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

We have implemented GAMYGDALA as a plugin in the existing GOAL source code. To do this, we first need to have a good overview of the GOAL architecture.

2.1 GOAL architecture overview

The GOAL source consists of two major parts: the grammar and the runtime. These parts can be characterised as the parser and interpreter respectively. In both these parts we have had to hook in our plugin. We will discuss the relevant parts of both the grammar architecture and the runtime architecture, and how we embedded our custom actions in these architectures.

2.1.1 Grammar overview

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 needed to make sure that such an action exists for each of our custom actions, and that it is created when necessary.

All of our custom actions share the same format, that is to say they all take a list of values as their parameters instead of some predicate. Since these actions all share this format, it makes sense to generalise them. This is exactly what we have done. There is an abstract 'parameter action', and to create a new action in the grammar we only need to create a new class for it in a certain package.

The classes that handle the 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.

To make sure that this agent validator class did not become much larger than it already was, we created a parameter action factory that handles the creation of the parameter actions. Using a class loader, this factory creates the specific parameter action that matches the operator. This is very easy, because every parameter action executor has an 'Action Token' annotation that tells the class loader for which operator it is used. To allow for easy class loading, the reflections package was used.

The grammar is not responsible for any communication with the actual implementation of the plugin; it just makes sure that an action is created, and the parameters, though 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 were made to allow for interpretation of the custom actions, and the actual interpretation is hooked to the interface between GOAL and GAMYGDALA.

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 needed to make sure that such an action executor exists for each of our custom actions.

Since all actions still share the same format, generalisation could also be applied in the runtime. We created an abstract 'parameter action executor', which all specific action executor implementations extended. Each action executor specifies for which action it is an executor.

All these action executors are put together in the same package. That way, the runtime can easily find the action executor that matches that action is has come across. Again, a factory that uses the reflections package was used, although this time annotations were not necessary. In this case, we could simply use type parameters, since we were now trying to match an action type instead of an operator integer. This way, the code also became cleaner, because by using type parameters a parameter action executor could never be used for the wrong type of parameter action.

Once the action executor is found and actually executes, the call is forwarded towards the GOAL-GAMYGDALA. Since this interface is built so that GOAL does not have to know anything at all about GAMYGDALA except for the actions it provides, the raw GOAL terms are passed to the interface.

In some cases, the emotions of the agent need to be updated after execution. For this purpose, we have created an emotion manager class. This class is responsible for updating the emotions in the mental state of the agent.

2.2 Design Patterns

2.2.1 Factory Method Pattern

In both the grammar and the runtime, we needed a way to create actions and action executors, based on operators and actions respectively. To make sure that this responsibility was separated from the rest of the GOAL source, we decided to create a factory in both cases. To make sure we also adhered to the open/closed principle, we decided to expand this to the Factory Method Pattern. By using this specific pattern, a new factory can always be created and substituted for the old factory in case anyone wants to implement a new way to create these actions and action executors.

2.2.2 Strategy pattern

In the runtime, there was already a strategy pattern in place for the actions, but as you might have noticed, we decided to take one step further and also implement the strategy pattern for the so-called parameter action executors. They are, in essence, all

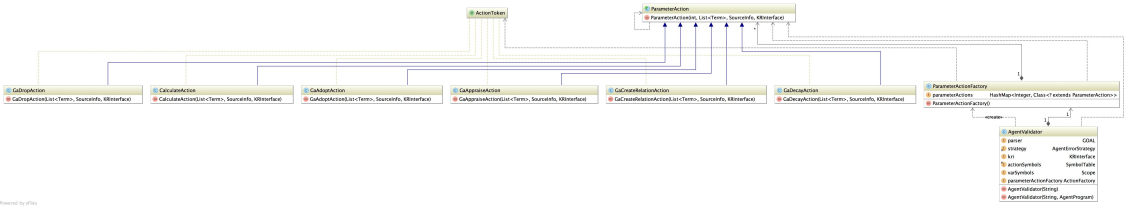

Figure 1: The UML-diagram for the grammar.

