser as a module. The Parser module covers both parsing the text messages and the usage of Mediators for sending/receiving messages to/from the Client.

The Parser itself is implemented using JFlex, specification of the JFlex grammar can be found in the file bot_msg.flex file.

Class overview

Almost everything is in package *cz.cuni.pogamut.Parser*. The Parser class also implements two interfaces from package *cz.cuni.pogamut.communication*. The Parser returns messages that are instances of class from package *cz.cuni.pogamut.MessageObjects*. There are class for each type of message the GameBots2004 can send in this package.

Package cz.cuni.pogamut.Parser

class GameBotConnection - wrapper for GameBots2004 socket
class Parser - translates text messages into Java objects
class ParserConnection - wrapper for the RemoteParser socket, used by Client (class Agent) when RemoteParser is used
class RemoteParser - class wrapping RemoteParserServer, allowing to run it as a program
class RemoteParserServer - acts as a server for Client (default port 4000), which connects to the GB2004 and applying delta compression to messages
class UnrealIDMap - class which maps Unreal ID strings to a numbers which are used to identify objects in the game thus saving the network bandwidth
class Yylex - generated class by Jflex (Java version of Flex) from bot_msg.flex file

Package cz.cuni.pogamut.communication

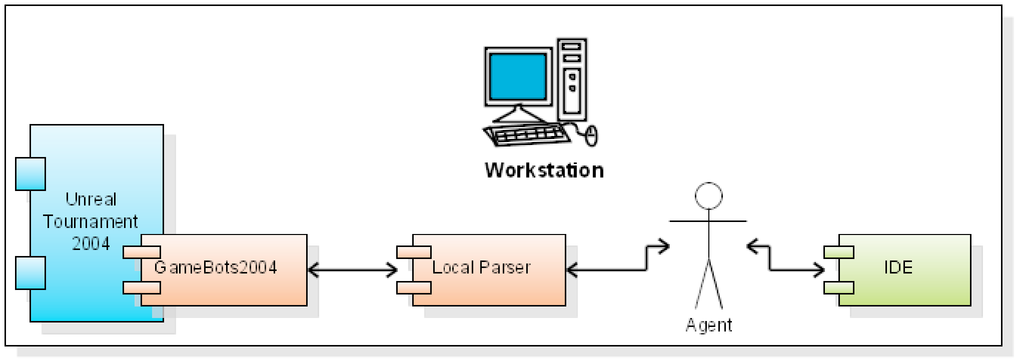
class MediatorParserInterface- interface for receiving messages from Parser for the Client
class MediatorGBInterface - interface for sending messages to the GameBots2004

Usage

There are two types of usage of the Parser module – Local Parser and Remote Parser.

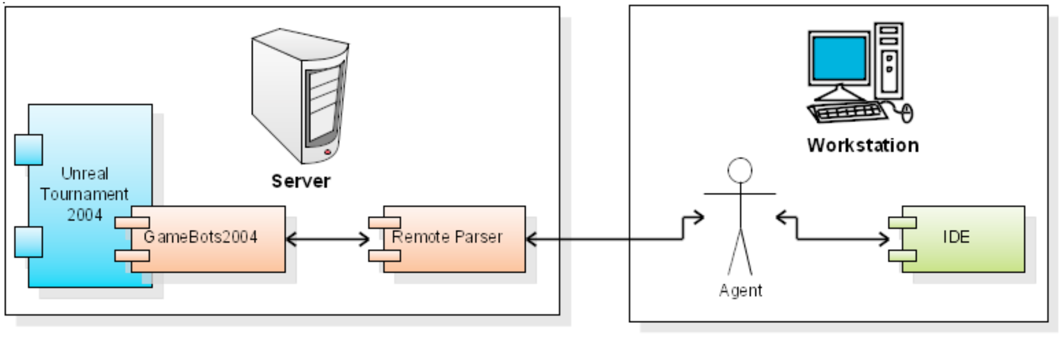
The Local Parser is meant to be run on the same machine as the Client (specifically Agent class). It should be used when the UT2004 server is running on the same machine as the Agent because there is no gain in delta compression in such a case (see Figure 4.1, “Local parser - everything is run on one machine”).

Figure 4.1. Local parser - everything is run on one machine



The Remote parser is meant to be run on the machine as the UT2004 server that is different from the machine where the Client runs. Therefore the communication between Client and Remote Parser goes via TCP/IP. See Figure 4.2, “Remote parser – parser runs on the different machine then the Agent itself”. If the user is running UT2004 server on different machine then it's advised to prefer Remote Parser over Local Parser to lower the data bandwidth due to the delta compression.

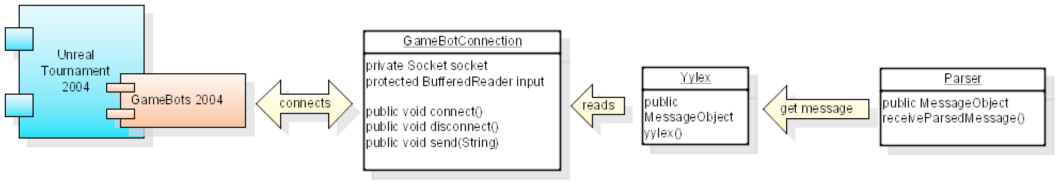
Figure 4.2. Remote parser – parser runs on the different machine then the Agent itself



Parser schema

Each Parser instance is configured by GameBotConnection that is used for receiving text messages from GB2004. See Figure 4.3, “Parser schema”. During the construction of the Parser an Yylex instance is also created that uses GameBotConnection's socket (BufferedReader input) to receive text messages from GameBots2004.

Figure 4.3. Parser schema



The Parser class implements interface MediatorParserInterface that contains method receiveParsedMessage(). Whenever that method is called by the Mediator, the text message handling routine is iterated through. The final product of the routine is MessageObject that is returned. This scenario is different when message END is received. This will be explained later on.

Text messages types

Text messages from GameBots2004 are of two types. Synchronous and asynchronous (for more informations see chapter GAMEBOTS). Asynchronous messages came at random and can't be delta-ed as they don't have any ID and are usually quite unique. Synchronous messages are always delta-ed before they are returned by the Parser.

Class MessageObject and Java message types

There is a class for each message the GameBots2004 protocol defines. All these classes are descendants of MessageObject class, which defines two key methods *diff()* and *update()*.

Diff() method takes as an argument a message object of the same class and has to tell whether the argument differs from object or not, also nullify every property which has the same as an argument.

Update() method writes also takes as an argument message object of the same class and writes all non-null properties to current object (used by the Client).

Those two methods realize the delta compression that will be explained later on.

Unreal ID, int ID

Each synchronous message also has an ID attribute. This attribute is filled by GameBots2004 with string that supplies UT2004. This Unreal ID string is unique for each object existing in the game. This ID is quite long and it would waste the bandwidth if sent every time with the message. In spite of this, there has to exist some ID for every object in the game so the Client can recognize it. Therefor the class UnrealIDMap exists. It assigns an unique number to each Unreal ID string so the messages will be processed according to int ID not the string ID thus saving the communication bandwidth.

Synchronous message batch, delta messages

The key concept of the GameBots2004 protocol are synchronous message batches, which comes in frequency about 10 batches per second. Information about what bot sees (navigation points, items, other players, etc.) comes in batches. Each batch can be viewed as a camera picture of what bot sees in a specific time. The Parser always have two batches stored. The last one and the current one. The Parser needs to know, which messages is the Client aware of, allowing him to create delta messages and notice the Client in case that something disappear from it's field of view.

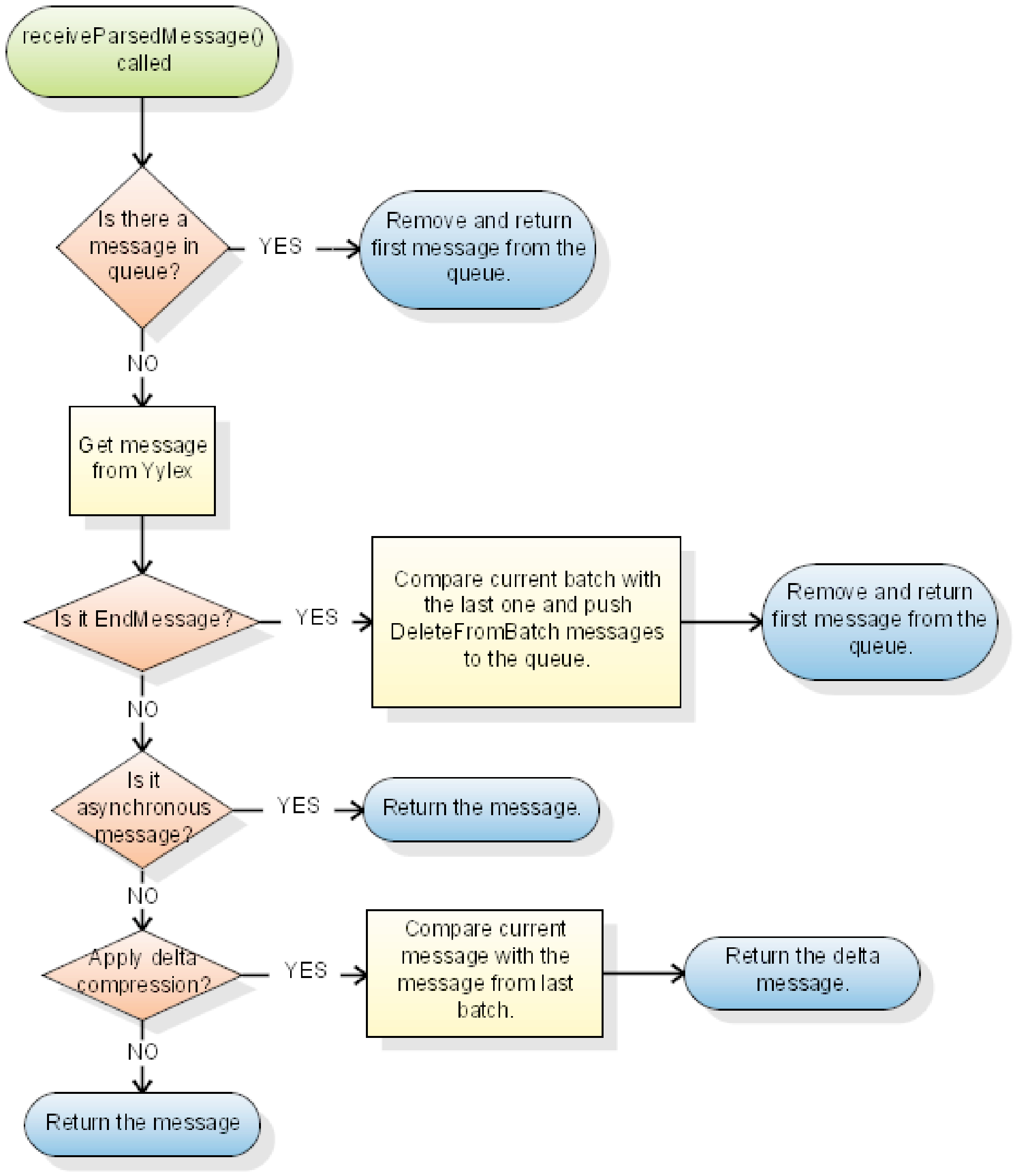
Usually when bot sees some player (for instance) it is visible to the bot for some time. That means the player stays in the bot's field of view for several batches. The player's name usually don't change therefore it's no use to send it's name over and over again. This is a part where delta messages steps in. The Parser takes the message (in this case Player object) and updates the message so it contains only information which changes using *diff()* method and returns that. In case of Player message it will probably be player's location.

