Taras Shevchenko National University of Kyiv
Faculty of Computer Science and Cybernetics
Department of Information Systems

COURSEWORK

Speciality 121 Software Engineering

Development of the Multi-Agent System for Game Scenario Design

Author:
3rd year bachelor student
Pavlo S. Volkovyy _____

Academic Supervisor:
Assistant Professor
Dmytro O. Terletskyi _____

I certify that this coursework does not contain any borrowings from the works of other authors without corresponding references.

Author _____

Kyiv — 2018

CONTENTS

INTRODUCTION

Relevance. Simulation. It is one of the most powerful tools for making right choices. In software, it can be seen almost everywhere, yet scientific computing and game design created the most favorable conditions for this mechanism to evolve. However, there are many issues that are caused by a necessity for most simulations to be constantly refined. In game design that brought the whole industry to spend massive amount of money on content generation: assets, storylines, etc. All of them are usually handmade for a simple reason: there is no affordable AI system that can reproduce such diversity and integrity in creative tasks like a human. On the other hand, there is an enormous amount of people spending their time to make narratives. The ability to automate this process even a bit can make it possible to create games much faster.

Practical implementation of the results. The results can be used to implement a framework for a diverse set of simulations. However, a framework that is described in this paper is optimized for usage in game design area. The main purpose of the designed framework is to provide means of creating the realistic simulation and data extraction. Another application of the framework is to test experimental game mechanics without its full specification, or by specifying the mechanics on demand during the simulation process.

Individual contribution. Several abstract architectures for a storyline simulation framework type were introduced:

— **Model0** and **Model0V** abstract architectures were developed for describing the functionality of the framework;

— **Model1** was introduced to clarify some questions that might arise during the implementation of the framework.

CHAPTER 1

GAME DEVELOPMENT

1.1. Developer Roles

Today there are game studios with more than a 1000 employees working on a same project for years. Usually, each developer has one of the following roles [1]:

— level/world designer;

— game writer/storyline developer;

— game artist/content designer;

— programmer/system designer;

— interface designer.

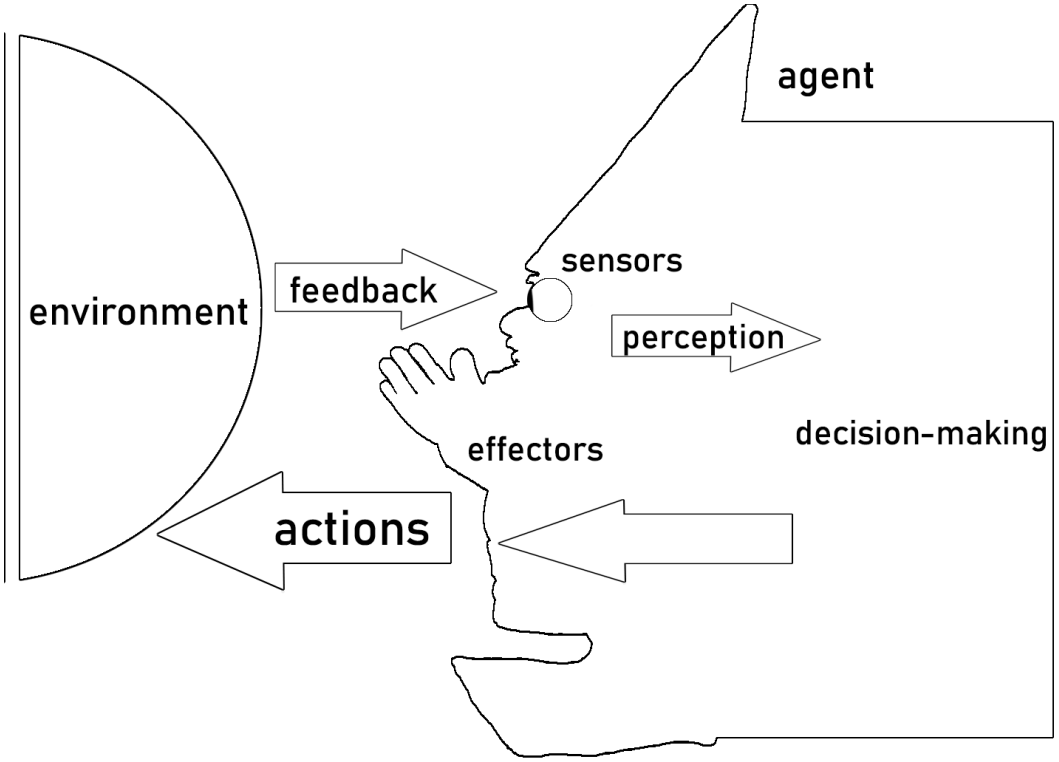

Figure 1.1. Developer roles

Each role is connected to a specific set of tasks: game artists produce content, level designers and storyline developers assemble the content and connect it to the logical

core that is made by programmers, etc. One of the ways to improve overall game development speed is to partially automate the work of storyline developers.

The problem lies in deep interconnection of all roles. Storyline developer strongly relies on level designer who usually is unable to work before the content is created. Being unable to communicate with others may result in storyline developer making a story that is impossible to be efficiently implemented within the game. Therefore, the result of storyline designer's work should be understandable for other team members as well as one should take into the account the capabilities of the people one work with.

1.2. Storyline Creation

All functions of a storyline developer can be specified using a $create\_storyline(W, T, B)$ function, where W is a world view of the storyline developer, T is the task the storyline developer was given, and B represents boundaries that are usually given by other developers in the team. The task of a storyline developer usually consists of statements. For example, a storyline developer might receive the request to create a narrative for an RPG (role-playing game) in medieval France with an army general as a main character. The output should be a detailed storyline, understandable for other members of the game development community.

Usually, a storyline consists of objects and characters descriptions, which are connected by actions that are performed by them or applied to them. Therefore, the storyline creation implies from storyline designer to execute several of the following actions:

— interpretation of a given input to an operable form;
— addition of new elements to the story;
— simulation execution for event sequence determination;
— collection of the useful data (while ignoring the other one);
— interpretation of the simulation result to a shareable output.

A good storyline can often be described as the one that makes sense or is simply

stunning. In order for the story to make sense the author should avoid inconsistencies between the game and the world view of "judges". The stunning storyline can be created with the knowledge about what triggers human emotions. Both of this qualities are common for human but, unfortunately, almost impossible to be effectively recreated with any kind of intelligent systems. Intelligent systems can only efficiently perform the simulation itself.

Games can feature different types of storylines. In some cases storylines are linear, but most of the modern RPGs have tree-structured storylines. In other words, developers have to create a large number storylines with almost identical preconditions in order to provide a player the ability to influence the storyline with one's actions. The problem of tree-structured storylines is that they are still look unnatural. One of the main reasons is because storyline developers (and game-designers in general) quickly become bored by describing the "boring storyline branches". For example, if the player in one of the modern RPGs decides to quit the main storyline and become a farmer, one most likely will not receive detailed quests such as "grow the plants" simply because the developers had no intention to waste their time on creation of the uncatchy part of the game. Even though such quests are simple and can be easily generated, lack of them makes games less realistic and more linear.

1.3. Domain Specific Modeling

One of the important problems in game development is the consequent task execution problem. The necessity of doing one task at a time often makes it inefficient to create big developer teams. The example of such problematic workflow is provided in **Example 1.1**.

Example 1.1. Five people decided to develop an RPG game. The development started in June 2014. The storyline designer generated the concept of the game till the end of July 2014. Design and implementation of the logical core was made by the end of August 2014. Meanwhile, the level designer decided what kind of assets one might need and requested from the artist to prepare them. At the beginning of September

2014 the logical core was completed. Nevertheless, there were no ready assets for the level designer to carry out one's part. Till the end of the month most of the assets were ready, so the level designer finally had a work to do. When the autumn ended, level design was almost completed. The storyline designer began filling the game world with available characters and narratives. In January 2015 the alpha version of the game was created, but after extensive testing several bugs were found. That is why 1 more month was spent on updating the logical core of the game. Overall duration of the game development process was 9 month.

During the particular moment of the time, there were a few people on the development team that had a work to do. One way to solve this problem is to make use of general-purpose modeling languages. The most popular example is probably UML. Due to this languages the new approach became available. The situation from the first example now could be solved as in the following one.

Example 1.2. Let us imagine the similar situation to the previous example. The development started in June 2015. The storyline designer generated the concept of the game till the end of July 2015. The model of the game was created in the first part of August. The resulting model dictated that there is an arcade game logical core to be written, a forest to be created by the artist and the story to be told. Second part of the month was spent on refining that model. The interfaces and internal structure of the logical core of the game were specified. The artist created several forest low-poly assets and the storyline designer made some example narratives featuring the forest. Since the logical core was specified, the artist could already start designing the protagonist. The level designer got all the information to prepare some informal level templates. Two weeks later the final details were embedded into the model. The input format for levels and assets was approved and the first sample backbone storyline created. During the next month the logical core was finished, all low-poly assets were made, and the storyline was extended. The level designer who used to operate with abstract assets created several basic game levels. Next month was spent on testing, detailing assets, narratives implementation. One more month for error handling and improving

the overall game experience. After less than seven month the game was ready.