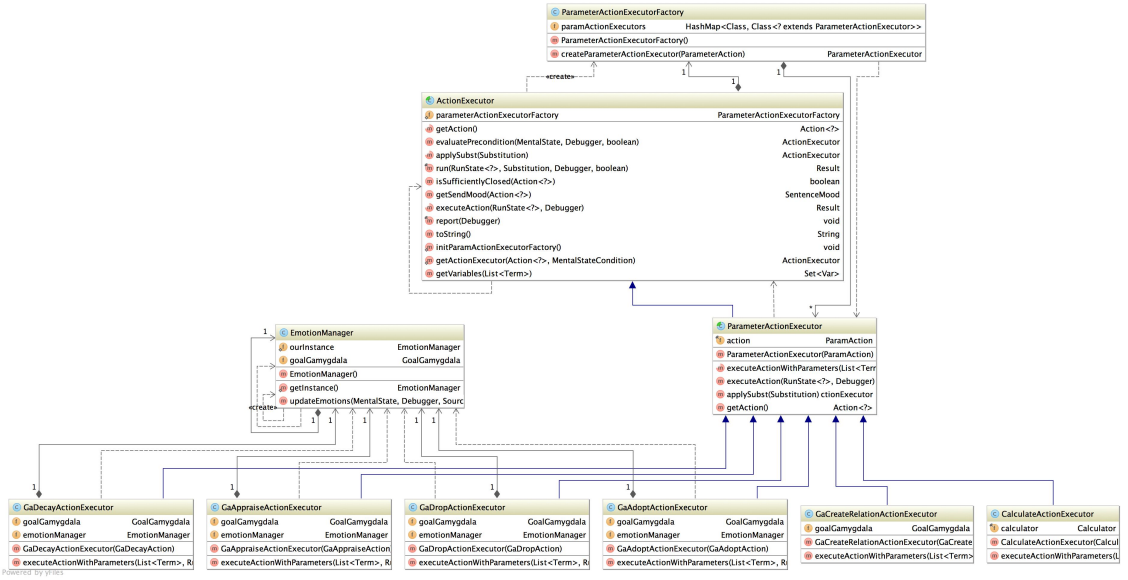

Figure 2: The UML-diagram for the runtime.

interchangeable, since they all 'execute something'. It is up to the factory to determine which of these executors is appropriate for the current action.

2.2.3 Singleton pattern

For the emotion manager, we have used the Singleton pattern. This allowed us to encapsulate the emotion updating functionality in a new class, while not having to instantiate this class countless times, wasting a lot of memory.

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.

In this section, first you will get insight into the current overall GAMYGDALA Architecture. After that the process of the initial port will be described. How the architecture of the GAMYGDALA emergent will be in the third subsection. And last, but not least, the design patterns within the GAMYGDALA port will be described.

3.1 GAMYGDALA Architecture overview

In the following section, 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.