The batch's end is marked with EndMessage. Upon receiving such a message – the Parser must check the current batch against the last batch. For each message in last batch that is not present in the current one, the Parser must send the DeleteFromBatch message to the Client so the information about what the Client sees is correct.

There is a possibility that many objects disappeared from the Client's field of view therefore the EndMessage should produce many DeleteFromBatch messages. In this case the Parser saves those messages to a queue and when asked for another message it takes message from the queue rather then calling Ylex.

Text message handling routine

Figure 4.4. Message handling flow chart


receiveParsedMessage() called
Is there a message in queue?
YES
Remove and return first message from the queue.
NO
Get message from Yylex
Is it EndMessage?
YES
Compare current batch with the last one and push DeleteFromBatch messages to the queue.
Remove and return first message from the queue.
NO
Is it asynchronous message?
YES
Return the message.
NO
Apply delta compression?
YES
Compare current message with the message from last batch.
Return the delta message.
NO
Return the message

The handling begins with the question if there is a message in a queue waiting for delivery (produced by some EndMessage in the past). If so, remove one message from the queue and return it. If not, call Yylex. Yylex class then reads and parses one text message from GameBots2004. It returns MessageObject to the

Parser and the Parser will process the message according to it's type. Asynchronous messages are returned immediately and synchronous messages are delta-ed and stored to a current batch.

If EndMessage arrives, compare current batch with last one and produce DeleteFromBatch messages if necessary. Also write current batch as the last one and begin new one.

The only difference in this behavior is when the list of map's NavPoints or items is received. In this case there is no need to create delta messages and the message is returned immediately.

End of communication

The communication can end either normally, when the MapFinished message is received from GameBots2004 or abnormally - network communication problem, socket closed on the remote side, etc.

When the Client receives MapFinished it has to terminate itself and shouldn't be requesting any other messages from Parser.

When an error occurs, which means the data can't be read from the socket by Yylex a message Disconnected is created and sent to the Client. The Client has to terminate itself after that.

Embedding Parser to a Client

The Parser as described above can be used by a Client as a local instance (Local Parser), or can be run on a different machine as a Remote Parser. These two kinds of usage Local Parser / Remote Parser reflects two scenarios.

The first scenario is when everything is run on one machine. Everything means UT2004 + GameBots2004 and the Client. In this case there is no need to run the parser as a different process and can be embedded right into the Client.

The second scenario is when UT2004 + GameBots2004 is running on a different machine (machine A) then the Client (machine B). In this case the Client needs to connect to a different machine using network protocols. If the Client will use Local Parser it will waste the network bandwidth as GameBots2004 always sends everything (doesn't do delta compression). Thus is advised to run RemoteParser process on machine A, which the Client will connect to.

JFlex grammar, bot_msg.flex file

The file *bot_msg.flex* contains specification for the JFlex that tells how to parse the text messages from GameBots. JFlex takes this file and transforms it into finite-state machine that parses incoming strings. The file contains support functions, state definitions and respective actions (Java code) that should be done when a part of the message is recognized.

The file was created according to the GameBots2004 API. It parses all messages that are specified there. For every message type exists a special Java class that encapsulate it. If the message is not simple - that means if it contains attributes - then a special state inside JFlex is created to handle it. For instance, the PLR GameBots2004 message looks like this: *PLR {UnrealID RemoteBot2.bot} {Position x,y,z}* ... So inside the flex file you will find the definition of the state STATE_PLR inside which the attributes are parsed.

For more information about the JFlex grammar please refer to the JFlex manual [http://jflex.de/manual.html]

Adding new message type

Because the message is processed by few modules of the Pogamut, you have to alter several places to add new message type.

First you have to add new message to GameBots2004 and push it into UT2004 server (for more information about this topic refer to GameBots documentation).

Then you have to create a new class in package cz.cuni.pogamut.MessageObjects that is derived from class MessageObject and implements interface Serializable. Define properties of class (in most cases are just taken from GameBots message). The class has to implement few methods:

- Constructor without parameters has to call super() (constructor of MessageObject) with type of message, every message has a type. Add new type of message to the enum MessageType. There are others constructors of MessageObject accepting other parameters as well, but in most cases we don't know the other parameters.
- toString() method should be overridden, since we print these messages for debugging.
- In case the message represents game entity (is synchronous message):
  - Override method hasID() to return true and add it to the constructor of SynchronousMessages class
  - Override methods update() and diff(), for details see JavaDoc of class MessageObject
  - Class has to implement interface Clonable and override method clone to return deep copy
- bot_msg.flex has to implement support for new message from GB (for more details about bot_msg.flex see JFlex documentation):
  - to state list of states add another state (e.g. MSG_FLOWER)
  - add to state YYINITIAL pattern for the name of message and to the code for the pattern put initialization of message handling. In most cases it means

    ```
    actObj = new FlowerMessage(); state_go(MSG_FLOWER);
    ```
  - in state MSG_FLOWER parse rest of message
  - use JFlex to create new yylex.java file from bot_msg.flex, replace current one and run parser
  - in new yylex.java find the row 'return YYEOF;' and change it to 'return null;', otherwise you will get compilation error

The last point will be handling of messages in the Client module, but that depends on a kind of message you're adding.

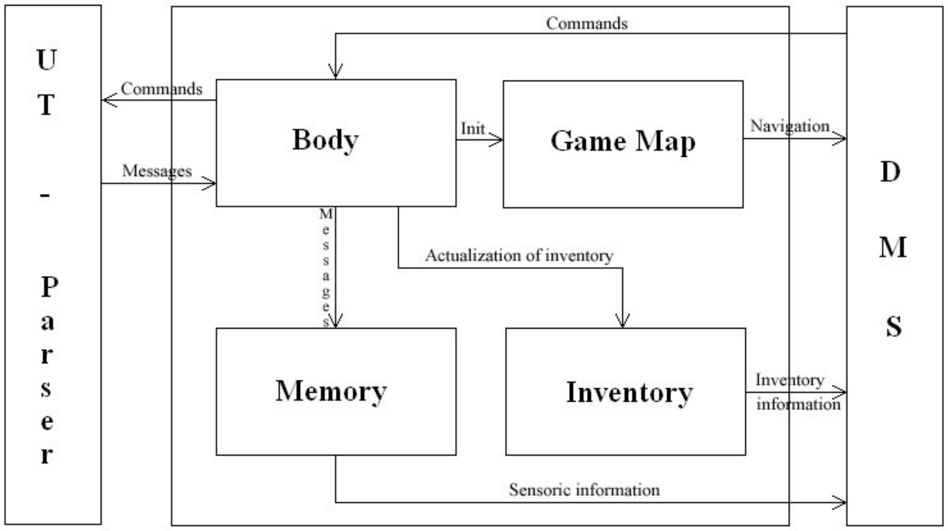
Chapter 5. Client

Architecture

Client is composed of Body, Memory, Inventory and Game Map modules. Each module is responsible for a part of services which are provided by Client. Those services are:

- memory – storage of sensory data
- inventory – item related issues
- notion of map – navigation
- commands – for the control of agent body in the environment

Figure 5.1. Client architecture



Now there will be a brief resume of functions of the modules.

Body (*AgentBody*) is a bridge between Client itself and Unreal Tournament 2004 (UT) (ergo Parser). It processes messages (Java objects) from Parser and fire events to notify listeners (at least Memory, Inventory). It implements interface *Commands* and therefore enables user to call predefined methods for agent control.

Memory (*AgentMemory*) contains *History*, which is the storage of all sensory data coming from Body. Memory implements interfaces *WorldView*, *RecentMemory*, *Knowledge* and *Inventory*, hence it includes all predefined methods for decision making system (DMS) to obtain sensory information (e. g. visible navigation points, players, weapons, etc.) and for work with inventory.

Inventory (*AgentInventory*) is basically an array of weapons. Agent in UT does not have any inventory and thus does not know about other weapons he picked up. Inventory gives the agent information about weapons he possesses, about available ammunition etc.

Game map (*GameMap*) is a module for navigation. It operates over a map representation and built-in A* algorithm. Map representation is initialized at the beginning of the simulation. Game map provides methods for obtaining path to desired location, for run along specified objects in the map (e. g. navigation points, weapons), etc.

Description of the communication with Parser

Communication is handled by the Body. It starts with the handshake with the Gamebots server. After that the Gamebots start sending messages – strings according to Gamebots API. Parser parses those messages to message objects – *MessageObject* descendants. Those messages are delta compressed and sent to Client. Client (Body) receives them, complete the compressed ones and fire event. Registered listeners are notified about the message and perform designated actions.