The main advantage of such approach is that, besides the increased amount of work made by each developer, the overall time of development is significantly decreased. In this example the model that was made on the first stages of the development served only the specification role. In some cases, with the help of code generators and other similar software, it is possible for the model to be interpreted into the final result automatically. One of the main drawbacks is that there are usually large parts of the development life cycle when developers are deprived of integration testing. The extreme example of such approach is DSM (Domain Specific Modeling) that is explained in [2]. The idea is to create the domain-specific modeling language, describe the future application in terms of these language and then convert this specification to the actual product (applicable not only to code) by using code generators and similar software that is either created earlier during the development process or reused by adjusting the configuration of already existing software. This approach requires a lot of work to be done on the first stage of development, but can simplify all other parts of the development life cycle. Among other, domain specific modeling is a great example of how people are willing to sacrifice their time to achieve the greater comfort in the future.

CHAPTER 2
MULTI-AGENT SYSTEMS

One way to sustain the simulation is to create a simulation process that will feature constant communication with a user of the simulation system. Describing game characters in terms of their beliefs, intentions and desires is a common way for humans to understand the world and, therefore, a comfortable way to recreate it in a computer simulation. Using agent notion it is possible to create a simulation environment, that would be easily understandable by a human being as well as use existing frameworks to reduce the amount of code, required to make an operable framework and create tests. This chapter provides information on how it can be achieved.

2.1. Agent Notion

All agent-oriented software is based on the notion of agent defined in [3].

Definition 2.1.1. Agent – computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives.

After disassembling this definition the list of characteristics that are required by each agent can be created. However, most people use the term "agent" when talking about the entity with a bit broader list of characteristics, also known as "intelligent agent" [4]. Main characteristics of intelligent agent are:

— **Autonomy**, usually perceived as an ability to operate without direct intervention as well as control over the internal state.

— **Social ability** – an ability to interact with other agents, usually essential to achieve the goal.

— **Reactivity** – an ability to receive information from their environment and respond to that information.

— **Proactiveness** – goal-driven behaviour.

In [**?**] the multi-agent system is defined as the one that consists of agents, but due to questionable restrictions the definition in this book is a bit different.

Definition 2.1.2. Multi-agent system is any system that includes more than one agent.

Agents are often being disassembled into 3 parts: perception (receiving and pre-processing the input), decision-making and effectors (responsible for influencing the environment). The structure is shown in Figure 2.1. Agents constantly receive information from the environment, process it, then use it to choose the information (depicted as "actions" arrow) they are going to send to the environment. Effectors and sensors are used to formalize and restrain the data that can be passed to and from the environment.

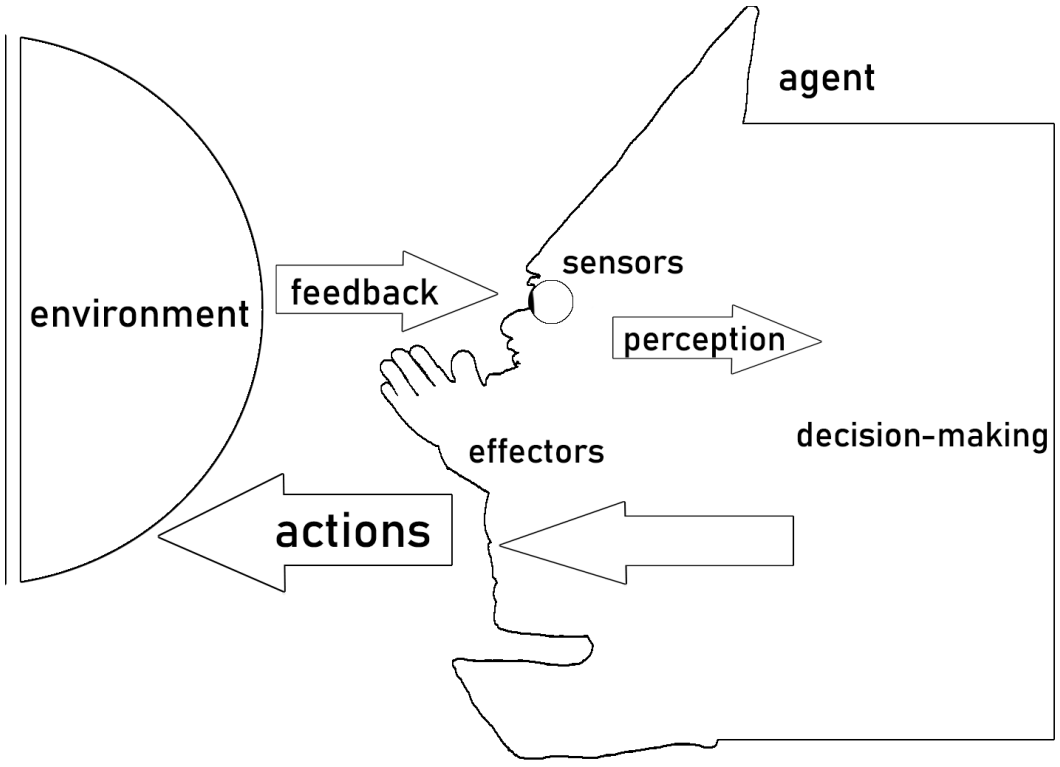

Figure 2.1. Gnome agent

2.2. Environment

There is a huge variety in multi-agent systems of functionality that can be provided by the environment. The environment itself can be implemented as one of the agents. On the other hand, it can be used to simplify the structure of all other agents. For example, environment can contain all internal memory of each agent and give their information to them on demand. There are several classifications of environments based on different parameters. Here are some of them according to [3] and [**?**].

— **Agent capacity.** Single agent environments are deprived of complexity that is common to multi-agent systems. The main problems that are nonexistent in single agent systems are agent communication and simultaneous access.

— **Observability (Accessibility).** The agent can obtain complete information about the fully observable environment's state in any particular moment. Partially observable environments are a lot more common in the real world. According to [**?**] fully observable environment must only provide all the necessary information for the agent to determine the optimal course of actions.

— **Determinism.** In deterministic environment the state of the environment is completely determined by the actions of the agent. In any other case it is called stochastic. The only exception given in [**?**] is uncertainty that is caused by the actions of other agents.

— **Dynamism.** Static environment is guaranteed to preserve its state when no actions are executed by agent. Dynamic environment, on the other hand, usually treats agent's delay of response as idle action. Most multi-agent systems are highly dynamic.

— **Discretion.** Discrete environment has finite amount of possible states. Continuous environments usually feature one or more continuous value that rises the number of possible states to infinity. Despite no continuous environments can be created within digital ones, some of the environments.

— **Episodicity.** The agent's actions does not affect the observable part of the state in episodic environment. No matter which action is executed by the agent

it will not affect any later percepts the agent will be given. The environment that is not episodic is called sequential.

2.3. Agent Architecture

Nowadays, there are a lot of agent architectures used in different practical implementations. In this chapter most of the attention is payed to agent's functionality rather than practicality. That is why this section contains only basic abstract agent architectures. Most of the information in this section was taken from [3].

Consider the environmental instance E, which can be described with finite number of discrete states $E = \{e, e', ...\}$. Now let us assume there is an agent interacting with this environment that can execute one of possible actions $Ac = \{\alpha, \alpha', ...\}$. A run r of an agent Ag in the environment is thus the sequence of interleaved environment states and actions:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_{u-1}} e_u \xrightarrow{\alpha_u} ...$$

R – set of possible runs,

R^{Ac} – subset of R with sequences that end with an action,

R^{E} – subset of R with sequences, that end with a state,

$\tau : R^{Ac} \to 2^{E}$ – state transition function,

$Ag : R^{E} \to Ac$ – generic agent model.

In this architecture, state transition function maps a run to a set of possible environment states in which the environment might be transferred by the last action from the corresponding run. This formula allows to describe nondeterministic history-dependent environments. The agents, however, must be deterministic. Here is a simple example.

Example 2.1. There is a room with a gnome and an apple. The gnome can eat the apple. This environment has 2 states: $E = \{e, e'\}$, e – apple is in the room, e' – there is no apple in the room.

The only agent in this environment is the gnome. Gnome's actions can be described as $Ac = \{\alpha, \alpha'\}$, where α is to do nothing and α' is to try to eat the apple.
And the example run would be $r : e \xrightarrow{\alpha} e \xrightarrow{\alpha'} e' \xrightarrow{\alpha'} e' \xrightarrow{\alpha} e'$.

In this run the gnome does nothing during its first turn, then eats the apple, thus changing the environment state. According to the results of this run it might be assumed, that informal specification of agent function would result in eating apple only if the gnome either did not try to eat or successfully ate the apple on the previous turn.

This architecture is suitable for describing agents with memory thus being able to used for description of any deterministic agent. The only drawbacks of such approach in designing a real agent are:

1. Absence of initial configuration (puts no constraints on the set of describable agents, yet makes overall structure rather complex and sometimes non-intuitive).
2. Lack of information about the internal structure of an agent.

However, there are subclasses of generic agent that can be described in a simpler / more detailed way [3].

— **Purely reactive agents.** These are basically the agents with no memory. Their decisions to take actions are dependent only on what they have perceived on this turn. They can be described by $Ag : R \to Ac$ function and can be easily tested due to being stateless.