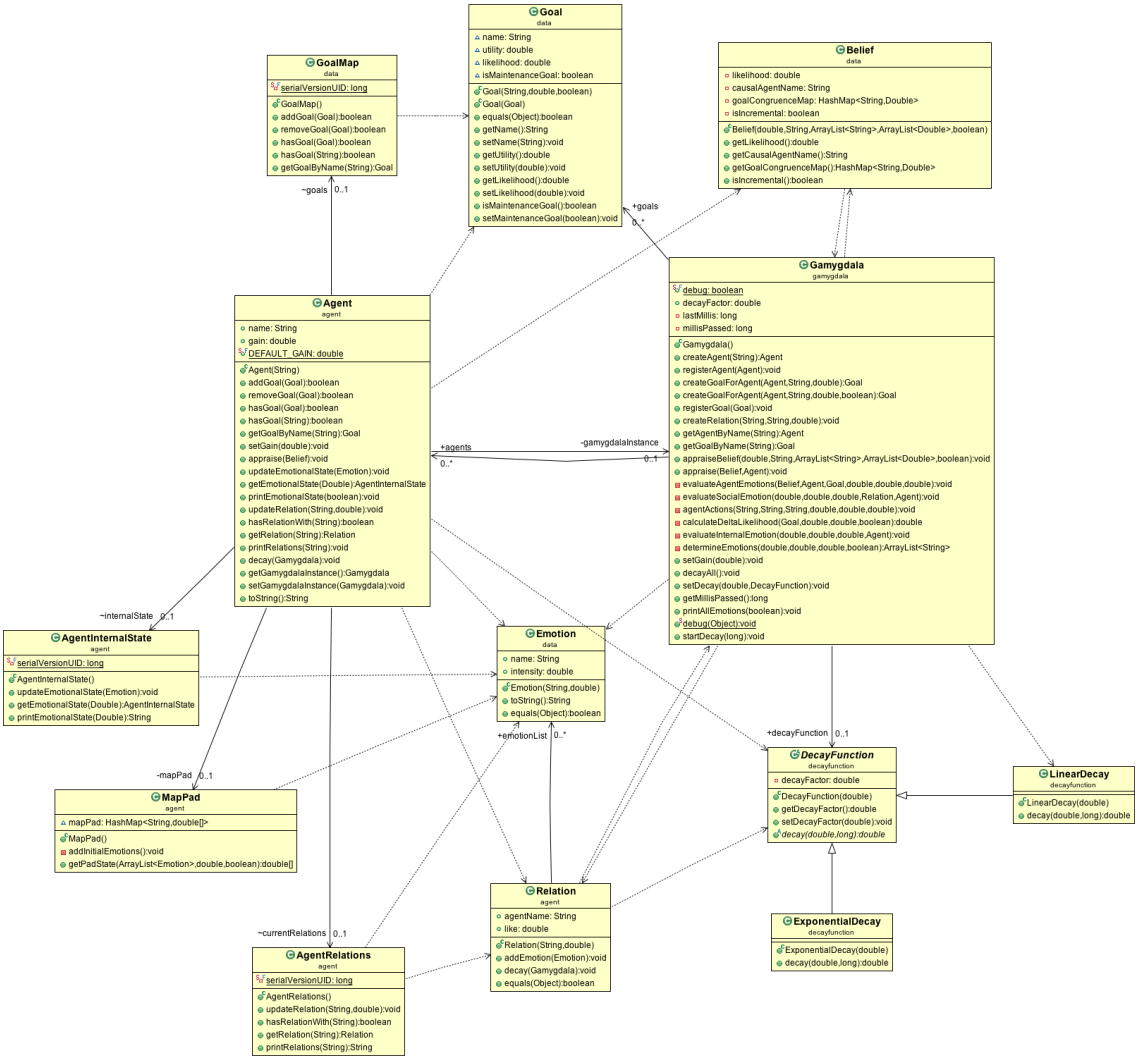

Figure 3: The UML-diagram for the initial port.

