TI2806 Contextproject
Emergent Architecture Design

Virtual Humans - Group 4 - Port

https://github.com/tygron-virtual-humans

Sven van Hal - 4310055

Tom Harting - 4288319

Nordin van Nes - 4270029

Sven Popping - 4289455

Wouter Posdijk - 4317149

June 19, 2015

Contents

1 Introduction

During the last sprints/weeks, our project has been divided into three separate sub-projects. These are the GAMYGDALA port, GOAL plugin and the GOALGAMYGDALA interface. This is why this report has three sections, containing an explanation for all the sub-projects.

During the last weeks we have improved, expended and updated this report so that it became more complete. In this document there will be a overview of the sub-projects with their systems. The systems will be explained by using UML diagrams, Design patterns and explaining the uses of the systems.

2 GOAL

Our goal is to implement a plugin in the existing source code. To do this, we need to have an overview of both the relevant parts of the GOAL architecture and the architecture of our own plugin.

2.1 GOAL Architecture overview

The GOAL source consists of two major parts: the grammar and the runtime. These parts can be characterized as the parser and interpreter respectively.

2.1.1 Grammar overview

In this section, the parts of the grammar that are relevant to the implementation of our custom action will be covered.

The grammar uses the antlr4 library to create clear grammar in .g4 files. It then uses this grammar to parse the program. We need to make sure that our custom action (also referred to as plugin) is correctly inserted in the existing grammar.

The parsing consists of multiple levels. The first level is converting the plain text to collections of very basic expressions. The second level turns these basic expressions into clearer expressions. In this level, every action has its own class, for example. We need to make sure that such an action exists for our custom action, and that it is created when necessary.

The classes that handle this second level of parsing are called validators, because they not only parse the program but also make sure that everything is correct. The grammar has four different validators for the four parts of the goal language: agents, multi-agent systems, modules and tests. Most relevant for us are the agents; we need to make sure that our custom action passes the validation of the agent's execution code, and also that the parameters that are passed to the action are preserved in the process.

The grammar is not yet responsible for any communication with the actual implementation of the plugin; it just makes sure that an action is created, and the parameters, though also not yet interpreted, are passed to it.

2.1.2 Runtime overview

In this section, the parts of the runtime that are relevant to the implementation of our custom action will be covered. In the runtime changes will be made to allow for interpretation of the custom action, and the actual interpretation will be hooked to the implementation of the plugin.

The runtime traverses the rules of the agent program, and when it crosses an action, it creates an action executor for said action. This action executor is then executed. We need to make sure that such and action executor exists for our custom

action.

Then, in this action executor, we need to implement the hook with the plugin implementation. We adjust the parameters so that they are in the correct format for the plugin, and then pass it to the plugin, which then handles the parameters and generates output This output must then be passed to the belief base.

2.2 Calculator Plugin overview

This section explains how the implementation of the calculator plugin is set up.

The program consists of many operations (e.g. plus, minus) that all derive from the Operation class. They have a string that represents their operator, and a function calc that gets a list of doubles as an input and produces a single double as the output. Each operation is in charge of setting its own precondition for the amount or the format of the parameters that are passed.

There is one main class, Calculator, that is responsible for finding the right operation and calling it, when being passed an operator string and the aforementioned list of doubles. It uses a class loader to find the correct operation, so that new operations can be added to the program simply by creating a new class that extends Operation in the right package.

3 GAMYGDALA

In the Gamygdala world, Agents have goals they want to achieve. Based on their beliefs, they assess whether or not they have achieved their goals, and for example what's the likelihood to achieve their goals. Based on these properties and based on the relation with other agents, the agent's emotion is calculated.

Each Agent has a separate Goal base which they update every evaluation cycle. Whenever an event happens, the impact on a particular goal has to be provided. New agent-specific values like the goal likelihood, change in likelihood, the overall utility of the goal (how useful is it to achieve the goal) are then recalculated. Finally, a new emotion is distilled from all these particular properties.

Because each goal is unique and interchangeable, the engine itself also keeps track of all goals.