— **Agents with state.** In this case, we impose restrictions on the internal structure of an agent. Each agent is given a function $see : E \to Per$ that determines what part of the environment state the agent can perceive. After information is being perceived, it can influence the internal state i of the agent. This process is described by $next : I * Per \to I$. Finally, when the next state is determined, it might trigger one of possible actions $action : I \to Ac$. The architecture is depicted in Figure 2.2.

Besides this architectures, there are also less formal and more human-like approaches. One of them is called BDI (Belief-Desire-Intention) execution model [3]. BDI archi-

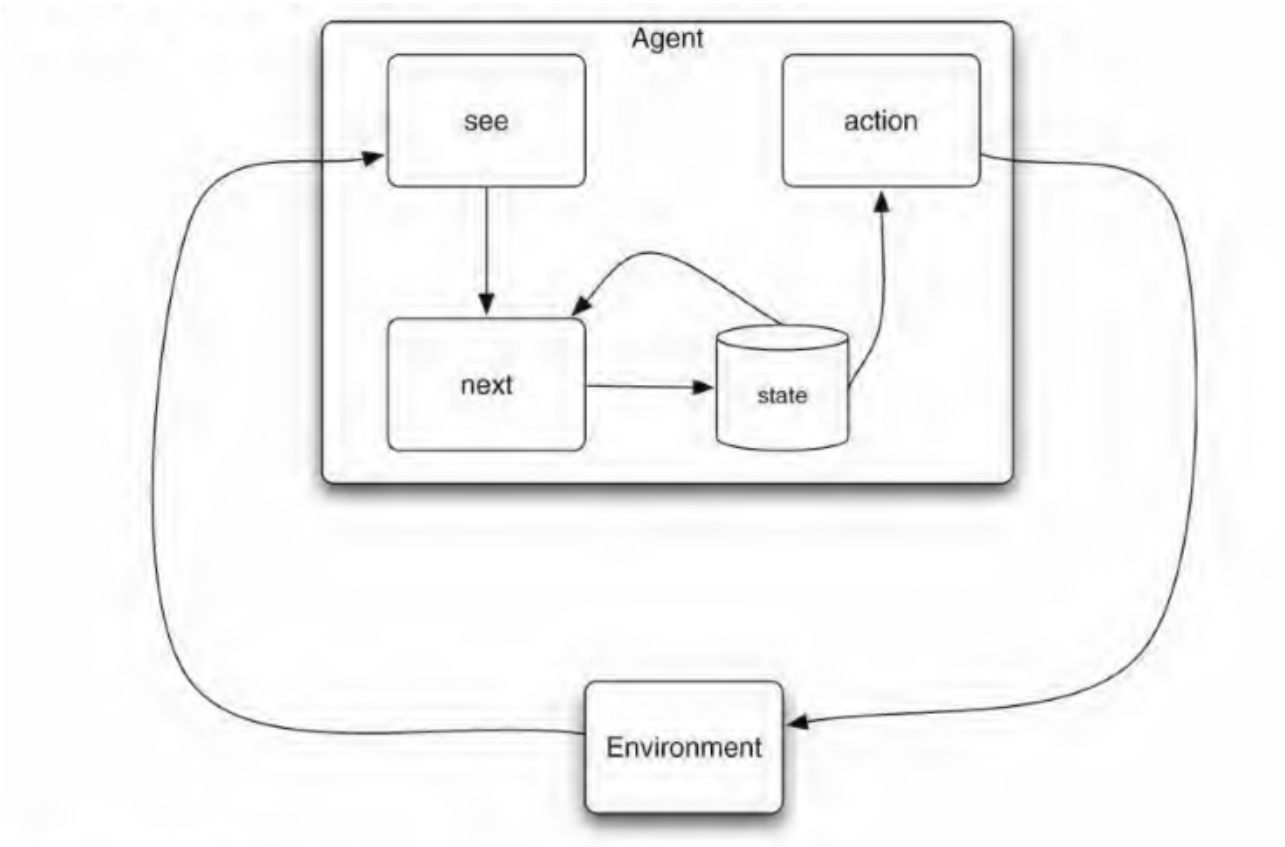

Figure 2.2. Agent with state.

tecture uses terms from a folk psychology to describe the agent's behavioral structure. The 4 key concepts of this model are Belief, Desire, Intention and Plan (see Figure 2.3). Beliefs represent the agent's internal state. Desires are used by the reasoner (logical part of the agent) to determine the optimal course of actions. Intentions consist of actions that are going to be executed by the agent in order for it to achieve some of its desires. And, finally, plans can be described as long-term composite structures, that encapsulate an enumerated set of actions (intentions), used to simplify the belief system [5, p. 150].

2.4. Utility Functions

The essential part of each intelligent agent is goal-driven behaviour. The most common way to determine the goal of an agent is to specify its utility function [3]. The perfect utility function result mirrors agent's success of reaching its goals. The simplest solution is for utility function to evaluate the current state of the environment

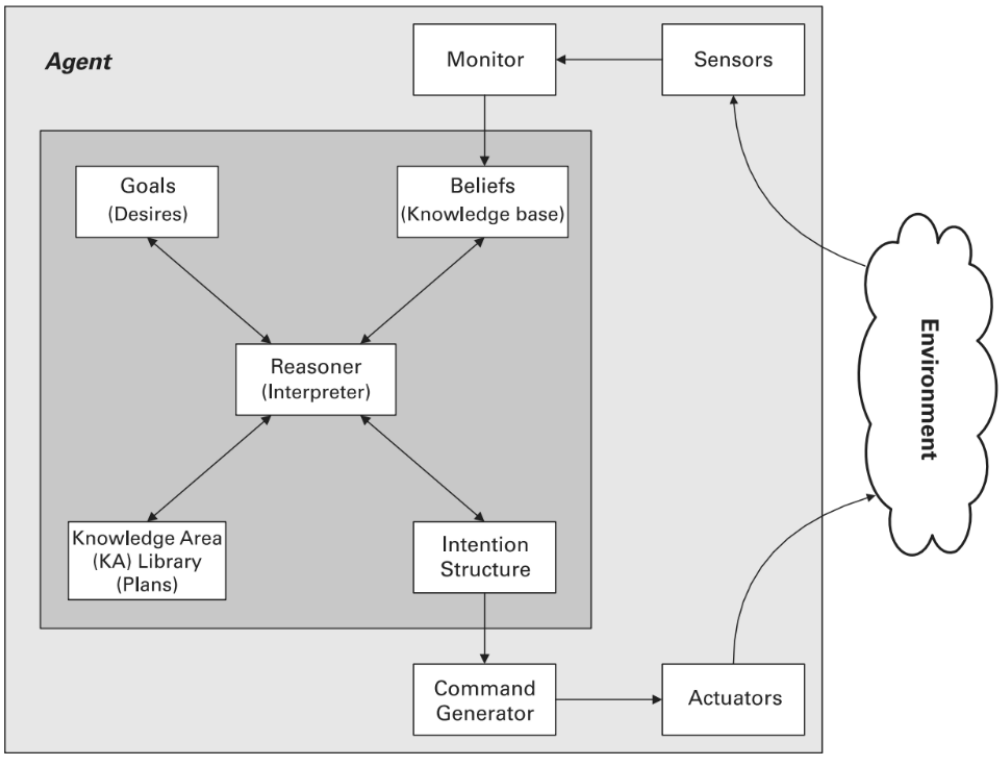

Figure 2.3. BDI agent architecture.

($u : E \to \mathbb{R}$). This kind of utility function proves itself to be inefficient on long-term strategies, but it is still widely used in different fields, such as neural networks. The solution may be for the agent to build a new utility function on top of the given one that will use one's state as an argument. Yet for those who do not want for their agent to feature complex reasoning mechanisms there is a broader class of utility functions which accept the whole run with all the information in it as an argument – $u : R \to \mathbb{R}$. This is the widest class of utility functions. It features such subclasses as predicates ($u : R \to \{0, 1\}$), useful for determining whether the particular goal was achieved.

When agent's utility function is constructed, the only thing agent is left to do is to maximize it. In real environments, which are mostly nondeterministic, highly dynamic and hardly accessible it is possible to operate only with probabilities. For example, sometimes there is no certain answer whether the environment would end up being in a particular state after the agent executes a particular set of actions. Nevertheless, by measuring the probability of this to happen it is still possible to determine the optimal

strategy. Each agent of course might influence the probability of a particular run:

$$\sum_{r \in R(Ag,Env)} P(r|Ag, Env) = 1,$$

where the $P(r|Ag, Env)$ is a probability of the run r to happen with agent Ag and the environment Env. $R(Ag, Env)$ is a set of all possible runs of the agent Ag in the environment Env.

The concept of expected utility is described by the next equation:

$$u_{exp}(Ag, Env) = \sum_{r \in R(Ag,Env)} u(r) * P(r|Ag, Env),$$

where $u(r)$ represents the utility value of a particular run. The optimal agent is considered to be the one that maximizes the expected utility, or

$$Ag_{opt}(Env) = \max_{Ag \in AG} u_{exp}(Ag, Env).$$

2.5. Agent Communication

Agent communication is one of the cornerstones of multi-agent systems. Since each agent is autonomous and features goal-driven behaviour, the overall communication mechanism is much more "human-like": instead of executing requests, it becomes more of a deal-making utility-centered conversation. But for that conversation to exist agents at least should understand each other. To make it easier to design the agent communication, different agent communication languages were created. The most popular are FIPA-ACL, KQML and KIF [3, 5–7].