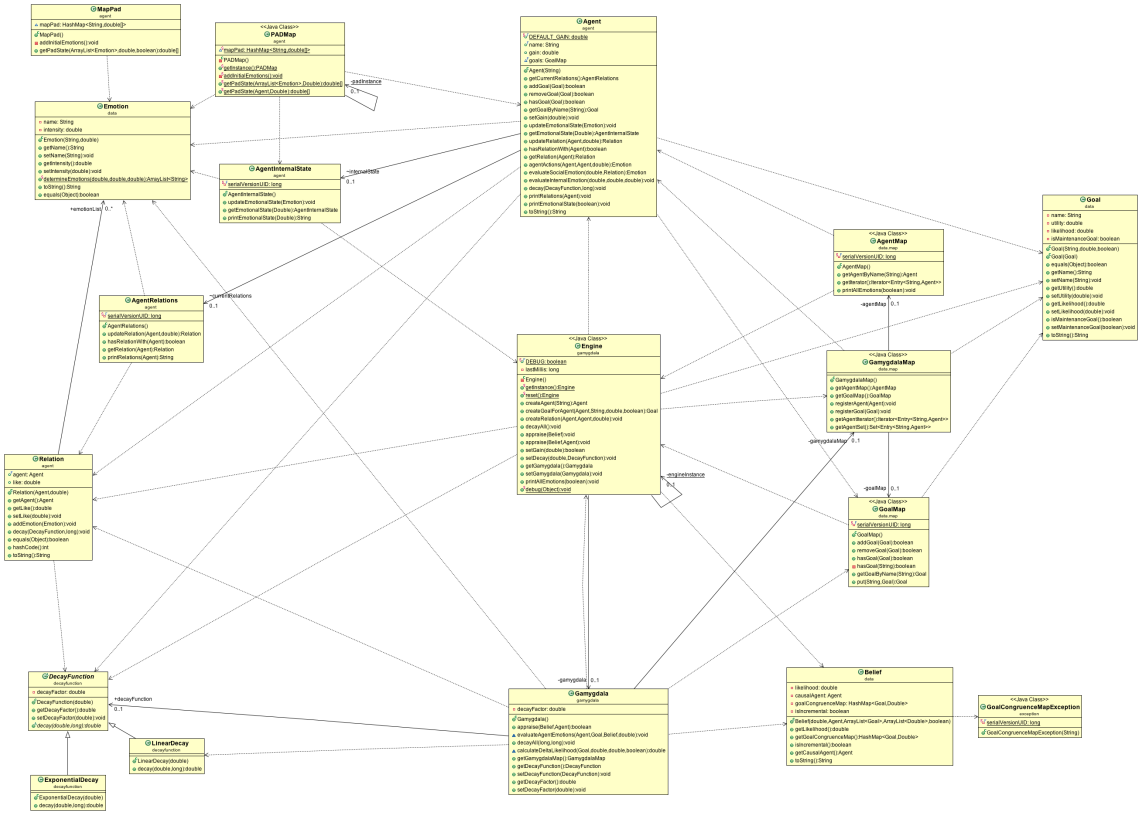

Figure 4: The UML-diagram for the re-factored port.

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, because the original GAMYGDALA already had an architecture. But this is 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.

Unit testing A Unit test determines whether a method still functions the same as intended 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 useful 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.

4 GOAL-GAMYGDALA Interface

Now that you have an understanding of how GOAL works and how GAMYGDALA works, it is possible to understand the architecture of the interface between the two, and the considerations that were made in the process of creating this architecture. There are some significant differences between the way GOAL works and the way GAMYGDALA works. Understanding these differences is fundamental in the process of designing the interface between the two. After all, the goal of the interface is to cross these differences the best it can.

4.1 Architecture overview

The GOAL-GAMYGDALA interface architecture consists of three major parts: the Term Parser, which parses the terms into basic java values, the main interface, which is the interface from GOAL to global GAMYGDALA actions, and the agent interface, which is the interface from GOAL to agent-specific actions.

4.1.1 Term Parser

When a call is made in GOAL, this is done in the form of terms. Terms define the basic structure of a GOAL value, though the only current implementation is in Prolog. Since our GAMYGDALA port is in java, it expects basic java values, so we need to transform these Terms to primary values or lists. This is done by using a term parser. We have used the Strategy pattern for the term parser: we defined a term parser interface that specifies all the types of values that the parser should be able to get from a Term, and we have created the specific prolog term parser implementation. This allows us to easily change the term parser in the case that another language might be supported by GOAL in the future.

4.1.2 Main interface

Since all calls need to be forwarded to the GAMYGDALA Engine, we have created a singleton called GoalGamygdala that mimics the essential actions of the Engine, though the names might be changed to better match GOAL conventions (since the interface is GOAL-GAMYGDALA and not GAMYGDALA-GOAL). These actions then make use of the term parser to parse the terms into values that GAMYGDALA can understand, and then they forward the calls to GAMYGDALA with the parsed values.

4.1.3 Agent interface

There are some actions that are agent-specific. Since it seemed rather faulty to lay the responsibility of forwarding these agent-specific actions with the main interface class, which is mainly responsible for forwarding agent calls, it made more sense to create an interface for the agents specifically. The only responsibility that is laid with the main interface is then allowing GOAL to get to the desired agent interface that matches the agent it wants to work with.

4.1.4 Relation management

Relations are the one specific case where differences between GOAL and GAMYGDALA really show themselves. The main problem is that all GOAL agents work on their own, and they might not be active at the same time. However, agents should be able to define relations to each other regardless of whether they are active or not. To achieve this, relation creation calls are not simply forwarded to GAMYGDALA. First, a check is done to see if the relation target is actually registered in GAMYGDALA. If it is not, the relation is saved and will be applied the second the target gets registered. Because of this, agents don't have to worry about loading times or sheer inactivity of other agents, and they can simply define relations whenever they want. Once these relations become applicable, they will be applied.

4.1.5 Agent management

Because both the relation manager and the main interface need to be able to access the agents, to create the relations and do other things respectively, we have created an agent manager singleton that keeps track of all agents. In that way, both the relation manager and the main interface are capable of accessing the agents when they so desire.

4.2 Design Patterns

Since the interface is not a particularly large piece of software, it did not need many design patterns. Only in cases where we truly deemed it important for future maintenance and expansion, we actively applied design patterns.

4.2.1 Strategy Pattern

The most important usage of a design pattern is the usage of the strategy pattern for the term parser. As mentioned before, GOAL is meant to be able to work with multiple languages. The values of the terms are, however, dependent on this language that GOAL might be working with. Because of that, implementations of the term parser needed to be language-specific. If we were to adhere to the possibility that new languages might be added in the future, we needed to create a basic term parser interface and thus apply the strategy pattern. This way, a different term parser can be created for each language, and these term parsers are interchangeable.

