DIPLOMA THESIS

Actuators for an Artificial Life Simulation

Submitted at the
Faculty of Electrical Engineering, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Master of Science (Diplomingenieur)

under supervision of

O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Dipl.Ing. Tobias Deutsch

by

Benjamin Dönz
Matr.Nr. 9826140
Hauptstraße 37c
2372 Gießhübl

Date 03.11.2009 ______________________

Kurzfassung

Moderne Gebäudeautomatisierungssysteme müssen immer mehr Aufgaben bewältigen und werden dazu noch komplexer. Abstrakt gesehen müssen die Kontrolleinheiten dieser Systeme die Fähigkeit besitzen zielorientierte Problemstellungen in einem gegebenen Umfeld und unter Einhaltung von Sicherheitskriterien und anderen Randbedingungen zu lösen. Bisherige regelbasierte Algorithmen können mit der steigenden Komplexität moderner Systeme nicht mehr umgehen und die Suche nach neueren und „intelligenteren" Methoden für die Entscheidungsfindung ist ein weltweiter Forschungsschwerpunkt.

Das ARS Projekt (Artificial Recognition System), im Rahmen dessen diese Arbeit durchgeführt wurde, entwickelt eine Simulation um Modelle der Psychoanalyse und Neurologie mit Methoden der künstlichen Intelligenz zu kombinieren und die Fähigkeiten dieses Lösungsansatzes zu untersuchen. Im Zuge dieser Arbeit wurde die Schnittstelle zwischen der Entscheidungseinheit und der virtuellen Welt entwickelt, mit Hilfe derer die Agenten Aktionen ausführen können.

Der agile Entwicklungsprozess des Projektes und andere Randbedingungen erfordern vor allem einfache Erweiterbarkeit und robuste, testbare Komponenten. Um dies zu gewährleisten werden bewährte Patterns angewendet um eine allgemeine Struktur und Steuerung zu implementieren, die unabhängig von den konkreten Aktionen ist. Diese erlaubt es Befehle aufzurufen und bis zum Ausführungszyklus der Simulation zu sammeln, wo sie dann nach bestimmten Regeln wie Energiebedarf oder gegenseitigem Ausschluss geprüft, und schließlich ausgeführt werden. Die ersten etwa 20 Aktionen, die zum Beispiel Bewegen, Essen, Angreifen und vieles mehr ermöglichen, werden ebenfalls implementiert wodurch bereits verschiedene Test-Szenarien simuliert werden können. Um die Weiterverwendung und Nutzung der Aktionen zu vereinfachen, wird auch ein einheitliches Dokumentationsformat entwickelt das sämtliche Parameter und Eigenschaften der Befehle dokumentiert und als Katalog für die übrigen Entwickler zur Verfügung gestellt werden kann.

Abstract

Modern building automation systems have to deal with more and more increasingly complex tasks. From an abstract view, the control units for theses systems have to solve goal-oriented problems in a given environment and meet security constraints and other conditions. State of the art rule-based algorithms are already reaching their limits and new, more “intelligent” methods for decision making are in the focus of worldwide research.

The ARS project (Artificial Recognition System) is developing a concept which combines models from the field of psychoanalysis and neurology with methods of artificial intelligence. To evaluate the possibilities of this scheme, a new artificial life simulation is being developed. In the course of this work, an interface between the agent’s body and decision unit is designed and implemented to allow the agent to perform actions in this virtual world.

Due to the agile development process of the project and other constraints, the new components are required to be particularly extendable, robust, and testable. Approved patterns are chosen to ensure this, and a general infrastructure for calling and executing actions is designed. The respective classes allow calling commands and collecting them until the execution phase of the simulation. Here they are validated and checked for energy demand, mutual exclusions and other restrictions before they are actually dispatched. The first 20 actions are also implemented and allow the agent to move, eat, attack, and more. Using these commands, the first real test simulations can already be carried out putting the new components to actual use. Since other developers will use and enhance the software later, a documentation template is developed to create a uniform, compact, but also exhaustive catalogue of actions.

Acknowledgements

I would like to thank the ARS team and especially my supervisor Tobias Deutsch for their dedication and support during this diploma thesis and also Prof. Dietmar Dietrich for the opportunity to participate in this project. Additionally, I want to express my gratitude to my colleagues and partners from ASE who made it possible for me to continue my studies alongside my professional obligations. Special thanks also go to my family, friends and my girlfriend Gerlinde for all their love and support.