More detailed descriptions of the communication follows.

Handshake with Gamebots

When the Agent requests connection to parser, parser creates new copy of itself and this parser is assigned to the Agent. Then it tries to establish connection with Gamebots. When ready, Gamebots send `HELLO` to Client. It responds `READY`. After that the Gamebots sends all navigation points and inventory items located in the location which is on the Unreal server. Those information are used for initialization of module Map (necessary for proper function of built-in A*) and Knowledge in Memory (known items, navigation points).

After that Gamebots sends the message which concludes game information (like name of the map, limit of points to win the match, type of the game etc.). After this message user can initialize agent’s logic and after that Agent sends `INIT` which indicates to Gamebots that they can spawn the avatar to the environment and start sending messages about agent’s surroundings.

Messages from Gamebots

All information about the information carried by messages from Gamebots can be found in Gamebots documentation. There are two main categories: synchronous and asynchronous messages. Synchronous messages are sent approximately every 100 ms and cover environment information (navigation points, players, etc.). Asynchronous messages are expressing events (damage, noises, pickup of an item, kill, etc.). Each message type sent from Parser to Client has its object type (subclass of *MessageObject*).

Optimization of network communication

Synchronous messages are delta compressed. They include information that client can remember and therefore Parser doesn't have to send. Those information are omitted when the message is send again.

But that is not the only optimization of the network communication. The other is again focused on synchronous messages. Synchronous messages create batches. They are bordered by `BEGIN`, `END` messages. Information in neighboring batches are quite similar and therefore it is not necessary to send all messages in them over again. Parser keeps its own copy of actual batch and than sends only updated messages (properties changed or are new). At the end of each batch Parser checks whether some of the messages disappeared and send message to notify Client about it.

And the last optimization is that every object (message) with unique UT identifier got assigned unique integer identifier and therefore the delta compression can save some network capacity using couple Bytes for integer instead of tens of Bytes for long string which is used as identifier in UT.

Example of optimization: navigation point is originally in GB a string with about 90 characters. In our version of communication Parser sends object with 3 doubles (for location of the navigation point), String with identifier – about 20 characters, some Boolean values and the integer identifier. The next time the same navigation point is sent, Parser sends only the integer identifier and Boolean values (those stands for visibility and reach ability of the navigation point).

Commands

Commands are messages sent by Client to Gamebots, which specify what the agent will do. They are represented by simple strings wrapped by an object and are composed in Body according to specified parameters. As there are not many commands sent per second, they are not optimized at all.

When a command is sent an event is fired, so anyone can observe the flow of commands (used in IDE).

Detailed communication description

Parser – Client communication:

- Parser creates full MessageObject from GB string message and takes old MessageObject of the same UnrealID from its database of known objects (KnownObjects: HashMap – UnrealID => MessageObject). Then it uses method Diff, which makes delta object from full one (set all unchanged properties to null). If there is anything different from old object, it sends it to the Client.
- Client receives a message, which is usually not completely initialized. There are three possibilities:
    - Message doesn't have ID (messages which are not compressed, mostly asynchronous messages) – pass it along without any processing
    - Message has ID, but it is new (not in KnownObjects) – add it to the them and pass it along as well
    - Message has ID and it has got old one in KnownObjects – use Update method of MessageObject (called on old one, accepts new as parameter), this will update old object. Than send the updated object to further processing (like processing items). Than pass it along – fire event.

Sending commands to Parser:

- Methods for creation of all commands specified in Gamebots API are parametrized only by objects which are easily accessible using methods of AgentMemory, AgentInventory, etc.
- Those methods compose proper string, which is then send to Parser which propagate it to Gamebots and to the server of the game.

Figure 5.2. Command example

```
Void runToLocation(Triple location){
      sendCommand(“RunTo {Location ” + location.toString() + “}”;
}
```

Communication states

During the handshake with GameBots2004 the agent goes through a different communication states that reflects the flow of the communication (namely handshake) with GameBots2004.

There are three types of states: Handshaking, Running, Final

Handshaking communication states:

- START - the Agent object has been just created and connected to GameBots2004
- MAP_RECEVIVE_NFO - the Agent recognize the remote side as GameBots2004 and expecting NFO message with basic informations about the current map (name, gametype, time limit, etc.)
- MAP_RECEIVE_NAVPOINTS_EXPECTED - NFO message has been received and now the Agent is expecting the beginning of the NavPoints list that are placed in current map as well as their connection informations (edges between them)
- MAP_RECEIVING_NAVPOINTS - GameBots2004 are sending informations about the NavPoints in the map, the Agent is storing them into the AgentMemory
- MAP_RECEIVE_ITEMS_EXPECTED - NavPoints were received, now the Agent expects the list of items that are available in the map and their respective locations
- MAP_RECEIVING_ITEMS - GameBots2004 are sending informations about the items in the map, the Agent is storing them into the AgentMemory
- AWAITING_LOGIC - handshake with GameBots2004 is complete, now we're waiting for logic to initialize, namely the *postPrepareAgent()* method is called, that can be altered by the user of the bot (hook for initialization of planners, engines, etc.

Running communication states:

- BOT_RUNNING - server is running and the bot in the UT2004 as well
- PAUSED - the bot is paused - either because the server is paused or the user has paused the bot using RemoteControl panel inside IDE

Final communication states:

- TERMINATED - the Agent was terminated using IDE
- FAIL - error during the handshaking phase occurred, probably due to the wrong GameBots2004 protocol or the exception
- EXCEPTION - an exception occurred during the execution of the Agent's logic
- MAP_FINISHED - the map on the server has finished and the connection has been lost

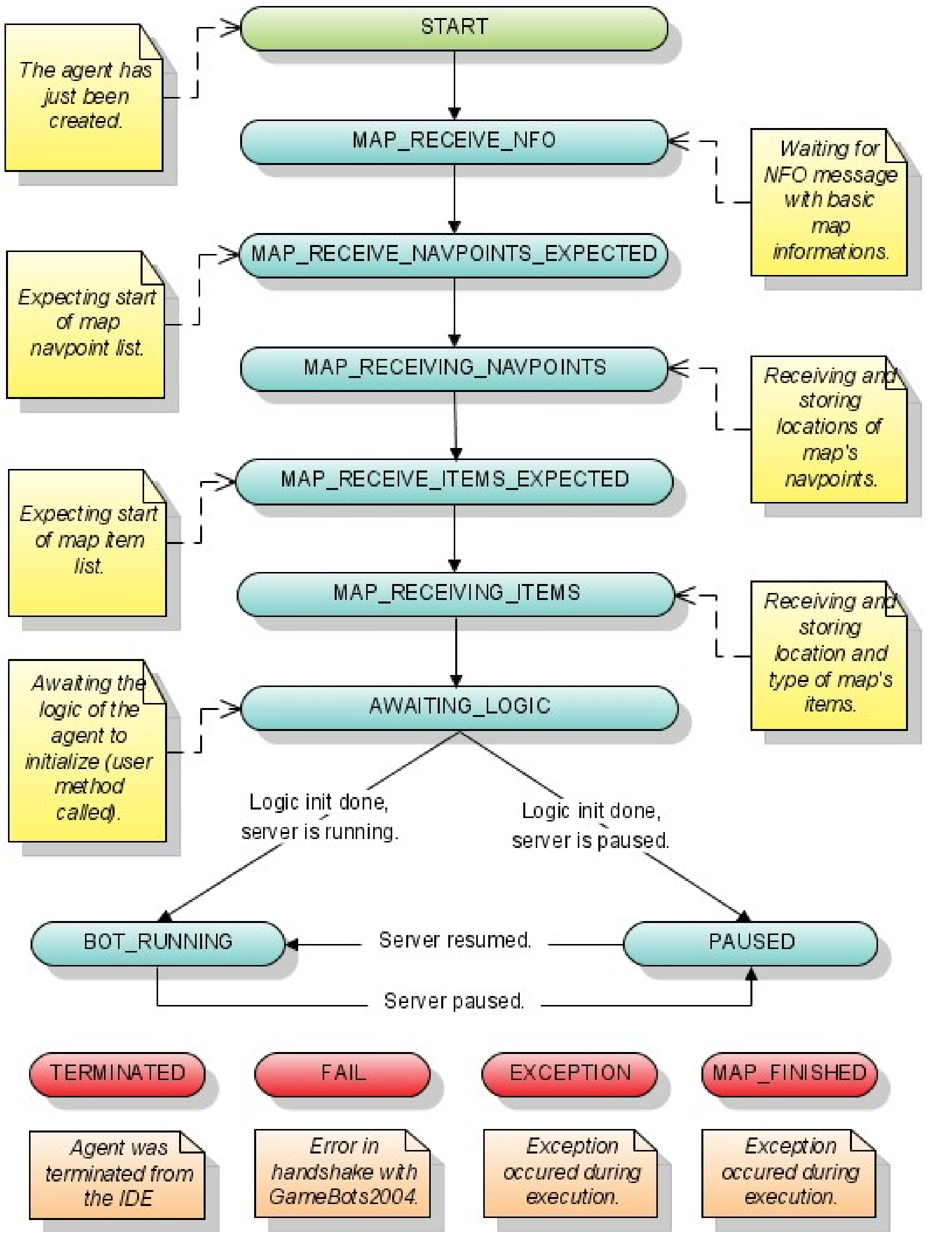
The first communication state is START - after that it will switch to the other ones in the order as described in the list above. Those states closely reflects the handshaking phase with GameBots2004 therefore they are called *handshaking types*. When the state is switched to AWAITING_LOGIC, the method from Agent object *postPrepareAgent()* is called and the Agent waits for it's end. The state is then switched to BOT_RUNNING or PAUSED depending on the state of the server.