2.5.1. KQML. KQML stands for Knowledge Query and Manipulation Language. This language represents a message as an object. Each message has a performative that specifies its type and a list of parameters, which are depicted as "attribute:value" pairs.

Some attributes of KQML

№	**attribute**	**meaning**
1	:content	content of the message
2	:force	whether the sender of the message will ever deny the content of the message
3	:in-reply	whether the sender expects a reply, and, if so, an identifier for the reply
4	:reply-with	adding the preferable response specification.
5	:sender	sender of the message
6	:receiver	intended recipient of the message

Therefore, a message in KQML may look like this one:

$$
\begin{aligned}
&(\ \text{ask-one}\\
&\quad :\mathit{content}\ (\mathit{COLOR\ GNOMEHAT}\ ?\ \mathit{color})\\
&\quad :\mathit{receiver\ tailor}\\
&\quad :\mathit{language\ LPROLOG}\\
&\quad :\mathit{ontology\ gnome-info}\ )
\end{aligned}
$$

"Ask-one", in the message above, is a performative. There are around 50 performatives in KQML, each describing the corresponding message type.

The dialogues in KQML can be either synchronous or asynchronous. The types of dialogues are represented on Figure 2.4.

2.5.2. FIPA. Since FIPA ACL was designed for the same purposes as KQML, this two languages have much in common. The goal behind the creation of the FIPA standard is to utilize it in the development of an agent communication language (FIPA-ACL). FIPA-ACL is superficially similar to KQML: it defines an "outer" language for messages, feature 20 performatives for defining the intended interpretation of messages,

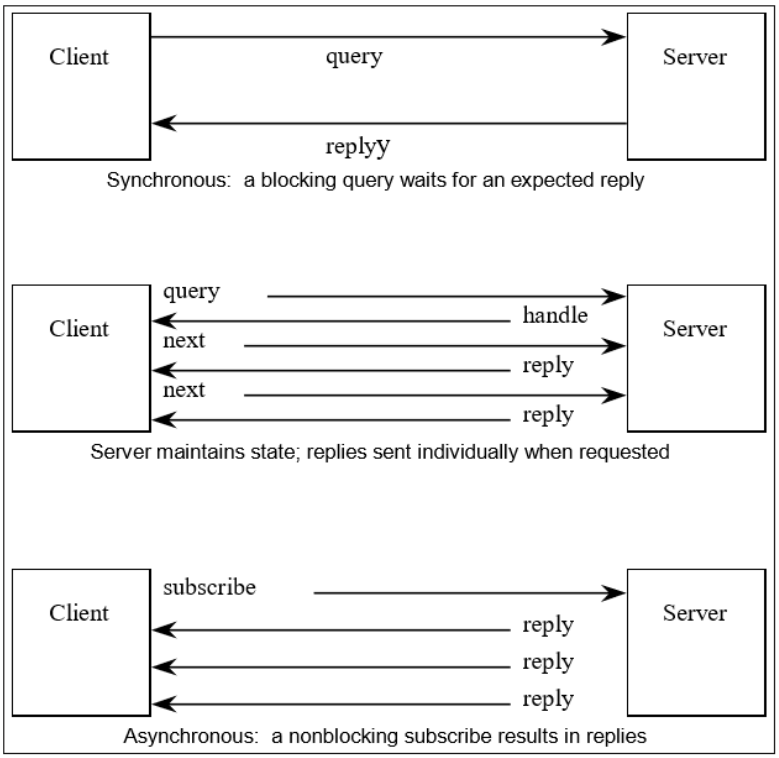

Figure 2.4. BDI agent architecture.

and it does not mandate any specific language for message content. In addition, the concrete syntax for FIPA-ACL messages closely resembles that of KQML. For example,the example KQML message from above when encrypted in FIPA-ACL using XML markup language is presented below:

$$
\begin{array}{l}
< \text{envelope} > \\
\quad < \text{params} index = \text{"1"} > \\
\quad\quad < \text{to} > \\
\quad\quad\quad < \text{agent-identifier} > \\
\quad\quad\quad\quad < \text{name} > tailor < /\text{name} > \\
\quad\quad\quad < /\text{agent-identifier} > \\
\quad\quad < /\text{to} > \\
\quad\quad < \text{from} > \\
\quad\quad\quad \ldots \\
\quad\quad < /\text{from} > \\
\quad\quad \ldots \\
\quad < /\text{params} > \\
< /\text{envelope} > \\
< /\text{message} > \\
\quad < /\text{performative} >
\end{array}
$$

$ask - one$
< /performative >
< /content >
?
< /content >
< ontology >
$gnome - info$
< /ontology >
...
< /message >

2.5.3. KIF. The knowledge interchange format (KIF) is a simple logic-based language that gained high popularity in expert systems, databases and intelligent agents. The initial design objective of KIF was to create a "mediator" language, useful for translating messages between other languages. The simple example featuring this language is given below.

(> (- (length sword) (shaftLength sword)) (- (length knife) (shaftLength knife)))

This example states that the sword has the longer blade than the knife does. Despite the existence of this communication protocols there are a lot of cases where it is much more rewarding to create personalized communication protocol for a particular system. However, KQML or FIPA-compliance can make a multi-agent system easily understandable and mergeable with others.

2.6. Software Overview

Since multi-agent systems use to be more complex and are usually less efficient than, for example, systems based only on functional or OOP paradigm, agent oriented programming is not among the popular programming concepts. Nevertheless, due to being used in various scientific researches and AI based systems, agent notion was used in a multiple frameworks and libraries. The comparative table of them can be found in **Appendix A**.

The problem with most frameworks is that they were originally designed to fulfill

the certain set of tasks: and none of the tasks was similar to the one given in this paper. Another problem is the commercial license. The most efficient systems have the highest prices. On the other hand there is an ability to completely customize open-source software.

Frameworks that are written to be used with custom or less popular languages are usually the least customizable and are useful only for simulating software, where productivity is of lesser concern than the ability to create the system rapidly.

The leading frameworks (in terms of the quantity of users) are usually implemented for Java or/and C++ programming languages that allows users to customize them in different ways.

CHAPTER 3
DESIGNING THE MULTI-AGENT SYSTEM

3.1. Task Specification

In the first chapter it was pointed out that since it is nearly impossible to create intelligent system that would carry out all the creative processes, it is only reasonable to augment it by using the story designer's "computational power". The task can be split in two parts:

1. Design the system, that will supplement some of the story designer's functionality. Later to be mentioned as **System0**.
2. Design the game development system (featuring other developers) in which *System0* can function and be useful. This one will be called the development workflow.

3.1.1. The development workflow. To design *System0* it is firstly needed to understand the exact features this system needs to help other developers the most. For that let us analyze *the development workflow* in detail.

As it was mentioned in the first section of chapter 1 , there are quite a few roles on the development team. In *the development workflow*, most of this people will exchange information with *System0* in different points of time. Consider a couple of examples

Example 3.1. The only person who influences the result of a storyline creation is the story designer. Other just process what was created. Each member has responsibilities shown below.

— **Storyline designer** provides the initial configuration to a storyline creating system (*System0*). During the simulation one provides information requested by the system.

— **Level/World designer** designs the game world according to the output of

System0.

— **Game artist** provides essential content to the level designer.

— **Game designer** analyzes *System0*'s output and then determines the optimal game mechanics. Afterwards, programmer creates the logical core of the future game and interprets the final story so it can be embedded into the game.

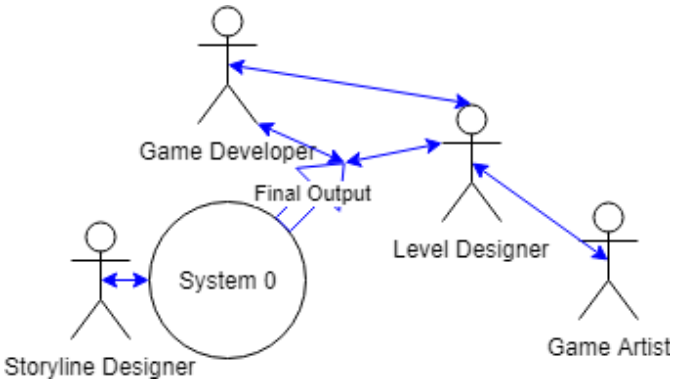

Figure 3.1. Example of game development interaction diagram №1

Dependencies between the developers are depicted on Figure 3.1. Arrows represent data flow during the development process.

That is one of the least interactive examples. On the other side, there can be workflows, in which *System0* is actively will communicate with every member of the development team.

Example 3.2. In this example,which is visualized in Figure 3.2, every team member exchanges data with *System0*. The list of responsibilities changes as well.

— **Storyline designer** does mostly the same actions.

— **Level/World designer** designs the world based on *System0's* output, but this time one is given an ability to have an impact on *System0's* result by specifying environmental details. For example, one can specify restrictions before the start of *System0* or be questioned by *System0* during the storyline creation.

— **Game artist** must provide the list of available content to *System0*. Also can put several restrictions on the amount of the new content that can be added. For example, if artist already has a model of a bush and a tree, one can allow

System0 to embed 2 random objects into the story that would need to be created by the artist afterwards.