Table of contents

Abbreviations & Acronyms

AI	Artificial intelligence
API	Application Programming Interface
ARS	Artificial Recognition System
ARS-PA	Artificial Recognition System - Psychoanalysis
ARS-PC	Artificial Recognition System - Perception
AUML	Agent Unified Modelling Language
BDI	Belief-Desire-Intension
BOD	Behaviour-Oriented Design
BW	Bubble World
CAVE	CAVE Automatic Virtual Environment
IDE	Integrated Development Environment
GUI	Graphical User Interface
OAK	Object Application Kernel
TC	Test Case
UC	Use Case
UML	Unified Modelling Language
SVN	Subversion
WYSIWYG	What you see is what you get
XML	eXtensible Markup Language
XP	eXtreme Programming

1. Introduction

Large office buildings of today have control systems, which manage lights, heating, air condition, access control, security surveillance, and many other services. The number of sensors these systems have as input ranges to tens of thousands. Due to decreasing prices and increasing possibilities of the equipment, more and more applications are put under control of these computer systems and add to the already startling complexity. This increases the possibility of failures and also the dependency on the system, which makes these failures more fatal.

To cope with this, many approaches are studied by the industry and universities. To reduce the complexity at the system core, more intelligent sensors are developed; to enhance failure tolerance, distributed and redundant systems are studied; to reduce installation and maintenance costs, field busses are used. Another new strategy is examined at the department of Computational Engineering at the Technical University of Vienna: Bionics is the study of solutions and concepts found in nature and biology, and the application of these for technical systems. The most complex and also most successful concept for a control system is the human being itself. We can handle huge amounts of input data, we are fault tolerant and we can produce decisions in real-time. Therefore the Artificial Recognition System (ARS) uses the human being as a model system for the next generation of control systems [Zuck2008].

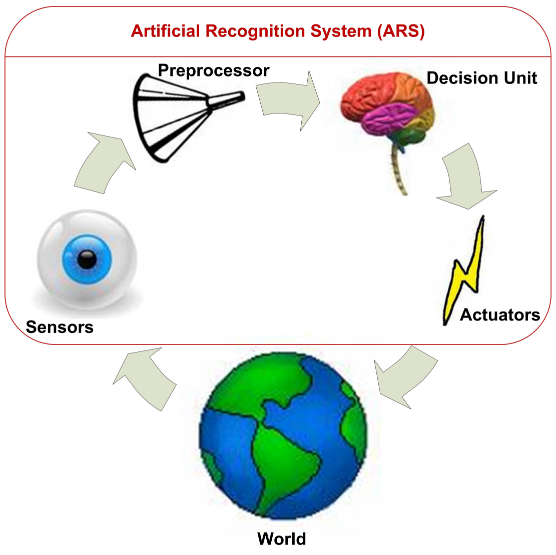


Figure 1: ARS Project and Decision Cycle [Deu2007]

As sketched in Figure 1, the project covers the whole cycle of decision making from the sensor input and pre-processing, to the decision process itself and finally to the controlling of the actuators. The first stages of the process, namely the pre-processing of the input, is studied in the perception sub-project (ARS-PC). This part is mainly responsible for breaking down the huge volume of information gathered from the sensors to an abstract, smaller, but more meaningful set of information. This data is then small enough in volume to actually allow to be processed by the decision unit: Here in a combined effort of neurologists, psychologists and engineers, a model for decision making is developed, mainly deduced from Sigmund Freud's second topic model. This sub-project is named ARS-psychoanalysis (ARS-PA) [Deu2007].