The problems may happen anywhere during the communication. When the problem is encountered the communication switches to one of the final state. When the state is switched to any of the final state, the agent is terminated. Reasons are:

- agent is terminated from the IDE
- GameBots2004 is of older or newer version and communication with different protocol
- exception occurred

- map on the server

Figure 5.3. Communication states and it's order during handshaking with GameBots2004


START
The agent has just been created.
MAP_RECEIVE_NFO
Waiting for NFO message with basic map informations.
MAP_RECEIVE_NAVPOINTS_EXPECTED
Expecting start of map navpoint list.
MAP_RECEIVING_NAVPOINTS
Receiving and storing locations of map's navpoints.
MAP_RECEIVE_ITEMS_EXPECTED
Expecting start of map item list.
MAP_RECEIVING_ITEMS
Receiving and storing location and type of map's items.
AWAITING_LOGIC
Awaiting the logic of the agent to initialize (user method called).
Logic init done, server is running.
Logic init done, server is paused.
BOT_RUNNING
Server resumed.
PAUSED
Server paused.
TERMINATED
FAIL
EXCEPTION
MAP_FINISHED
Agent was terminated from the IDE
Error in handshake with GameBots2004.
Exception occured during execution.
Exception occured during execution.

Map representation, navigation

Map representation and corresponding methods are covered in the class *GameMap*. *GameMap* has access to navigation points sent at the beginning of the communication from Gamebots. Those points include their neighbors and therefore there is a full representation of graph which represents the map agent is playing in.

The built-in A* algorithm can be used by agent’s logic for reasoning about static objects on the map (it is not working with dynamic game entities – e. g. Players and dropped items).

There are couple frequently used methods. First is **nearestItem**. It uses A* to find out the distance from current agent location to all items of specified type and returns the closest one. Its extension nearestHealths returns specified number of health items of at least specified strength.

Another very useful method is **runAlongPath**. This method covers running along a list of navigation points. It is capable to handle lifts which are almost in every map. It simply navigates agent from one node to another using `MOVE` command. Anytime it come across navigation point with “Lift” in the unreal identifier it starts sequence for lift. First wait for lift to come down if necessary (message `MOV`), then enters the lift and waits till the top and then resumes normal running along.

RunAroundItemsInTheMap takes care about a continuous move of agent along specified items in the map. It uses previous method for running between them. Moving along some points is frequently used in behaviors like patrolling behavior or just movement along weapons in DeathMatch mode. It is responsible for all necessary steps for run along items – pick a path to item, run along it, when close to the item, switch to the next item in the list.

SafeRunToLocation can lead an agent along path to the provided location. It cares about all related background work – obtaining path to the location and running along it.

As an agent can be disturbed during running along by some higher level task it is necessary to keep variables used by **runAlongPath** up-to-date. As it could become a complicated task as agent has many paths to run along (one to a player, another to weapon and yet another one to health item), there is an auxiliary class *PathManager* which helps *GameMap* with path management. It ensures that current path is properly initialized. There are two vital methods – **checkPath** and **preparePath**. Recommended policy is to call **preparePath** until **checkPath** succeed and after that it is safe to call runAlongPath.

As someone may not intend to use those all-in-one methods, GameMap contains methods for obtaining path to a location using built-in A* or Gamebots.

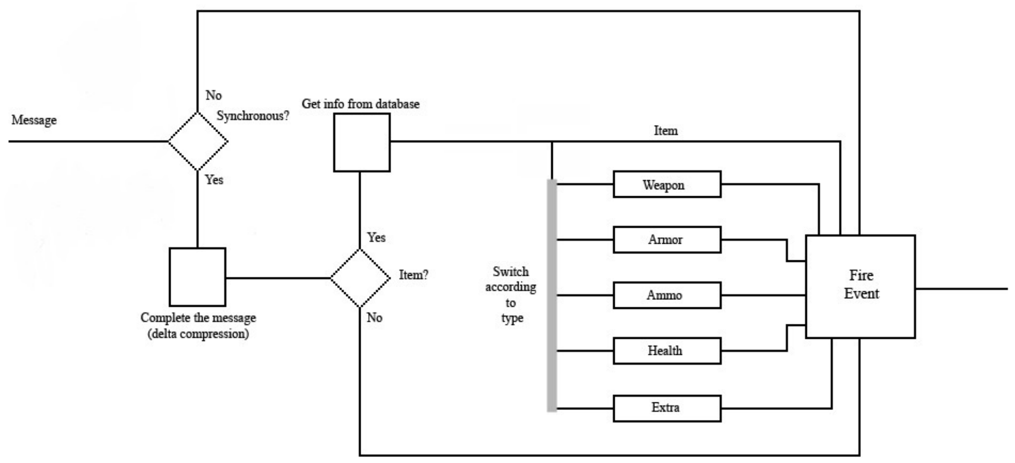
Items and Inventory

Items

Items in original Gamebots and Unreal were only of one category – inventory. This was a bit inconvenient as any developer needs common categories like weapons, armors, ammos (classes *Item*, *Weapon, Health, Ammo, Armor*). Another natural thing to know is which ammo is suitable for which weapon.

When Body receives an inventory message, it processes it and according to information stored in the database of item categories and properties, it creates an object of proper type and with appropriate values of relevant variables.

Figure 5.4. Scheme of the flow of the messages in the Body.



Example: Body received inventory with the Unreal Class which includes a string „ShockRifle“. It finds the record in the database and creates *Weapon*, which includes information about current weapon ammo, maximum capacity of ammunition, maximum effective distance for shooting and whether the weapon is suitable for ranged or melee combat.

All processing of items rely on properly filled database of items (class *ItemCathegories*). This is a bit tricky, because this database is partly filled according to personal experience from the game and effective distances of weapons could be inaccurate. Unfortunately there is no way how to find out exact values. Uncertain information are effective and maximum distances for weapons and whether weapons are better for ranged or melee combat.

Note that every incoming inventory/picked-up inventory fires two events – one for general item/pick-up, one for particular item/pick-up (e.g. *Weapon*).

User-defined items

Every item category has its own class, which is a descendant of *Item*. Problem could be with user-defined items (Unreal Tournament allows new objects in the environment). Those new inventory items are classified as *Extra* items and stored in this class. *Extra* has the same properties as *Item*. User has to handle specific attributes of his new objects by himself.

Inventory

There is no accessible inventory for agents in UT, so it is handled by Inventory module (class *AgentInventory*). This module contains a list of weapons. It registers couple listeners for inventory related events (agent picked up weapon, ammo, agent's internal status). There is one important thing to remember. Picked-up items have separate class hierarchy which is similar to the hierarchy of pick-ups (items which lay on the ground). Following classes are: *AddItem, AddWeapon, AddAmmo, AddHealth, AddArmor, AddExtra, AddSpecial*).

Inventory processes every pick-up message. If it is a new weapon, it adds it to the list of weapons. If it is an ammo, it adds appropriate amount to the ammo of proper weapon in the list. If there is not a suitable weapon, it adds the ammunition to the ammo list and every time new weapon comes, search this list for suitable ammo. It also updates current weapon ammo according to agent status information (message `SELF`).

Inventory provides information like suitability of provided ammo (when agent sees an ammo pack, he can determined, whether it is useful to pick it up) - *suitableAmmo*, whether agent has loaded weapons (ranged or melee) – **anyLoaded**. Very useful method for beginners is **getBetterWeapon**. This method do simple reasoning about weapons. It returns the most suitable weapon for the supplied positions of agent and his target. The picked up weapon is the loaded one, which effective distance is the lowest from distances greater than the distance between the agent and the target – that gives good chance of getting most devastating weapon for the situation.

Memory

Memory (class *AgentMemory*) implements four interfaces:

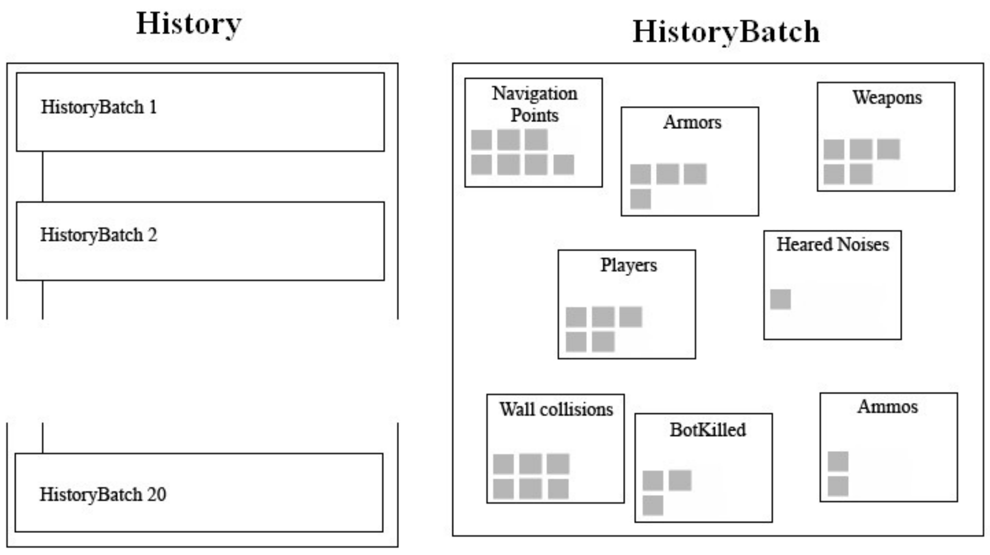
1. *WorldView* – information about what agent currently sees and about his internal status, e. g. agent's current health, weapon, ammo; visible navigation points, enemies etc. All information comes from the first batch stored in the history.

2. *RecentMemory* – a bit older information. Batches of messages (class *HistoryBatch*) are stored in the class History. Recent memory provides access to those batches and can return information like recently seen navigation points, players, ammo, etc.

3. *Knowledge* – knowledge is for persistent knowledge about the map. There are two auxiliary structures in *AgentMemory* to fulfill this task – *KnownItems, KnownPlayers*. *KnownItems* are initialized at the beginning of the game by the batch of `IINV` messages sent by Gamebots. *KnownPlayers* are updated continuously.

4. *Inventory* – an access to methods supplied by Inventory.