3.1 GAMYGDALA Architecture overview

In the following sections, there will be an overview of how GAMYGDALA is build up from the inside.

The GAMYGDALA architecture starts with the Engine class. The Engine class contains the GAMYGDALA itself. The Engine class is responsible for maintaining the GAMYGDALA engine. It manages all the engine aspects, like the decay function and the decay itself.

The Gamygdala class manages the overall aspects of agents, goals and beliefs. When an event happens, it creates a belief with certain goals that are affected by that event. The Gamygdala class manages the appraisal of the belief over the agents.

The Agent class is by far the biggest class in the GAMYGDALA port. This is the class that handles all of the agents activities. The Agent class is the biggest class, because GAMYGDALA is build around agents, their goals, their relations and their emotion. The agents are the center of the GAMYGDALA emotional engine. The personal emotions of an agent are managed in its internal state object. The social emotions are managed in its relations object. Both object are lists of respectively emotions and relations. Relations also contains emotions.

The Belief class is the last very important class of GAMYGDALA. A belief is made when an event happens in-game. A belief then contains the goals it affects and the congruence of the belief on that certain goal. A belief also contains a likelihood, which tells how unique the event is. A more unique event will cause more effect on an emotion than a less unique event.

3.2 Porting

In the process to porting the Gamygdala engine from Javascript to Java, we have implemented a number of design patterns to better structure the code. Since Gamygdala was originally developed as a gaming engine, we had to separate the core logic from other facilitator functions. The core logic is also separated into classes with a single-responsibility and coupling is avoided wherever possible.

During an analysis of the code, we found out that he five main classes of the engine are:

- **Agent** The agent which interacts with the environment.
- **Goal** Goals which Agents want to achieve.
- **Belief** Beliefs which alter the Agent's view on the likelihood of achieving the goal.
- **Emotion** The Emotion which an Agent has after processing Beliefs (based on its Goals).
- **Relation** The Relation between two Agents.

To support our claim that we improved the code on the software engineering part, we made two UML-diagrams. The first diagram is made using our first version of the port, this was an almost exact copy of the original GAMYGDALA code. The second diagram is made using our re-factored code, it was not perfect but it was a big improvement over the original code. The UML-diagrams can be found on the next two pages.

3.3 GAMYGDALA Emergent Architecture

This project aims to create a good GAMYGDALA port in Java. With that in mind you could say that the architectural design was not emergent, but that was not true. Porting from JavaScript to Java brings a lot of needed creativity with it. For example JavaScript works mostly with String and Java with objects, this transition was a real brainteaser. To implement better software engineering methods, the architecture of the port needed to emerge, but the function needed to stay the same. We ensured this with the following actions: unit testing, benchmark testing and re-factoring.

3.3.1 Testing

Testing is one of the most important software engineering aspects. Especially for a port is testing very important, because the port needs to work exactly the same as the original program. To make sure the port works exactly the same, we made use of two different types of testing. The first is Unit testing, and the second is a Benchmark.