4.2.2 Singleton Pattern

Since the main class of GAMYGDALA is a singleton, it only made sense to make the main class of the interface also a singleton. That way, it could be accessed from all different action executors in the runtime. Also, it would be ensured that all these different executions shared the same instance of GAMYGDALA.

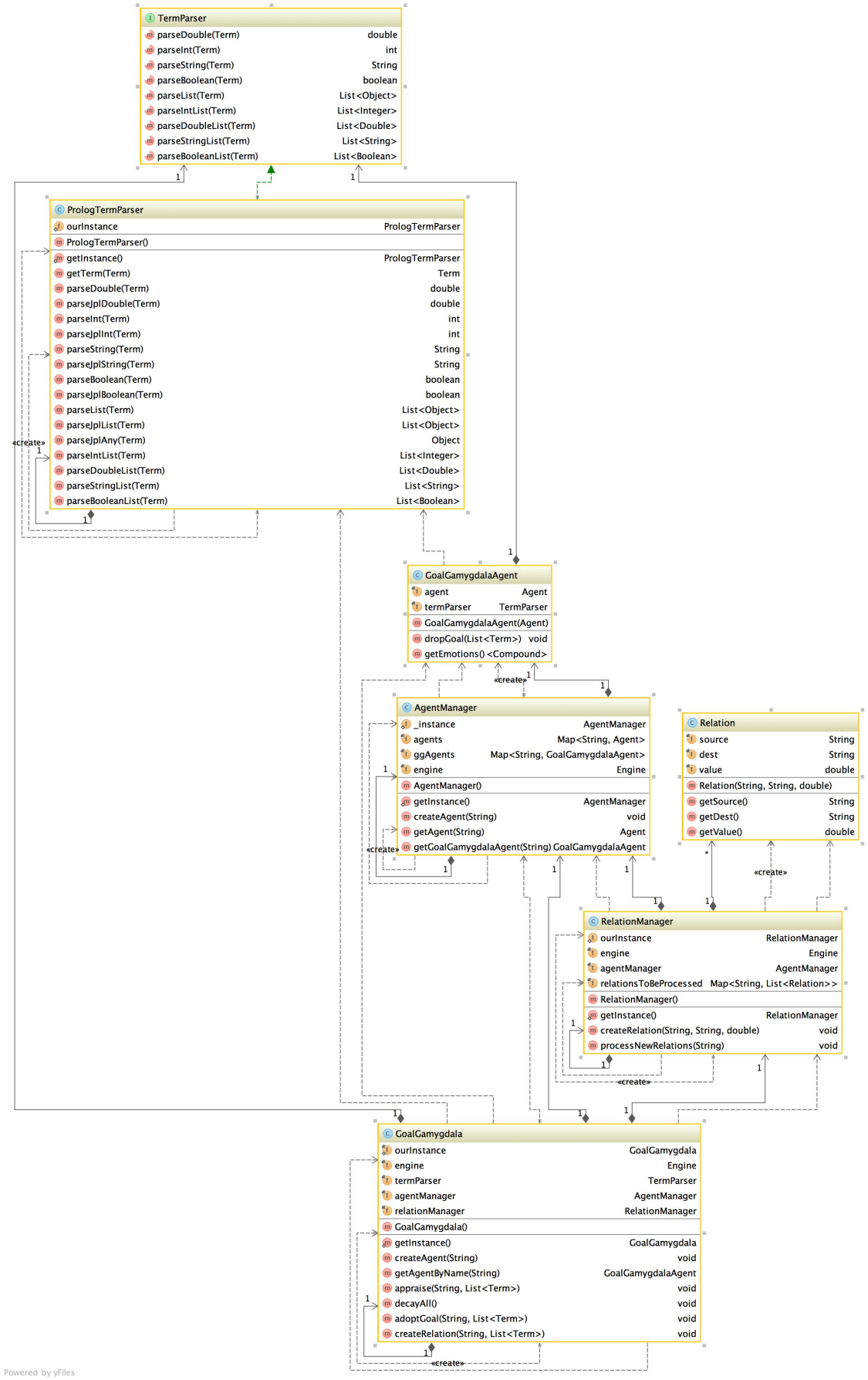

Figure 5: The UML-diagram for the GOAL-GAMYGDALA interface.