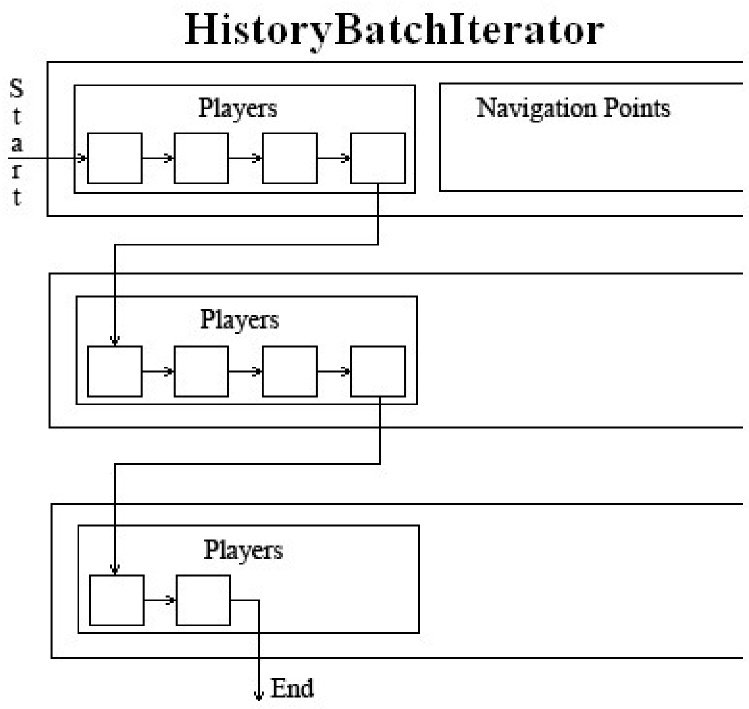
Figure 5.5. *History* is a list of *HistoryBatches*. Each batch contains a hash map in which are stored lists of messages of each type (key is a type of message, value is a hash map of messages (indexed by unique ID)).



Memory is composed of *History, Inventory* and structures for known items, players and navigation points. The most important component is *History*. This class is behind everything concerning actual and recent perception. *History* is an array of *HistoryBatches*. There are only a limited number of them stored and they are connected as linked list.

History has listeners registered for all types of messages and adds incoming messages to the first batch. When the batch ends, new is created as a shallow copy (saves a lot of space) of the previous one, so there is a continuous notion of agent surroundings. As it is only a shallow copy, when the message is different from the stored one it replaces it. This is caused by the fashion of the network communication optimization, messages are sent only when their properties changed (e. g. player's position) and hence those new messages override the old ones in the first batch. Thus batches are keeping authentic image of past seconds (it is possible to track the progress of enemy position to guess his tactics for instance).

Figure 5.6. : Example of iteration through Player messages up to 3 batches to past.



There is a *HistoryIterator* defined for iteration through messages of the same type up to the specified time.

Action selection mechanism and Client

The architecture of Client is robust. System of listeners allows for event driven action selection mechanism. Memory, inventory and knowledge allows for long-period planning and reasoning. Example of such a system is POSH which is integrated to the IDE and therefore available at once for use.

Frequency. As the action selection mechanism works in iterations, there is a property of agent called *logicFrequency*. It should be set between 5 – 20 Hz. Nevertheless it could be useful to set it to lower values – for example during debugging agent can make decisions every 1s. Higher values are not recommended as the frequency of incoming environmental information is only about 10 Hz. Then why is the top recommended value 20 Hz? There are asynchronous events and higher frequency gives agent possibility to react on them almost instantly.

Typical use of the Client

Typical user can use IDE to create a project. All projects are based on class Agent. If user don't want to use IDE inheriting class Agent would suffice to use all presented functionality.

Implementation of the agent should include overriding following methods:

- doLogic() - definition of logic.
- prePrepareAgent() - place to run all necessary initializations prior agent's connection to the simulation
- postPrepareAgent() - place to run initializations before agent is spawned to the connection. It is already connected and has all information about the map – list of all navigation points, items etc. It is a place for some map preprocessing etc.

There are several issues related with running agent that programmer should be aware of.

Restart of the agent. When agent dies it is necessary to restart History and AgentInventory. Agent therefore registers listener for BotKilled message and when it comes it calls restartAgent(). There comes a tricky part. Due synchronization issues it is necessary to reinitialize History and AgentInventory when logic is not running. Therefore it creates new instances of history and inventory in restartAgent() and when the logic is ready it calls switchMemories(). This method replaces old instances by new one and logic can keep on running. The test whether logic is ready is performed in the run() of Agent.

PostPrepareAgent. It is not allowed to call any Commands in this method. All commands are working after the INIT message is sent to GB. This message is sent right after PostPrepareAgent().

Known Issues

There are several known issues.

HealthVial. This problem usually occurs when agent is trying to pick up HealthVial (item, which increases agent's life by 5 points). It happens only if agent is going over the spawning point of the item just in the time it is being spawned. Hence agent would like to go to the spawning point and thinks that there is a health vial which he had already picked it up. This situation ends with a stuck if there is no timeout for run to item.

Built-in A*. It is recommended to use built-in A* only for estimation of distance to items, enemies etc. but not for navigation itself. Built-in A* sometimes uses transitive edges in the graph and as the two points are sometimes not reachable from each other it causes agent hit the wall.

JavaDoc

More information about any issue presented here can be found in designated JavaDoc.

Chapter 6. IDE module

Overview

Pogamut IDE is implemented as a plugin for NetbeansTM development environment. Main function of Pogamut plugin is to support implementation of bots, their debugging and validation of implemented model. These stages are supported by:

- Implementation – Pogamut supports three types of bots: Java bot, Scripted bot (implemented mainly with Python in mind) and POSH bot. Each of these bot types has its own Project type implementation conforming to NetBeans platform.
- Debugging – is supported by:
    - List of registered servers
    - List of running bot instances
    - Introspection of running bots
    - Log viewers
    - Bot remote control panel
    - Server control panel
- Validation – bot's behaviour can be validated by declarative experiments. Experiments are supported by Experiment project type.

Pogamut Plugin is build on top of the Pogamut Core module and provides GUI to it's functionality.

Class overview

Root package of NetBeans plugin is *cz.cuni.pogamut.netbeansplugin*. Most of the important classes that „do the job“ in this package and it's subpackages are implementations of some classes from NetBeans Platform API. You have to be familiar with NetBeans Platform API in order to understand the big picture. Only the important classes will be highlighted in this overview, the exhaustive list of all classes is in enclosed JavaDoc.

Package cz.cuni.pogamut.netbeansplugin

class BotNode – node representing a running bot, it is displayed under UT server node in „Runtime“ panel.
class BotNodeChildren - list of nodes under BotNode (Logs, Introspection etc.).
class NbUTServer - UTServer implementation with some features specific to NetBeans platform, it raises events in case of connecting new bots and registering experiments etc.
class UTServerNode - node representing Unreal Server (NbUTServer object) in NetBeans and providing all the associated actions.

Package cz.cuni.pogamut.netbeansplugin.exceptions

Package containing exceptions specific to the Pogamut plugin.

Package cz.cuni.pogamut.netbeansplugin.experiments

Runtime support for experiment project type.

class ExperimentNode - represents experiment at runtime. It provides common actions, shows log etc.

Package cz.cuni.pogamut.netbeansplugin.introspection

Nodes for introspection of bots properties and their periodical updating. Introspection isn't even driven, IDE has to update properties by itself.

class IntrospectableNode – node representing some introspectable object (e.g. *Agent*), it is wrapper of *IntrospectableProxy* object (found in PogamutCore). It contains nested class *Root*, which acts as root node of introspection and is responsible for periodical updating of all properties.

Package cz.cuni.pogamut.netbeansplugin.logging

Logging package is being used for viewing Bot logs (Platform log, User log, In log, Out log) and for Experiment log. This package contains both presentation classes (*LogNode*, *LogViewerPanel*, *LogViewerTopComponent*) and business classes (*LogRecordsSource*, *LogTableModel*, *OutProxy*, *InProxy*, *…*).

Standard Java logging isn't sufficient for purposes of interactive IDE. Therefore Pogamut has it's own logging API. Main class of this API is *LogRecordsSource*, it enhances functionality of standard *Logger*. Logger simply sends incoming messages to all *Handlers* and they filter the messages on the fly. *LogRecordsSource* works similarly but it caches some amount of last *LogRecords* so when the associated filter (*LogRecordsSource.Filter*) changes it can provide new filtered sequence of *LogRecords* received in the past to all *LogRecordSourceListeners*.

Business classes:

class LogRecordsSource – main class of whole package, it enhances standard Java *Logger*.

- When it receives new log record it filters it and send that record to all listeners *(LogRecordListener.notifyNewLogRecord(LogRecord r))*.
- When the filter changes it computes new filtered sequence and sends it to all listeners *(LogRecordListener.setNewData(Collection<LogRecord> r))*, they are supposed to discard previously received records and use this new sequence.

interface LogRecordListener - Listener for changes in *LogRecordSource* object. There are two types of change:

- New record arrives
- Filter of records source has changed

class LogTableModel - Table model designed to cooperate with LogRecordsSource object through *LogRecordListener* interface. When the user changes *Filter* for actually viewed log, the *LogRecordSource* fires *notifySetNewData()* event which causes update of all listeners, including this table model.

class InProxy (*OutProxy*) – these classes are listeners on all incoming (outgoing) GB messages and translate them to *LogRecords*. They both extend *LogRecordsSource* class.

class LogProxy – adapts standard Java Logger to *LogRecordsSource*.

Presentation classes:

class LogNode – represents LogRecordsSource. It provides filter through “Properties” window and opens log viewer window when user double clicks it.

class LogViewerPane – GUI component showing log records in table. The table shown in this panel uses *LogTableModel* mentioned above.

Package cz.cuni.pogamut.netbeansplugin.options

Classes for Options panel „Pogamut“ shown under Tools->Options. These classes were generated by NetBeans „Option Panel“ wizard and then manually edited. Options are stored using Java Preferences API, for details see *load()* and *store()* methods in class *NetbeanspluginPanel*.