— **Game Developer** can provide *System0* with the information about the game mechanics to achieve more desirable output. That can be highly useful if the game's logical core is already written.

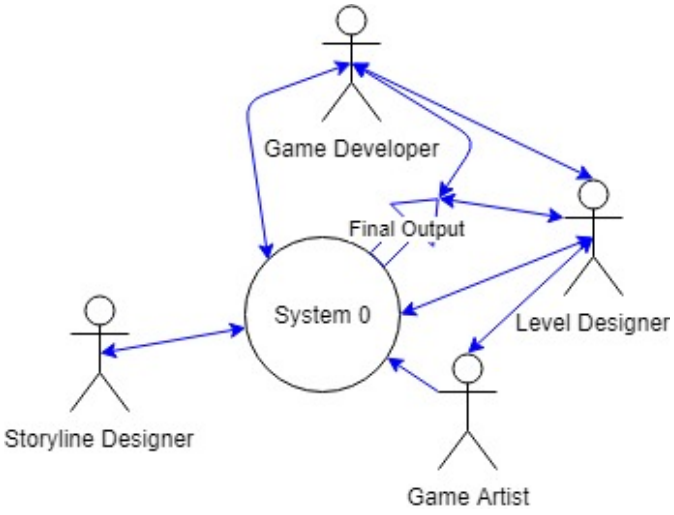

Figure 3.2. Example of game development interaction diagram №2.

This example closer resembles the real development process. The main reason to this is an ability to take into account the capabilities of each individual on the development team. In this case, *System0*'s output can theoretically be an actual game. Also this example allows minimizing the dialogue between the story designer and *System0*. The data retrieval process for *System0* can be partially distributed among other members of the development team.

In conclusion, attempts of reducing the concept to a system with a concrete desired functions and parameters were not successful – there are to many possible cases for such system to stay efficient. The diversity of possible uses makes it effective to design an expandable framework. This approach can result in an ability of end users to create a personalized system by filling it with required functionality.

3.2. Creating Structure

In previous chapters it was determined that the aim of this paper can be achieved by creating an agent-based framework, and the first step to create a framework is to specify its structure. The current goal is for this structure to model the structure of games. So here is a simple example of a couple of different games from different times.

PAC-MAN. Released in 1980, this simple arcade game is not hard to be described. The goal is to collect all little yellow dots while avoiding collision with monsters.

The Witcher 3. One of the most complex RPG games released in 2015. Features enormous game world, thousands of characters, and much more assets.

The idea is to take a closer look at this 2 games and try create the optimal classification for their entities. Let us start with the easy one.

As simple as it is, PAC-MAN features 3 classes of entities, little yellow dots, monsters, and the PAC-MAN itself. Since we are using agent notion, the next step will be to describe this entities in terms of BDI execution model. Let us start with PAC-MAN. It has at least 1 obvious goal – to collect all yellow dots. As for its intentions and beliefs, they are determined by the player. Monsters on the other hand have a desire to reach PAC-MAN and individual plans of doing so. It seems like the only entity that is left is the small yellow dot. It has no obvious goals, it is not autonomous, and that lead us to a rational conclusion it should not be represented as an agent. Therefore, we need already create 2 classes of entities. Let us call them `GameAgent` and `GameObject`. But there is one more entity. The environment in which all other entities are situated – the map. In general, the only functions it must have are to contain other objects and provide the means for their execution and communication. But in our case we may combine it with an actual map, featuring walls (because walls can not be influenced by anything) to simplify the structure. The name for this type of entities will be `GameEnvironment`. At this point the results can be shown in Figure 3.3.

Now let us look how does this simple structure can be applicable to the second game. The `GameAgent/GameObject` classification mechanism stays the same – all agents are entities easier to be described with their beliefs and desires than with al-

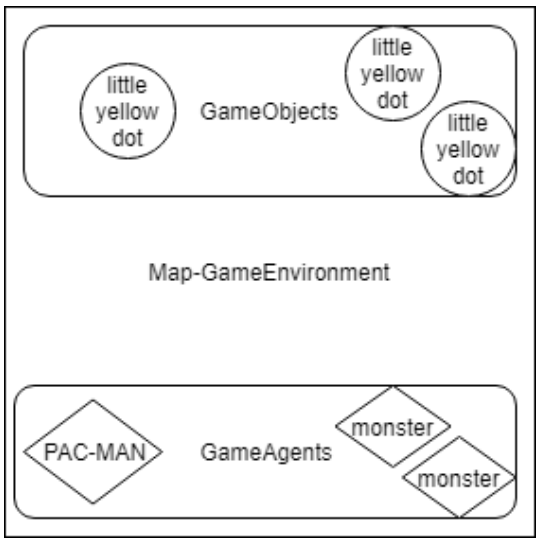

Figure 3.3. Agent/Object/Environment structure for PAC-MAN game example.

gorithms. However, there are entities with the similar functionality to `GameAgent` or `GameObject` that encapsulate other agents, as it does `GameEnvironment`. The obvious example is the house that is implemented as a single `GameObject`, but may contain other `GameAgents` and `GameObjects` inside. Therefore, each game environment can also be a `GameObject` or `GameAgent` in another game environment. Also every `GameAgent` entity extends `GameObject`.`GameAgent` can be described as `GameObject` with added reasoning mechanism and individual computational power. Class relations are depicted in Figure 3.4. Despite the initialization, there was no specification provided to the classes. The set of already defined classes along with their specifications can be later referenced as **Model0**.

Model0 specification

3.2.1. Communication between Model0 entities. The communication is a process that includes sending data to another structural entity. The list of possible events that trigger communication mechanisms in GameEnvironment include:

1. The `GameAgent` executes an action.This action is the most general one, yet in most games it has a specific meaning. Consider the following situation: you are in the train trying to explain someone a theorem, but all the noises and voices around stop your constantly ruin your attempts to deliver the information to your interlocutor. The environment is responsible for the choice whether your

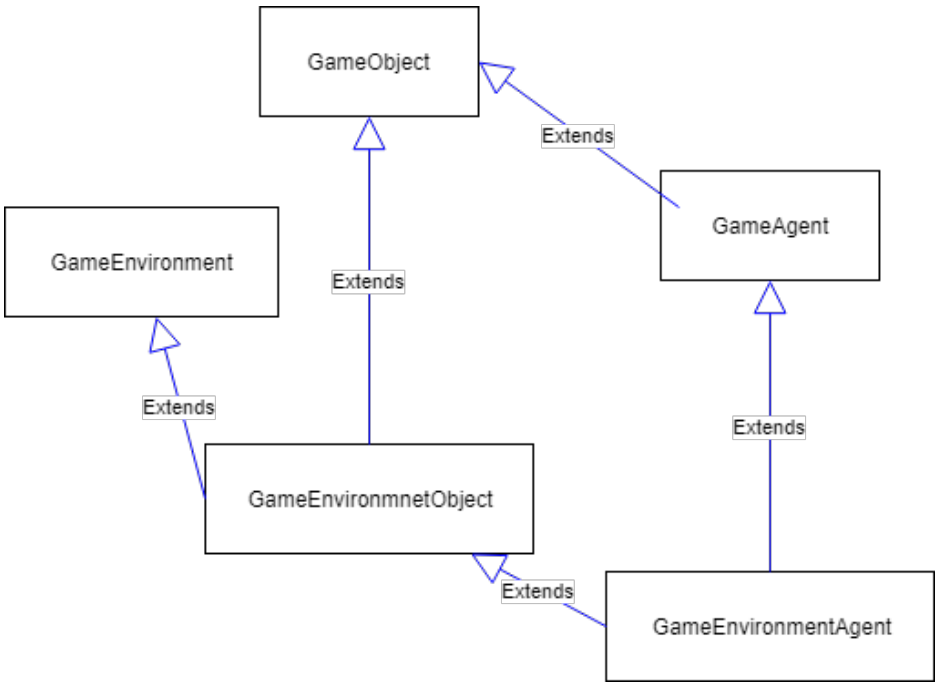

Figure 3.4. Class inheritance for Model0.

actions succeed or fail to execute. That is why in reality, and, therefore, in most games, each action attempt should be double-checked by the environment itself. Also environment controls whatever response this agent might get afterwards. It requires the communication mechanism between two `GameAgents` to look similar to the one in Figure 3.5. If M is the set of all possible messages

Figure 3.5. Message passing example.

(including message without data) and E_i is the state of the environment (that includes every information environment possess, including accessible information from its objects), then each message, sent from one object to another one undergoes transformation, specified by a function $correct : M \times E_i \to M$. However, in reality, while processing the message, environment might send and collect several messages from any objects that it contains.

2. The `GameEnvironment` executes an action. In most games time is one of the

global parameters. The responsibility to change time in most cases should be granted to the environment itself to avoid the possibility of loosing or destroying the agent and the following problems. By making `GameEnvironment` a time-managing entity it becomes a perfect choice for triggering time-specific events. Time-specific events may include creating, destroying, modifying and notifying aggregated entities of the `GameEnvironment`.

On the other hand, time can be managed by some aggregated entity like a `GameAgent`, yet , while simplifying `GameEnvironment's` structure, this design choice might make the overall framework less understandable.

3.2.2. GameObject. This class (depicted in Figure 3.6) should describe every possible object that have state and can be created, destroyed or changed during the simulation. Now it is possible to assemble it piece by piece by mapping different examples on PAC-MAN entities, which are mentioned earlier in this chapter.