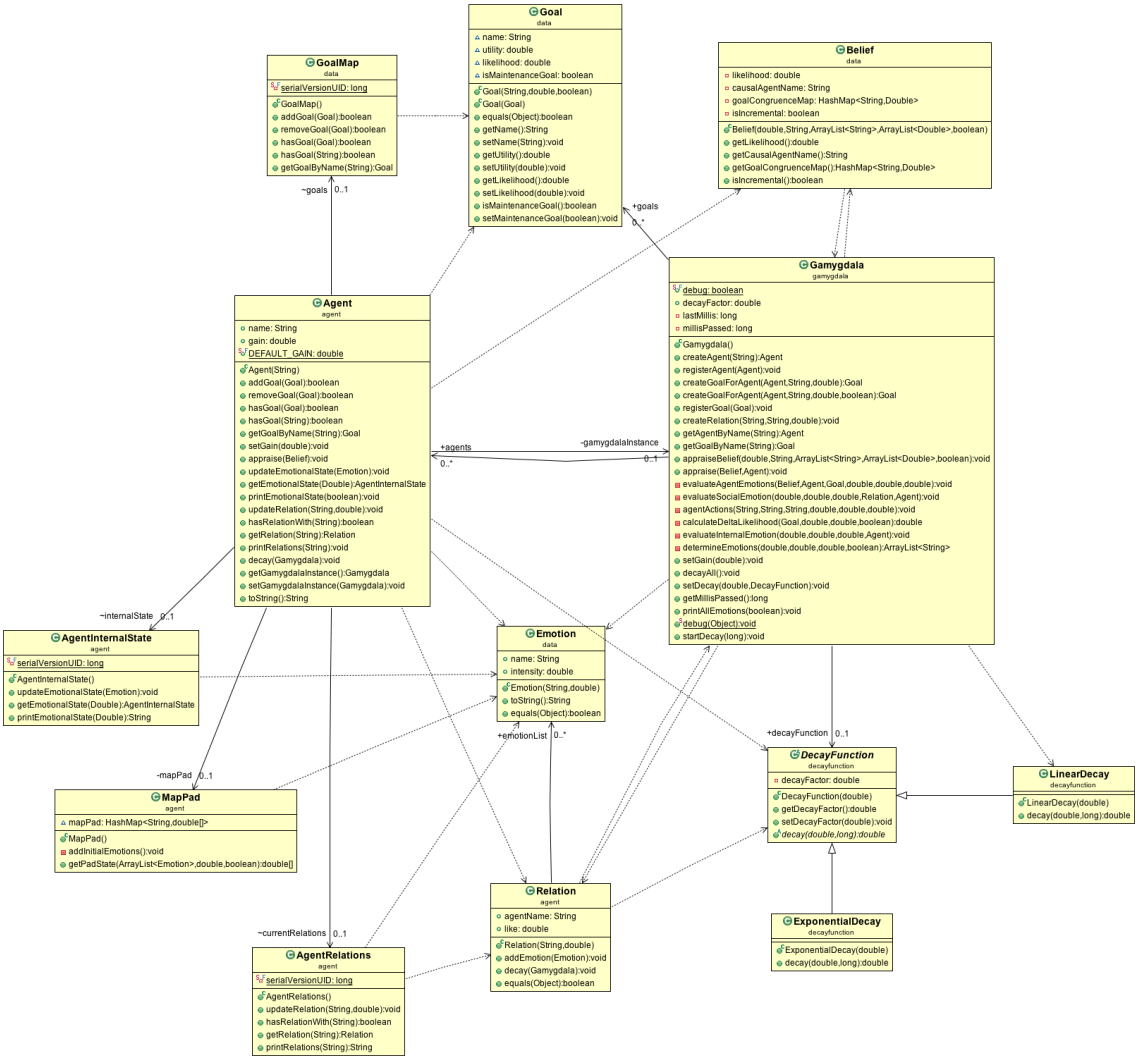

Figure 1: The UML-diagram for the initial port.