Package cz.cuni.pogamut.netbeansplugin.project

Implements all necessary classes from NetBeans Platform's Project API needed to set up these types of Pogamut projects: Java bot, Posh bot, Scripted bot and Experiment.

All bots are being run inside the same JVM as IDE. This greatly simplifies communication between IDE and running bot. If the were to running in different JVMs then RMI or Corba had to be used.

Java bot project

Java bot project uses standard NetBeans infrastructure for Java SE projects. The only difference is in build script (build.xml). Java SE projects are being run in standalone JVM. Java bot project type has modified „run“ task. This task launches BotLauncher program (*BotLauncher* class resides in cz.cuni.pogamut.netbeansplugin.project package, in project BotLauncher, not in NetBeans plugin) which connects to *LauncherServer* running inside IDE. This way Java bots can be run inside IDE.

Posh bot, Scrip bot and Experiment projects

These are completely new project types implementing all classes required by NetBeans Platform. Tutorial on writing new project types can be found at http://platform.netbeans.org/tutorials/nbm-povray-1.htm. Our implementation is inspired by this tutorial.

class PogamutProjectFactory – ancestor of classes responsible for identifying project directory on disc (in open dialog).
class PogamutProject - ancestor of all Pogamut projects. Provides common functionality required by NetBeans.

Method run(File file) is the place where bots are loaded from source file, instantiated and connected to the server selected in the IDE.

class PogamutProjectLogicalView – tree structure showing source files and associated user actions (“run”, “delete”, etc)
class LauncherServer – server waiting for requests on launching Java bots inside the IDE.

Package cz.cuni.pogamut.netbeansplugin.project.templates

Each project type has its project template. Project templates are shown in “New project wizard” and unpack empty projects parametrised by user input (project name) to specified location.

Project templates were generated by “New Project Template” wizard and then manually edited. Customization of unpacked templates is performed in **WizardIterator.instantiate()* method.

Chapter 7. Mediator

Overview

Mediator can be viewed glue between parser and client or as a messenger delivering messages from parser to client and vice versa. It wraps threads that are waiting for the message from one side to be delivered to the other side. It is used by the Client either for the Local Parser or Remote Parser (see chapter Parser). The Mediator has also some knowledge about the GameBots2004 protocol. It recognizes the end of the communication (when MapFinished or Disconnected message arrives) and correctly terminates itself at the end.

Class Overview

All classes and interfaces of the mediator can be found in package *cz.cuni.pogamut.communication.*

class Mediator - implementation of the Mediator, messenger between Parser and the Client
interface MediatorClientInterface - interface that every Client has to implement, it allows the Mediator to receive messages from the Client for the delivery to the Parser and sending messages to the Clint that were received from the Parser
class MediatorForClient - class which implements MediatorClientInterface and is used to by RemoteParser for creating new Mediator
interface MediatorGBInterface - interface for sending messages to the GameBots2004
interface MediatorParserInterface - interface for receiving parsed messages (derived from MessageObject)

Class Mediator

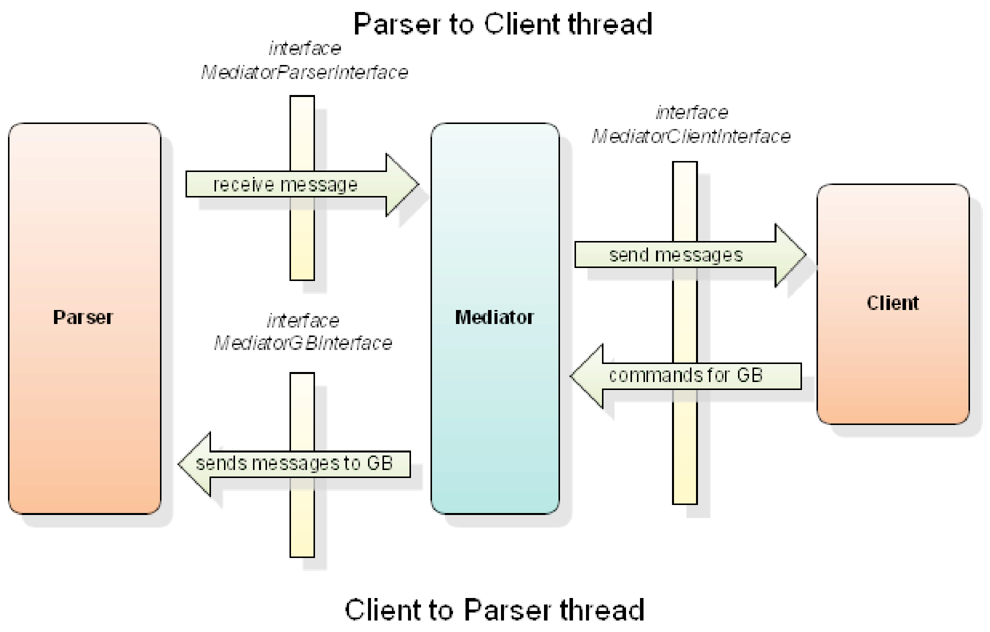
The Mediator is like messenger between two sides. On the right side is Parser, who is producing parsed messages from GameBots2004 for the Client. On the left is the Client or somebody who accepts parsed messages from GameBots2004 and produce String commands. The Mediator implements the delivery of those messages.

It is configured by three objects. They sequentially implements *MediatorParserInterface*, *MediatorClientInterface*, *MediatorGBInterface*. Through those interfaces the Mediator is receiving and sending messages. It creates two threads to achieve this. One thread for one way of the communication. Each thread is transporting the messages from one side to another.

The Mediator is aware of the protocol of GameBots2004. When the message MapFinished or Disconnected is received, it shut downs itself – terminating the delivery of messages and stopping threads.

Whenever any exception is raised, the Mediator catches it and shut downs both threads.

Figure 7.1. Typical usage of the Mediator, one thread is transporting messages from the Parser to the Client, another one from the Client to the Parser



Usage

The Mediator is used at two places.

First it is used to link the Parser class instance with AgentBody (LocalParser) class that is processing parsed messages from GameBots2004.

Second it is used by RemoteParser server to connect new client with the Parser class.

Chapter 8. Experiment

Idea

Experiments should allow user to write script that would set up the server – change map, create and spawn bots and then run the desired scenario up to a certain point defined by user where it should terminates. Experiment definition also contains a list of events and actions (e.g. triggers).

Events definition can express any first order logic sentence. Implementing this is not a trivial task therefore we are using already existing engine, namely JBoss Rules 4 (formerly known as Drools).

JBoss Rules (Drools)

Drools is an open-source rule based engine that may be distributed and used for free. It defines it's own language for writing the rules and allows user to call Java from the rules. Basically the rules are if-then rules. They have precondition and effect. Any sentence from first order logic may be written into the precondition thus it is giving the user to express a lot.

Drools are using Java Beans specification to access the properties of the inserted facts. Therefore it is quite easy to write the preconditions for the events as you are using the same names for object properties as in the code of it's class itself.

You may find more information and the documentation of Drools at web page http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossRules.

Package cz.cuni.pogamut.experiments

class Experiment- main class wrapping whole experiment

class ExperimentConfiguration - class that contains configuration variables for the Experiment object

class ExperimentGlobals - extends HashMap and serves for storing global variables created during the experiment, it is used by the user (programmer) of experiment

class ExperimentRules - wraps a few Drools classes that are needed by the Drools engine to startup such as configuration object, compiled rules and custom class loader

class ExperimentStartup - class that serves as the mark up of the beginning of the experiment inside Drools engine, it is added as the first fact to the session so it let the user to declare startup rule that should initialize experiment

class Message - simple class containing string and integer property that can be used by user to assert facts into the Drools engine

class Parameter - parameter of the experiment that can be specified via IDE (important files)

Class Experiment

Initialization of the object

For creating an Experiment object we need three things:

- Drools rules – text definition of the experiment in Drools language provided by the user, internally it's passed as an *InputStream*
- object implementing *UTWorld* interface – environment where the experiment will be run, internally it is always UTServer or it's descendant NbUTServer
- output directory where we should save results (e.g. logs), the class requires the directory to be empty

Those variables are part of ExperimentConfiguration. Experiment class contains only one constructor with parameter ExperimentConfiguration.

Drools rules are then passed to the ExperimentRules that is calling Drools engine and compiling them. During the compilation (and the whole process of Experiment instantiation) it may raise Exception (e.g. because of compilation error, file is missing, etc.)

Problems with class loader during evaluation of the rules

Drools version we're using (version 4 RC2) strangely handles the class loaders. The custom class loader may be specified during the compilation of the rules allowing the user to use other Java libraries for instance, but the same class loader is not used during the execution. That leads to the exception during the evaluation of the rules when the user is trying to instantiate an object form his or her library. We need this custom class loader for accessing the agents the user is created through our IDE.

We have been reading the source code and has found out that thread's context class loader is used as the helper when the class is not found during the evaluation. So we are setting the thread's context class loaded to our *BotProjectsClassLoader* instance that is loading the jars from projects directory of the Pogamut platform directory.

We feel that this is rather ugly solution but as the Drools doesn't (at present time) contain any mechanism that will allow us to specify the custom classloader that should be used during the evaluation the rules then we had no other chance.

runExperiment()

After the Experiment object is successfully instantiated (no exceptions occurred) it can be started with method *runThread()*.

The method *runThread()* will start the experiment in separate thread. The thread will first call the method init() that initialize the Drools stateful sessions (see Drools documentation for more) and inserts first facts and sets up the global variables.