— Properties.

Each object is defined by its properties. These properties serve as arguments for other objects/agents. For example, each object in PAC-MAN must have its own position. There are two types of properties:

Definition 3.2.1. External properties. This kind of properties is directly accessible to the environment (*read*, *modify*, *add* and *remove* permissions), yet can be inaccessible to the containing agent itself. The only way to to perform any direct access action (*read*, *modify*, *add* or *remove*) for the agent is to send a message request to the environment.

Definition 3.2.2. Internal properties. These properties can be inaccessible to the environment, but must be directly accessible to the agent itself. For example, if a box contains an apple, environment has no need to know what is it in the box until it is opened. When the box is opened and the insides of the box becomes an external property, environment can decide whether the apple could be in the box and, if it violates some of the rules, simply deny it.

— Event handlers

Definition 3.2.3. Event handlers. These handlers are responsible for responses of the `GameObject` to external messages that are created within the `GameEnvironment`. For example, when someone tries to fire a gun, the corresponding event handler of a gun would determine the appearance and the direction of the bullet.

3.2.3. GameAgent. `GameAgent` (depicted in Figure 3.7) features all the structural details from the `GameObject` class. External properties have the same specification. However, internal properties are also used to contain agent's belief structure, so it will be easier to refer to them as beliefs or the internal state. At this point, internal structure of a `GameAgent` may remain undefined. However, since there should be some way for communication with environment, `GameAgent` must be given a `GameEnvironment` upon aggregation.

3.2.4. GameEnvironment. The `GameEnvironment` class (depicted in Figure 3.8) is the least clearly specified. For example, in the PAC-MAN model it was used to represent some of the most simple objects (maze blocks). The pure `GameEnvironment` class consists of `GameEnvironmentMessageHandler`, `GameEnvironmentObjectContainer` and `GameEnvironmentResourcePool`, each of which executing its own functions.

— **GameEnvironmentMessageHandler** is responsible for message control. It receives messages, determines how to change them (based on a set of rules it is given) and who should receive them. The mechanism of changing messages gives the `GameEnvironment` a complete control over agent interactions within the `GameEnvironment`.

— **GameEnvironmentObjectContainer** is responsible for containing objects. Its functions may include distributing memory as well as computational power between entities. It also should make it possible to dynamically create and to destroy entities.

— **GameEnvironmentResourcePool** is a container for resources that are given to the environment.

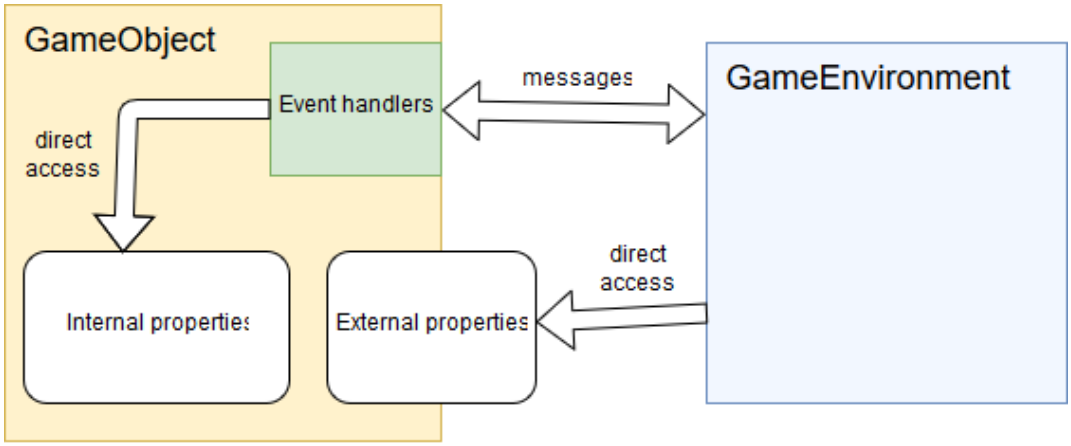

Figure 3.6. GameObject structure in Model0.

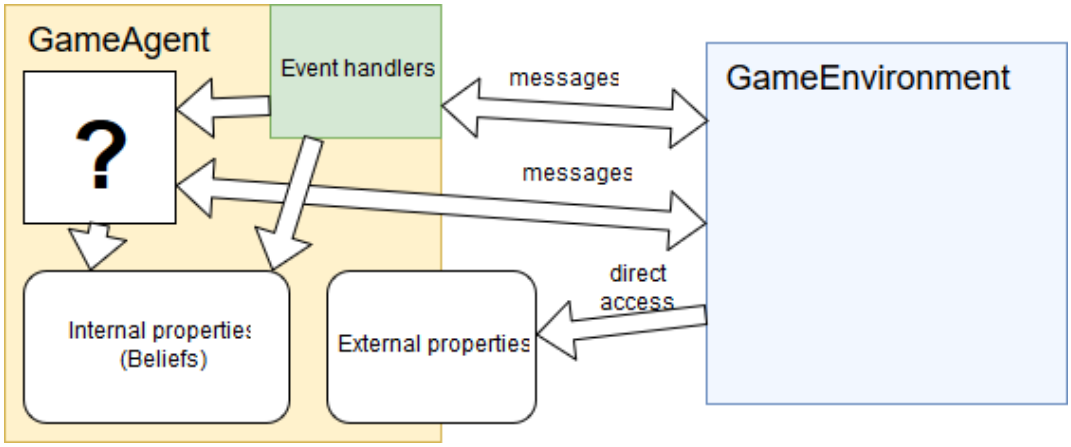

Figure 3.7. GameAgent structure in Model0.

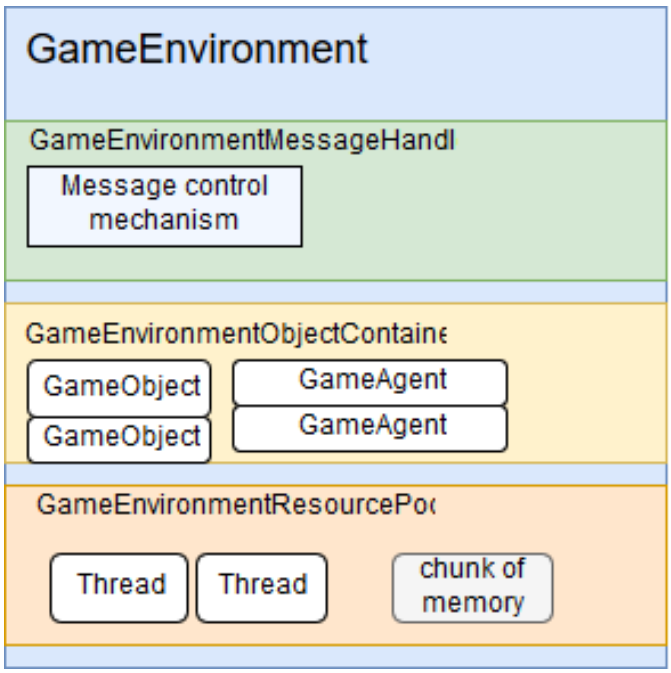

Figure 3.8. GameEnvironment structure in Model0.

3.2.5. GameEnvironmentObject/GameEnvironmentAgent. These are the most complex classes in *Model0*. The goal of these classes is to provide an ability of creating a recursive depth in the system by creating the `GameObject` or `GameAgent` that also includes `GameEnvironment` functionality. The `GameEnvironment` that contain `GameEnvironmentObject` (or `GameEnvironmentAgent`) is called external environment. The `GameEnvironment` that is a part of the describable entity is called internal environment. The described classes behave similarly to the `GameEnvironment` class but also feature the unique subclass, `GameDoubleAgent`, or `GameDoubleObject` respectively. This class differs from `GameObject` (or `GameAgent`) in ability to send messages to both internal and external environment. The feature creates a connection that can be used to pass data between environments.

3.3. Human Integration

In this section there is a description of a mechanism that allow *System0* to communicate with external reasoning and information retrieval services. These services to some extent imitate the creative process in human mind.

On some level of abstraction, association and attention are important concepts in human thinking process. Attention to an entity creates associations with it, and associations are the main mechanism to transfer attention. When we want to create a game plot we do not need to create a completely filled environment to simulate – all we need is to follow our associations. In imagination entities are created on demand and destroyed after there is no attention left to them. That is how we can drastically optimize the simulation process while using human mind based mechanisms.

Let us compare human mind creative process to the corresponding example based on *Model0* in order to expand the later one to contain the former one's functionality. The storyline-creator's activity can be decomposed into several smaller activities. To simplify the terminology these small activities will be referred as "voices". Let us consider the example voice set depicted in Figure 3.9:

— **The thinker's voice** is used to answer questions, for example, "can a chicken

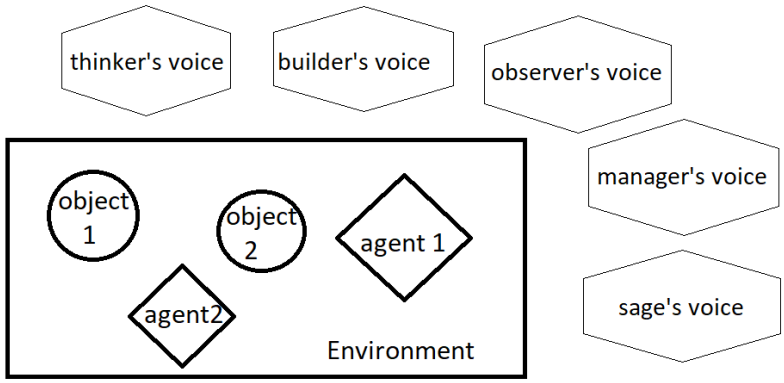

Figure 3.9. The set of voices to be used to model human thinking process.

fly?".