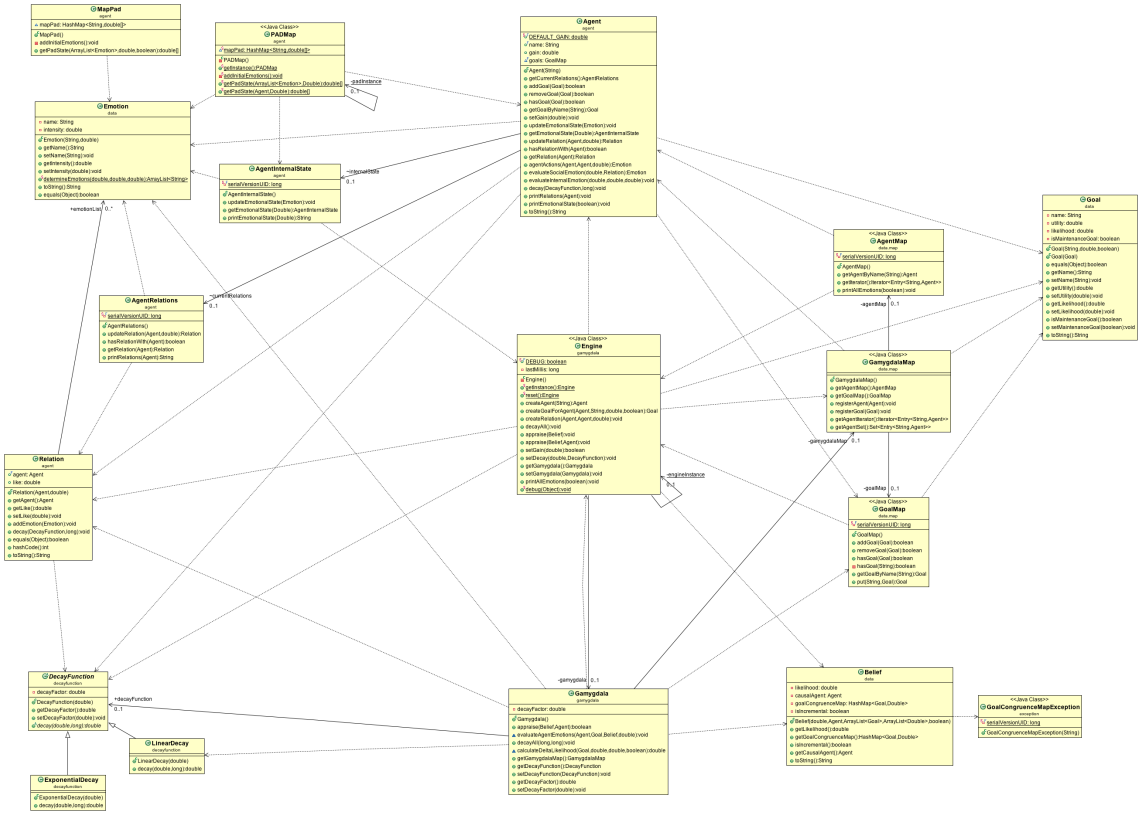

Figure 2: The UML-diagram for the re-factored port.

Unit testing A Unit test determines whether a method still functions the same as intented to do. After the initial port, useful unit tests were made to ensure the purpose of the functions. After every re-factored piece of method, running the unit tests made sure that the method still functioned the same. By this way of testing there were some bugs found in the new code after we did the re-factor.

Benchmark testing Benchmark testing was very important for this system. A benchmark tests whether your whole programs functions like it was intended to do. If the right Emotions where calculated when certain Goals and Beliefs where added to the Engine. And whether certain relations where made when a Goal belongs to another Agent, but it affected the other. After a successful benchmark test, the system is functionally correct.

3.3.2 Re-factoring

After the initial port and re-factoring of GAMYGDALA, we came to the conclusion that the Java code was still not very pretty. It had long functions, big classes and still classes with multiple responsibilities. To counter these faults, we started a big re-factor of the code. First we divided some functions in smaller new functions with less responsibility. We distributed the smaller functions over different classes. These changes made the bigger classes smaller and gave them less responsibilities.

3.4 Design patterns

Within the GAMYGDALA port there are several design patterns used. These patterns help to improve the code and make it easier to use and develop. In the following subsections, we will present to you the design patterns used within the code.

3.4.1 Singleton pattern

The singleton pattern is very useful in cases where you want to have at most 1 object of a class. The Engine Class has the singleton pattern build in. This is because it is not useful to have two GAMYGDALA engines running beside each other. Emotions can be calculated for each agent, and relation too. So there is no need for a second engine. It could only confuse programmers if they had accidentally created two different GAMYGDALA engines and has been working with both of them.

3.4.2 Strategy pattern

The strategy pattern is very usefull when you have something that needs to be calculated and it depends on certain conditions how it needs to be calculated. Because of this we used the strategy pattern to determine the kind of emotions. The emotion calculation is based on the likelihood. For certain values of the likelihood there is a different strategy to calculate the kind of emotion.