Global variables:

- *utWorld* – interface to the server that allows the user to control the environment
- *experiment* – actual instance of the experiment that should be used only for stopping the experiment via experimentEnd() method
- *log* – Logger instance for the whole experiment
- *globals* – instance of ExperimentGlobals class that serves for storing objects that may be created during the course of experiment (we have found that Drools doesn't preserve changes to global variables throughout different rules execution

- *parameters* - map of parameters of the experiment

Facts:

- instance of *ExperimentStartup* with property *startup* set to true – this allows the user to write the rule that can fire at the beginning of the experiment
- each parameter is also asserted as a fact

Handling of the agents

New facts are added to the Drools session every time the new agent is launched by some of the experiment's rule into the *UTWorld*. Experiment is implementing *BotEnteredWorldListener*, *BotLeftWorldListener* and register itself as the listener into the *UTWorld*, therefore it always knows when new agent is started. Two facts are added or removed when new agent enters resp. leaves the environment:

- instance of Agent – the whole agent that was created and entered the environment
- instance of AgentMemory – the Memory instance of the agent that has entered the environment

We need two facts because the Drools doesn't support nested identifications of the variables. You can't write a rule that depends on the state of inner property of the property of some object (e.g. on the name of the agent that is stored in the property memory inside the agent instance).

Evaluating the rules

The rules are evaluated every time a fact is inserted to the session or it has changed inside the session. That is including:

- adding new agent facts – every time new facts about some agent are added or removed from the session, the rules are evaluated
- agent's logic thread iteration end event – the Experiment is registering itself as an listener for agent's logic thread iteration end event, every time the agent finishes it's **doLogic()** iteration, the facts are updated and rules are evaluated

Saving logs to hard drive

Experiment log and all logs from every agent that has ever entered the environment during the run of the experiment is saved to the hard drive to the directory specified during the instantiation of the object.

Termination of the experiment

Experiment may terminate due to an exception or by calling **experimentEnd()** method on Experiment instance. After the end of experiment, all logs are closed and the Drools session is destroyed.

Running Experiment from command line

The class Experiment contains also a **static void main(String args)** method that allows the user to run the experiment from the command line. If it is started without parameters, it will give this help:

```
==================
Pogamut Experiment
```

```
==================

This class is meant for running experiment without gui in batch mode.
After you debug your experiment using Pogamut GUI (NetBeans) you may
create a batch file that will run the experiments using this class
main method. Using batch file you may run several experiments over night.

Usage:

    java -cp ./src;./lib cz.cuni.pogamut.experiments.Experiment
         -f file.drl -h server:port [ADDITIONAL OPTIONS]

    Note that you have to specify java '-cp' flag, where you have to
    specify classpath for sources and libraries.

Required options:

    -f experiment_rules.drl      ... rules file of the experiment
    --file experiment_rules.drl

    -h host[:port]               ... host where UT2004 with GameBots2004
                                  is running, where to run the experiment
    --host host[:port]

Additional options:

    -n number_of_repeats         ... how many times to run the experiment
    --number_of_repeats                                         (default 1)

    -o output_directory    ... directory where to save results (default '.'),
    --output output_directory           if doesn't exist, will be created

Example:

    java cz.cuni.pogamut.experiments.Experiment -f myExperiment.drl
                       -h artemis.ms.mff.cuni.cz:3001 -o /tmp -n 5

    This will run experiment defined in myExperiment.drl (Drools
    rule file) on host artemis.ms.mff.cuni.cz (where UT2004 GameBots
    server is running). The experiment will be repeated 5 times and
    results will be saved to the /tmp directory. This example assumes
    you have your PATH set to java.
```

Chapter 9. Introspection

Classes under *Introspection* package provide infrastructure for introspecting arbitrary object implementing *Introspectable* interface. Introspectable interface has only one method returning an IntrospectableProxy object. IntrospectableProxy is main object for introspection, it should reflect properties of object being introspected (the one that returned this proxy). It is “logical view” of the introspected object.

IntrospectableProxy interface has two methods *getChildren()* and *getProperties()*. Method *getChildren()* returns array of *IntrospectableProxy* objects that should be presented in tree views as children of this proxy. Method *getProperties()* returns array of *Property* objects, these are for example shown in “Properties” panel in the IDE.

There are two approaches for creating list of properties and children for introspection:

- automated introspection - based on Java Reflection API in the case of Java bots or on traversing script context in the case of Posh bot.
- user defined introspection – programmer overrides getIntrospectableProxy() method in *Introspectable* interface and returns his own implementation of *IntrospectableProxy* that will provide different logical view (with properties that cannot be directly obtained by introspection or some simplified view).

There is default implementation of *IntrospectableProxy – DefaultIntrospectableProxy*. Its facilitates lazy initialization of properties and introspectable proxies, it is used by automated introspection.

Pogamut implements two types of automated introspection – Java introspection and Python introspection. But it is easy to provide introspection support for other types of scripting languages. You have to:

- implement *ScriptProxy* that will be aware of scripting language, it will probably work with internals of ScriptEngine for that language (e.g. Contexts, globals)
- implement *ScriptProxyFactory* that will create *Script proxy* object if it will recognize given *ScriptEngine*.
- register that *ScriptProxyFactory* through SPI

Class *ScriptProxyManager* is then responsible for getting the right *ScriptProxy* for your *ScriptEngine*. It will ask all registered *ScriptProxyFactories* and stop when first of them returns *ScriptProxy* object.

Python introspection is a good example of this mechanism.

Package cz.cuni.pogamut.netbeansplugin. project.introspection.java

Introspection of Java objects is provided by class JavaReflectionProxy. It uses Java Reflection API to get list of all children and properties. Children are all fields implementing *Introspectable* interface. Properties are all fields marked by *@PogProp* annotation that has registered property editor (*PropertyEditorManager.findEditor(field)* returns non null value).

Chapter 10. Bot samples

Simple bot

Simple bot is like a slight introduction to the platform. Its logic is quite primitive and it just demonstrates basic use of client's libraries.

```
protected void doLogic() {
  // IF-THEN RULES:
  // 1) are you walking?     -> (check WAL)
  if (this.memory.isColliding()){ this.stateWalking(); return; }
  // 2) do you see item?     -> (pick the most suitable item and run for)
  if (choosenItem != null || this.seeAnyReachableItemAndWantIt())
        { this.stateSeeItem(); return;
  // 3) do you see navpoint? -> (pick navpoint randomly and walk towards)
  if (this.memory.getSeeAnyReachableNavPoint())
        { this.stateSeeNavpoint(); return; }
  // 4) true                 -> (not seeing any navpoint, turn a bit)
  this.stateTurnAround();
}
```

Simple bot's `doLogic` procedure shows main outline of the simple bot's intentions. Its logic is based on if-then rules. There are only four of them.

1. Collision - fires if bot collides with something - this method checks for wall and player collisions. If some of that is true, bot tries to jump. It usually helps as collisions are frequently caused by small obstacles in the way.
2. See item - fires if bot spots an item, it tries to pick it up. And now something a bit interesting, how bot runs to the item? The simplest way would be to run directly to the item. But such approach would not work every time. Why? Imagine, that bot spots an item lying somewhere on raised platform. He attempts to run to it and hit the wall. So the better way is to use GameMap method `safeRunToLocation()`. Which will guide bot safely to the chosen location along the path obtained from UT server. This path is computed using server's A* algorithm.
3. See navigation point - as we said in the introduction to the Simple bot, it is a simple bot. Therefore it aimlessly wander around the location. For such a purpose serve last two if-then rules. First is fired when bot spots a navigation point and run to it.
4. Second rule is turn around. It fires only when bot is not colliding, is not seeing any item or navigation point. Then it turns around in the hope of spotting something.

Prey

Prey is only a toy for the hunter and as a toy should at least move along and withstand some playing around. So pray is simply running around the medical kits placed in the location.

```
protected void doLogic() {
  // 1) is colliding?     -> go to WALKING       (check WAL)
  if (this.memory.isColliding()){ this.stateWalking(); return; }
  // 2) go around health items
  this.stateGoAroundItems();
}
```

Prey's logic is even simpler than the Simple bots one. Though there is one thing worth describing. It is the second rule named `stateGoAroundItems()`. This procedure is responsible for the moving-around-medkits behavior. How can we get such a behavior easily? GameMap contains method `runAroundItemsInTheMap()`. This method accepts a list of items as a parameter. So we need to obtain those items. As they are all the time the same, it will not be effective to obtain them every call of `doLogic()`. There is an easy solution to this little problem. As you may recall, there are other methods of agent then `doLogic()` which could be overridden. In our case, we need to override `postPrepareAgent()`. This method is called right after bot receives map and game information. Therefore he knows, how the map looks like and knows where are all medkits. So we can easily obtain this list once for all the live of the bot.

Next piece of code shows how to obtain all health objects and insert them to the list of Items which is necessary for the `runAroundItemsInTheMap()` method (note that `getKnownHealths()` returns `Health` objects, so it could not be simply put to the array of `Item`).

```
this.healths = new ArrayList<Item>();
for ( Item item : this.memory.getKnownHealths())
  this.healths.add(item);
/** shuffle the items so no bot will go into the same river twice */
Collections.shuffle(healths);
```

As bot now possesses list of items, it can call `runAroundItemsInTheMap()` all the time he is not colliding with something. That would be all about the Prey.

Hunter

Hunter is the most advanced example of bot. He is capable to choose the best weapon according to the current combat situation. He engages enemy when he spots him. When hurt he searches for the closest medkit and when he spots some item he makes some reasoning before he picks it up so he is not picking up useless crap. For such a complex behavior we will first present his list of if-then rules and then will explain one rule by another highlighting usage of some special methods of the platform.