— **The builder's voice** is used to generate new entities and destroy the useless ones. Creates the chicken.

— **The manager's voice** checks the simulations for exceptions and errors. Handles system's response on the flying chicken.

— **The observer's voice** collects the most useful data for the final result. Creates the legend of the flying chicken.

— **The sage's voice** retrieves requested information from a knowledge base. States that usually chickens can not fly, but they are yellow, small, and produce the sound, contained in ‘‘`chicken_sound.mp3`’’.

Each of this voices can be supplemented to *Model0* as interface that can either be automated or used to transfer information between human and simulation. In terms of simulation framework this voices may correspond to the following classes:

— **GameThinkerVoice** is called whether there is no method defined to handle an event.

— **GameBuilderVoice** is used to fill the environment with appropriate `GameObjects`, `GameAgents`, `GameEnvironmentObjects` and `GameEnvironmentAgents`. Also it can dynamically add properties. To separate these 2 activities it can be divided in 2 voices: `GameCreatorVoice` and `GameRefinerVoice`.

— **GameManagerVoice** coordinates the actions of another voices. Handles er-

rors in case other voices are malfunctioning. Also it can be responsible for pausing, stopping, and restarting the simulation.

— **GameObserverVoice** retrieves and processes useful simulation data while having full access to every bit of information in the simulation.

— **GameSageVoice** is used whether there is a need to define any properties of an entity.

Messages to the voices can be either sent by `GameAgents`, `GameEnvironments` or by another voices. Messages from the voices bypass the standard `GameEnvironment` *correct* procedure.

Definition 3.3.1. Model0V is an extension of *Model0* that is created by including voice classes described earlier. In other words, *Model0V* = *Model0* + Voices.

3.4. Framework Design

This section describes how to move from the abstract architecture created earlier to a more specific framework design. However, the design should not connected to any particular programming language.The results off this formalizing stage will be referenced as **Model1**.

Model1 specification

At first there are more detailed specifications of internal structure of each class that was introduced in previous chapters.

3.4.1. GameObject. `GameObject` class is not given by default any computational power. When event handler is executed, it uses computational power of the process that triggered this event handler. `GameObject` should not be active unless it was "disturbed" (the event handler was triggered). However, when "disturbed", `GameObject's` response is not bounded in time. For example, once activated, alarm might be sending alarm messages until it will be deactivated. To handle such "disturbances", `GameEnvironmentBroadcaster` class is created (one for each environment). The purpose of this class is to send specified message to each registered `GameObject` with specified time interval in between. Each `GameObject` can send a message to the

GameEnvironment with the desired return message and invocation interval to attempt registration within the GameEnvironmentBroadcaster. If the registration is successful, GameObject receives the corresponding message. The mechanism of deregistration is similar. In any moment of time GameObject can be also deregistered from the GameEnvironmentBroadcaster by the GameEnvironment itself. In this case it still receives a message with information about the deregistration event.

External and internal properties are saved as two map containers with name as a key to improve access speed. The environment can get, set, create and delete external parameters. The map container for internal properties is useful in case *Model0V* is being taken into account. Events are passed as a message, response is given as a message as well.

3.4.2. GameAgent. One of the key differences between GameObject and GameAgent is that GameAgent can initiate communication with its environment. There are no stateless GameAgents. GameAgent's state should feature at least the assigned GameEnvironment. GameAgent is should be implemented as autonomous entity, therefore the easy solution is to assign individual thread to each agent. Inefficiency of this solution is mostly seen when there is a huge amount of agents that can not influence the environment. Alternative solution is shown in the next subsection.

3.4.3. GameEnvironment. It is easier to specify GameEnvironment part by part.

— **GameEnvironmentMessageHandler** is used to analyze all messages that are passed through the environment. This peculiarity does make GameEnvironmentMessageHandler the main bottleneck of the system. To solve this problem, GameEnvironmentMessageHandler can use computational power of the sender agent when a message is passed. Also it should feature the individual computational power in order to maintain the environment when all other agents are deactivated.

Along with sending messages, this part can invoke setters and getters of any entity in the corresponding GameEnvironmentEntityContainer.

— **GameEnvironmentEntityContainer** should implement the effective mechanism of resource distribution among `GameAgents`. Since every time the message to be passed `GameEnvironmentEntityContainer` *find* function is invoked. That grants the `GameEnvironmentEntityContainer` for example, the ability to collect useful data about how many times the particular `GameObject` was invoked in the nearest period of time, store that data in the `GameObject's` external properties and use that data to calculate the amount of computational resources the agent will be given.

— **GameEnvironmentResourcePool** is the module in which all resources as well as resource-giving classes are being held and given out. In particular, it contains `GameEnvironmentBroadcaster` object.

3.4.4. GameEnvironmentObject/GameEnvironmentAgent. The peculiarity of `GameDoubleObject` has additional *setInternalMessage* function that can be invoked by its event handlers. To send a message to the external environment after being "disturbed" by the internal one or vice versa the `GameDoubleObject` should have the destination of the messages which it can send, therefore it should be given internal `GameEnvironmentMessageHandler` as well as the external one upon creation similarly to `GameAgent`.

3.4.5. Advantages of Using *Model0V* Voices in *Model1*. Attention driven resource distribution. To achieve better distribution of computational resources each agent can feature the external property of attention, assigned and constantly changed by the `GameObserverVoice` entity. If the attention to the entity drops below minimum, it is reasonable to exclude this entity from the equation by destroying or "freezing" it.

3.5. Choosing the Framework

As it was explained in the previous chapter, there are a lot of agent based frameworks, yet most of them are designed for a narrow set of tasks. To choose the framework correctly one should know what features one need in the multi-agent system. Each feature or concept can be either provided by an operational system, by a programming language, by a 3rd party library or by a framework itself (Figure 3.10).

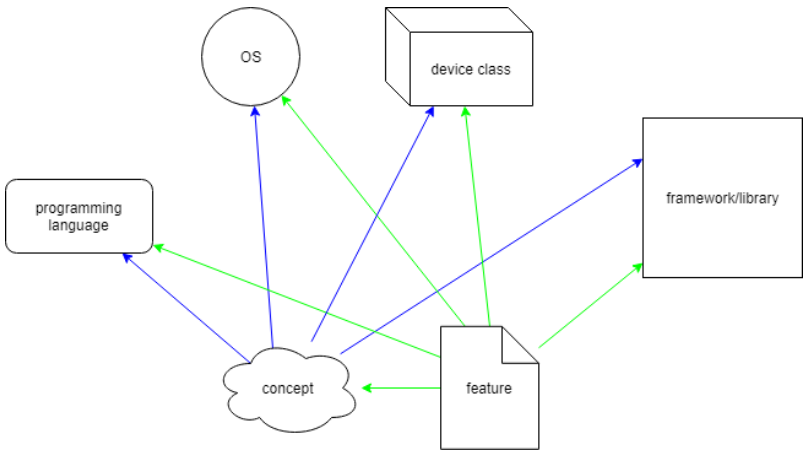

Figure 3.10. Possible feature containers.

By looking through the frameworks it can be seen that some of them are implemented for multiple programming languages. Most of the languages (like C++, Java or Python) are cross-platform. To sum up, it is almost possible to choose the language, the framework and the platform separately. Nevertheless, after a closer look on the implementations, it became clear that most framework ports on other languages are either poorly made or already abandoned. In addition, each framework included some mechanisms that were redundant when applied to the chosen task. Usually, agents were either automatically provisioned with individual threads or were executed by

some non-configurable custom scheduler. For some of the frameworks it was hardly possible to find any readable documentation, therefore their learning curve was far from being optimal. After comparison was made JADE Java framework was selected.

The main reasons to choose it were speed [8] (for there are no nondetachable visualization or control mechanisms involved), extensible documentation (mostly gathered on their official site [9], especially [10]), alive community and FIPA-compliance [11]. Other than that it does feature some tools for debugging and visualization of the created system that may simplify the future development. Of course there is a point where the system specification of the system goes beyond the JADE agent approach but using JADE will be plausible to create the test version of the framework, where the efficiency of the end product is of no concern.

3.6. Examples of the Framework Usage

The framework described above is meant for creating storylines for any possible game. That does not mean that its specification can not be extended to cover only the particular game genre or type. Let us use a RTS game such as Warcraft as an example. The vague specification of this game can be found in [12].

Example 3.3. Agents of the game can be divided into Commanders (such as the player), loyal units, loyal structures and neutral units and neutral structures.

The Commander's perceptions equals the combined perception of all loyal units and loyal structures under one's control. The external properties of a Commander feature resources (gold and wood).

The loyal units and loyal structures are given the owner upon creation and are able to execute owner's requests as well as start interaction with their owner in any particular moment. For example, citadel can execute a Commander's request on creating a builder unit. When it receives gold, it tries to pass it to the owner of the citadel. Moreover, there is a possibility for loyal units / loyal structures to communicate between each other. These make coordinated behaviour between loyal units and even loyal structures possible.