Hunter is a good example of introspection as well. He allows user to disable some parts of his if-then rules. As we can see bellow, some rules has as their first precondition `this.shouldXXX`. Those boolean values are enabled via introspection in IDE and user can then for example disable bot from engaging enemy by a simple click.

```
protected void doLogic() {
  // 1) see enemy and has better weapon? -> switch to better weapon
  if (this.shouldRearm && this.memory.getSeeAnyEnemy()
   && this.hasBetterWeapon())
       { this.stateChangeToBetterWeapon(); return; }

  // 2) do you see enemy?     ->   start shooting / hunt the enemy
  if (this.shouldEngage && this.memory.getSeeAnyEnemy()
   && this.memory.hasAnyLoadedWeapon()) {this.stateEngage();return;}
  this.enemy = null;

  // 3) are you shooting?   ->  stop shooting, you've lost your target
  if (this.memory.isShooting()) { this.stateStopShooting(); return;

  // 4) are you being shot?  ->  turn around, try to find your enemy
  if (this.memory.isBeingDamaged()) { this.stateHit(); return; }
  // 5) do you have enemy to pursue? -> go to the last enemy position
  if ((this.lastEnemy != null) && (this.shouldPursue)
   && (this.memory.hasAnyLoadedWeapon()))
         { this.stateGoAtLastEnemyPosition(); return; }

  // 6) are you walking?              -> check WAL
  if (this.memory.isColliding()) { this.stateWalking(); return; }

  // 7) do you see item?  -> pick the most suitable item and run for it
  if (this.shouldCollectItems && this.seeAnyReachableItemAndWantIt())
        { this.stateSeeItem(); return; }

  // 8) are you hurt?            ->  get yourself some medKit
  if (this.memory.getAgentHealth() < this.healthLevel
   && this.canRunAlongMedKit()) { this.stateMedKit(); return; }

  // 9) run around items
  this.stateRunAroundItems(); return;
}
```

Now we will describe all 9 rules.

Has better weapon

This rule is the first, because bot needs to have the best weapon before he starts engaging the enemy. Hunter uses one of those magic methods of `Inventory` which can be found in memory. This method evaluates all weapons which are currently available to the bot and returns the most proper one. It makes the reasoning according to the distance between the bot and his opponent.

How is this reasoning performed? Every type of weapon has some additional, hard-wired properties. We will need maximal and effective distance for this time. The values of those are just guessed, so they are not exact. Nevertheless for such reasoning they suffice. The chosen weapon is then the one loaded, with the lowest effective distance - lowest means that it is the most deadly ones (like the flak cannon for instance) and with maximal distance greater than the distance between the player and his enemy. As we can see, such reasoning is not always right, but in most cases it is sufficient as bot usually does not possess all types of weapon.

Engage

Engage is fired when hunter possesses loaded weapon and spots an enemy. He starts to fight with him. Procedure `stateEngage` first updates information in `enemy` and then starts hunter to shoot at enemy if he is not shooting already. If he is far from enemy, he runs straight towards the enemy.

Stop shooting

Stop shooting is crucial rule even though it is quite simple one. It fires if bot is shooting. As it is after the engage, bot no longer sees any enemy and therefore should stop shooting. The problem with shooting is that UT does not control such things and therefore bot has to care about that.

Hit

Hit is fired when bot is being damaged. As it is after the engage rule it means that someone shoot at the bot and he is not aware of his presence. So he turns around hoping that he will spot enemy in the time and starts to engage him.

Pursue

This state serves for pursuing the enemy. This behavior uses variable `enemy`. If `enemy` is `null`, he does not perform any pursuing. If there is some opponent stored, bot runs to its last position and if he does not meets him, he set `enemy` to `null` and therefore ends the pursue. Running to enemy's last position is performed using `safeRunToLocation()`.

Walking

`StateWalking` is the same procedure responsible for collision with walls and players as in previously described bots.

Grab item

This rule is fired when hunter spots some item and want it. Procedure `seeAnyReachableItemAndWantIt()` contains a bit of reasoning about what is useful for bot at the moment. The reasoning differs according to the category of the item. If it is weapon, bot want the item only if it is for short distance fight and bot already has long distance fight weapon and vice versa. Medkits are wanted only if bot has lower than maximal health (normally 100). Armor is chosen if bot has not reached maximum possible armor and ammo is chosen if bot possesses weapon for it.

If, after all that reasoning, is the weapon worth for bot, it runs to it using `safeRunToLocation()`.

Medkit

Medkit rule fires when bot has low health level. He runs around closest medkits to heal himself. There hunter uses another useful method of `GameMap` - `nearestHealth()` which returns specified number of `Health` objects of at least specified strength. Then he can run along those objects using `runAroundItemsInTheMap()`.

Run around weapons and armors

Last rule is fired every evaluation of logic, when bot has nothing better to do. Therefore it stands for default behavior of bot when there is nothing more important. As the name could tell, it makes bot running around spawning positions of weapons and armors in the location. List of those weapons and armors is obtained in the `postPrepareAgent()`. For running along we use `runAroundItemsInTheMap()`.

Conclusion

For the more detailed description of Hunter please see the Manual of Pogamut, where is complete description of every single part of the hunter.

SPOSH bot

SPOSH bot is an example of bot whose logic is driven by SPOSH plan. POSH is a Parallel-rooted Ordered Hierarchical Slip-stack planner whose description can be found on [WWW]. The design methodology used when working with POSH is BOD - Behavior Oriented Design. This design paradigm states, that the agent has acts and senses. Acts represent his actions in the environment. Senses are connected with his perception of the environment and of his internal state. As the description of POSH and BOD is beyond the scope of this documentation we will explain the functionality of this bot presuming that the reader is familiar with the principles of BOD and POSH. Yet we will try to write it as simply as possible so it hopefully will be possible to get the main ideas even without this knowledge.

Main difference on the first sight between the SPOSH bot and previous bots is in the decomposition of the bot. Previous bots have if-then rules sequence in the `doLogic()` and methods which are used by those rules were in the same file. There it uses by default following structure. Bot is decomposed into three main categories of files. First is the bot himself - file where we can override methods of agent like `postPrepareAgent()`. Second is the POSH plan. This file contains description of the plan - similar to if-then rules in `doLogic()` in previous bots. Third category contains usually more files describing behaviors. Each behavior contains some acts and senses which are necessary for some behavior - an example of behavior could be movement, combat, communication with others, etc. This concept allows for dividing complex bot's behavior to logical parts. We will discuss this matter lately, first we will introduce the POSH plan.

```
((documentation "" "Ondrej Burkert" "Simplified Hunter")
  (DC PoshBot (goal ((fail)))
  (drives
   ((rearm          (trigger ((seeEnemy)
                              (hasBetterWeapon)))rearm ))
   ((engage-enemy   (trigger ((seeEnemy)
                              (armed)))         engageEnemy ))
   ((stucked        (trigger ((stucked)))       jump ))
   ((shooting       (trigger ((isShooting)))    stopShooting ))
   ((low-health     (trigger ((health 80 < )
                              (knowMedkits)))    runAroundCloseMeds ))
   ((see-item       (trigger ((seeItemAndWantIt)))   runToItem ))
   ((run-around     (trigger ((succeed)))       runAroundItems ))
)))
```

As we can see, the POSH plan contains rules as well, those rules are called drives. This example of POSH bot is in the matter of fact more or less just a reimplementation of Hunter's behavior. POSH capabilities are far beyond of this plan but they are not necessary for such a straight forwardly design bot .

So the bot is again rearming if he spots an enemy and has better weapon for the situation. He engage him just after the rearm. Jumps when stuck - wall / player collision, stops shooting when shooting and not seeing enemy etc.

A bit interesting drive is low-health. We will use it as an example of POSH syntax.

```
((documentation "" "Ondrej Burkert" "Simplified Hunter")
  (DC PoshBot (goal ((fail)))
  (drives
   ((low-health     (trigger ((health 80 < )
                              (knowMedkits)))   runAroundCloseMeds ))
)))
```

This would be a simple plan with only one drive. POSH syntax is inspired by the Lisp (logical language which is used a lot in the USA), so there are many brackets, be careful about having them all right. The DC means the Drive Collection and its goal is to fail. That means that bot will never reach the goal and will run forever. If you set the goal on something more specific or rather possible, bot will finish when reaching the goal.

Drives are stored in the list of drives [`(drives (()) (()) (()) )`]. They are ordered and evaluated according to that order. Each drive is a triple name, trigger and action. Trigger can contain more conditions which are linked by logical AND. There we can see two preconditions which has to be met before the drive fires. Conditions are using senses from behaviors. For instance health is a sense which returns actual level of health of the bot (number between 1 - 200). Condition `(health 80 < )` is met when bot has the health lower then 80 points - POSH syntax uses postfix notation. As an action of the drive we use there directly actions specified in the behavior.

Behaviors are then just lists of acts and senses.

```
    public boolean sense_fail() {
        return false;
    }
```

This is an example of very simple sense fail which is used as a goal in the root of the plan. As we can see, method name is begins with `sense_` which means that it falls among senses.

```
public void action_runAroundCloseMeds() {
  this.log.info("Action RUN_AROUND_CLOSE_MEDS.");
  this.bot.getMap().runAroundItemsInTheMap(this.medkitsToRunAround,false);
}
```

Next code example introduce the action `runAroundCloseMeds`. Again we have special prefix for the actions - `action_`. For the simple SPOSH bot we used only one file with behavior. The example can be currently found in the package `cz.cuni.sposh.java.examples`.

Khepera-like bot

Khepera like bot demonstrate tracing and autotracing capabilities of the agent in Pogamut. He is inspired by the well-known project Khepera [WWW] which is from the robotics field. Our bot has three "infrared" sensors - autotraces. Those sensors are in fact vectors of specified length, which starts from the center of the bot's body and goes to 3 directions - one straight forward and other two 45° to the left and the right sides from the center vector. Sensors returns true if they intersect with something solid in the location.

As we have three sensors available, there are 8 possible combinations of their states - imagine a binary code. So in the `doLogic()` of Khepera-like bot is a big if-then tree which covers these 8 possibilities and defines proper action for each one. So for instance if bot hits with his front and left sensors it turn horizontal to the right a bit and sleeps for a while to allow its avatar to finish the turn. Nevertheless there is may be a good place to note again that bot will sleep for a while after every iteration of his logic - that is according to the `logicFrequency`. We can see corresponding part of the if-then tree in the following torso of the code.

```
            if (sensorFront) {
                if (sensorLeft) {
                    if (sensorRight) {
                        // LEFT, RIGHT, FRONT
                        body.turnHorizontal(bigTurn);
                        Thread.sleep(turnSleep);
                    } else {
                        // LEFT, FRONT
                        body.turnHorizontal(smallTurn);
                        Thread.sleep(turnSleep);
                    }
```

So the Khepera-like bot is again very simple, but you can find there examples of proper use of autotraces and traces which can be very important for example for steering behavior of the bot.