The neutral units have no owner, therefore must act on their own.

All agents (except Commanders) can interact with all other type of agents that have a different owner, yet within some constraints. These constraints can be changed by relationship modifier between owners, for example the "allies" relationship.

The objects in Warcraft can be divided in various subclasses, but each of them has its own position as its external property and is being contained by the map – the only environment in the game. There are agents that can contain some of the objects, but while doing so, they have no ability to create the simulation in which that objects would operate, so it will be simpler to depict the possession relationship as the external property of the owner.

The text description model of the game environment was given, but to create an interesting storyline it is essential to make the structure more complex, include dialogues or extensively describe the events. The first one is either made by creation of additional goals for units. The second require either collaboration with a human or an advanced AI technology. The third can be done by using GameObserverVoice class introduced in *Model0V* architecture. Having all the information from this chapter on the table it is not hard to modify the system to the point where it would efficiently implement the recently created entities. Embedding some code from the actual game into the simulation environment can prove itself to be a good way to test whatever was embedded and develop the gameplay in general.

3.7. Possible Usage

The initial goal is to create a system for storyline creation. But the resulting architecture can be used for different other purposes.

Testing Game AI. Since in *Model0V* architecture human can control the behaviour of all other classes by resolving messages and event handlers, *Model0V* based software provides the developer ability to test artificial intelligence algorithms in a simulated environment without actually implementing the environment and having full control over each entity in it.

Directing Movies. Since the *GameObserverVoice* class is capable of channeling the received information outside the program, it may be used to control entities in another environment (for example unity) that can capture the result to form a movie sequence or other kind of informational entity.

NOT Playing Games. *Model1* specification was designed to accept every kind of possible game with all the complexities of each possible entity. That is mostly due to the high level of abstraction and human based structure. On the other hand, such approach leads to an extremely low computational performance. While computer is not busy calculating physics of particles and rendering high resolution images such performance is plausible, but playing high-end games that are based on *Model1* specification from the previous paragraph would not be possible. However, in some cases, it might be useful to connect the developed framework to a game engine in order to obtain more interactive type of result and minimize the work of the developers by channeling data from the storyline creation system directly into the game.

CONCLUSIONS

This work captured an attempt to partially automate storyline designer role in game development by designing the corresponding framework. Different challenges that might occur during the framework implementation (such as resource distribution) were analyzed and for some of them possible solutions were proposed in *Model1* specification. The applicability of the created framework architecture was shown on the particular game.

Now, when the architecture is already specified it is possible to create an implementation of the framework, therefore might be done by the author during the next academic year.

REFERENCES

[1] Video game development. [Online]. Available: https://en.wikipedia.org/wiki/Video_game_development

[2] H. Tratteberg, A. I. Wang, and S. Gao, "A workflow for model driven game development," *IEEE 19th International Enterprise Distributed Object Computing Conference*, 2015.

[3] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. 2009 John Wiley and Sons Ltd, 2004.

[4] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1955.

[5] L. Sterling and K. Taveter, *The Art of Agent-Oriented Modeling*, 1st ed. The MIT Press, 2009.

[6] T. Rendon-Sallard and M. Sanchez-Marre, "A review on multi-agent platforms and environmental decision support systems simulation tools," [Available online]. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/87344/R06-4.pdf

[7] G. Weiss, *Multiagent systems - a modern approcach to distributed artificial intelligence*, 1st ed. Massachusetts institute of technology, 1999.

[8] P. Vrba, "Java-based agent platforms evaluation," 2003.

[9] Jade site. [Online]. Available: http://jade.tilab.com/

[10] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. John Wiley and sons ltd., 2007.

[11] R. Leszczyna, "Evaluation of agent platforms," 07 2004.

[12] Warcraft: Orcs humans. [Online]. Available: https://en.wikipedia.org/wiki/Warcraft:_Orcs_%26_Humans

[13] C. Badica, Z. Budimac, H.-D. Burkhard, and M. Ivanovic, "Software agents:

Languages, tools, platforms," *Computer Science and Information Systems*, no. 18, pp. 255–298, 2011. [Online]. Available: http://eudml.org/doc/252587

[14] K. Kravari and N. Bassiliades, "A survey of agent platforms," January 2015, [line]. [Online]. Available: http://jasss.soc.surrey.ac.uk/18/1/11.html

[15] F. Leon, M. Paprzycki, and M. Ganzha, "A review of agent platforms," November 2015.

[16] Comparison of agent-based modeling software. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software

Appendix A

The Comparative Table of Multi-Agent Frameworks, Platforms and Simulation Toolkits

The choice of properties for this table was highly influenced by [13]. The broad set of available agent-based frameworks was taken from [6, 11, 14, 15]. Also [8] and [16] were taken into account.

Agent-based frameworks				
Framework name	**License**	**Documentation**	**Programming language**	**Short description**
Agent factory	LGPL, +open source	docs, mailing list, forum	Java, AFAPL, AgentSpeak	The open source collection of tools. Features wide diversity of tools, languages and supported systems. There is a separate version for constrained devices.
AgentBuilder	Proprietary, commercial	+ consulting training, example, FAQ, docs, defect reporting, mailing list	KQML, Java, C,C++	KQML-compliant commercial agent platform.
AgentScape	BSD, open source	+ forum, email, docs	Java (plus use of XML)	Agent platform that is targeted to scalability and autonomy.
Aglobe	LGPL, open source	+- docs, mail contact	Java	Designed for testing experimental scenarios with agent's communication inaccessibility. Can be effectively used for real world simulations.
Anylogic	Commercial	++ extensive documentation and supporting tools	Java, UML-RT	Commercial simulation software with integrated graphics engine and visual coding.
Cormas	GPL, open source	+ forum, email, docs	SmallTalk	Smalltalk agent platform. Was designed to simulate systems that feature renewable resources. Now the developers are working on MIMOSA meta-modeling simulation platform.
Cougar	COSL, open source	+ FAQ, docs, forums, mailing lists	Java	FIPA-compliant. Cougar's cognitive architecture is optimized to resemble human thinking process.

Framework name	License	Documentation	Programming language	Short description
CybelePro	Commercial	? sales support	Java	Commercial high performance infrastructure for easy and fast development of large-scale, high performance agent-based systems.
EMERALD	LGPL, open source	+- documentation, mail contact	Java, JESS, RuleML, Prolog (plus use of XML, RDF)	New experimental framwork for integrating interoperable reasoning among agents in the Semantic Web, by using third-party trusted reasoning services built on top of JADE.
GAMA	GPL, open source	+ forum, email, docs, wiki, social media	GAML	Simulation platform mostly used to generate and visualize agents in a classic 3D environment. Makes use of its own programming language.
INGENIAS	CC By-SA GPLv2, open source	+- docs, mail contact	Java	A great example of DSM. Uses visual editors and its own languages to specify the multi-agent system and then interprets it to the actual programming code.
JACK	Commercial	++ extensive documentation and supporting tools	super-set of Java	Commercial ultra fast environment for building multi-agent systems using BDI execution model.
JADE	LGPLv2, open source	++ FAQ, mailing list, defect list, API, docs, textbook	Java, C#	Extremely popular, extensively documented, originally implemented in java. FIPA-compliant.
JADEX	LGPLv2, open source	+- docs, mail contact	Java	BDI agent system. Allows for programming agents to be created with either XML or Java.
JAMES	JAMESLIC, open source	+- docs, wiki, mail contact	Java	The set of efficient algorithms and tools. The simulation technique can be modified by the plug-ins with *plug'n simulate* architecture.
JAS	LGPL, open source	+ API, docs, tutorials, email authors	Java	Java Agent-Based Simulation library. Based on standard discrete event simulation paradigm.
Jason	LGPLv2, open source	++ manual, book, news/mail list, API	Java, AgentSpeak	A multi-agent development platform that uses an extended version of AgentSpeak to program individual agents.
JIAC	open-source	+ FAQ, docs, mailing lists	Java	Framework that can be used throughout all the development process.
MaDKit	GPL, open source	+ docs, forum, mail contact	Java, C/C++, Python	Framework that implements (Agent/Group/Role) organization model.Aslo uses concept of artificial societies (based on roles of agents).
MASON	Academic, open source	+- documentation, mail contact	Java	Simulation toolkit with visualization tools that can be detached from corresponding models.

Framework name	**License**	**Documentation**	**Programming language**	**Short description**
NetLogo	GPL	+ FAQ, docs, mail contact, tutorials, examples	NetLogo	This modeling environment can be used by specialists without prior knowledge of any programming language.
Repast	New BSD, open source	+- docs, mail contact	Java, C#, C++, Lisp, Prolog, Python	A widely used, cross-platform, agent-based modeling and simulation toolkit.
Sesam	LGPL, open source	+ Wiki, docs, documentation	Java	The main entities in a SeSAm model are agents, resources and the world.includes visual programming software.
Swarm	GPL, open source	+ Wiki, docs, mailing lists	objective C and Java	Reusable software tools created for artificial life applications.
EVE.	open source	+- docs on site (not full)	Java / javascript	New framework by Almende. Extremely flexible. Features the ability of creating agents with custom internal structure.