To evaluate the concept, explore the possibilities and compare it to other state-of-the-art practices, simulations must be done. These simulations are agent-based, which means that the simulation consists of a computer simulated environment and autonomous agents, which are software entities that can communicate and manipulate the environment without further control from outside the agent's program. These agents can gather information only through virtual sensors, use this to decide what to do next and then perform one or more of a fixed list of actions. The simulator for the project also has to allow to be configured for several different purposes, so different and increasingly more complex problems can be examined. In the beginning, the simulator will be set up to copy well studied problems from other artificial intelligence simulations like the "fungus-eater" of Masanao Toda [Tod1982]: Here a single agent lives on a planet with only two resources – uranium and nutritious fungus. The agent's goal is to mine as much uranium as possible, but moving around and mining consumes energy, which, in turn, can be filled up by finding and eating fungus. The decision unit of this fungus-eater agent receives sensor-information as input (it's energy level and a map of the world containing the locations of uranium and fungus) and has to produce a command (move, eat, mine) as output. The agent's performance will vary depending on the strategy it applies and therefore allow comparisons. Later simulations will add features like agent-to-agent interaction, dynamic environments, different classes of agents (predators vs. pray), manipulating objects and others. The actual requirements for the simulator were developed in a cooperation of engineers, psychologists and neurologists and are written down as short stories or descriptions of problem-situations very much like the description of the fungus-eater above. This way of implicitly writing down the requirements rather than explicitly listing them is in accordance with agile development methods and is a more natural way of expressing them, especially when parties from such different fields of expertise are involved.

First evaluation of the concept was done with simulations made with a professional simulation tool called Any-Logic. This allowed creating a prototype quickly and producing requirements for a more sophisticated simulation-tool. The latter is now being developed using a simulation toolkit called MASON, which is a framework and tools-library for developing any kind of custom agent-based simulation. It is open-source, programmed in Java and comes with optional additions like a physics engine, which was used to create a more realistic environment. But MASON itself only provides the skeleton for the simulation; the actual environment, agents, sensors, actions and everything else must be programmed individually. This allows producing very powerful and complex simulators, but it also takes time before first results become available. The software is developed by a core team of doctoral candidates, which created the basic components and manage the development process, but also by several students who participate in the course of internships or diploma thesis like this one. To allow these fluctuations in the personnel of the team a feature driven development process is applied: A list of individual features is derived from the requirements, which can then be implemented independently piece-by-piece. This keeps the amount of communication and interdependence at a minimum and can also cope with changes and additions to the requirements, which come up after first experiences have been made.

The new simulator consists of five main components:

- The simulation-framework: This is mostly provided by the MASON libraries, but configured for the project. In addition several utility-functions and the main simulation cycle were created so other parts can be fitted in.
- The simulation-environment: It is two dimensional, bounded and can hosts the entities, which can be anything from stones, plants that can be grown and eaten, and of course agents. The environment is governed by a physics engine, which ensures realistic physical interaction of the participants like surface friction, collisions, forces joining objects, etc.
- The agent's environment-body interface: It contains attributes like size and weight for the physics engine, but also sensors for gathering information from the environment (vision, collision, audio, etc.) and actuators for performing actions in the environment such as moving, attacking, or communicating.
- The agent's body: It holds energy and inventory states, manages the communication and has several organs for different purposes, for example a digestion system which converts nutrition to energy and excrements, a health system which stores state and effects of damages by collisions or attacks.
- The agent's mind: It takes sensor information as an input and delivers actions as an output. An interface allows swapping this decision making unit and study different implementations. This component is the actually studied object and the theoretical models developed in the project are implemented here and tested in the simulation to determine their properties and performance in various situations.

In the course of this thesis the actuators of the agent's environment-body interface are designed and implemented:

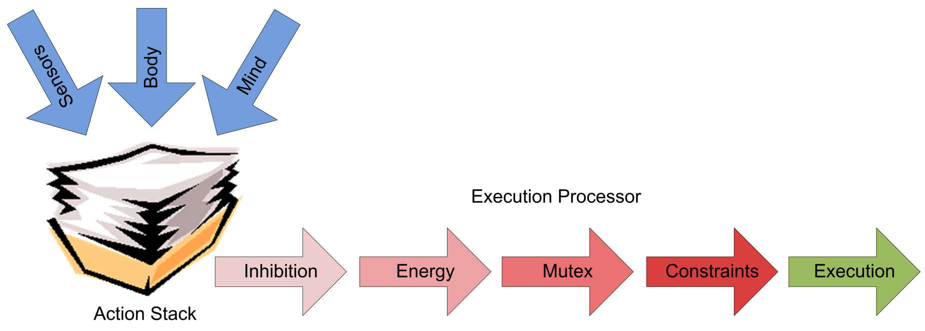


Figure 2 Execution Stack and Processor

Actions are launched from several sources and collected in a stack as can be seen in Figure 2:

- Sensors: As a direct reaction to a sensory input actions can be launched, comparable to reflexes.
- Body: Internal units of the body can produce actions such as a change of the agent's appearance like it's body-color or eye shape as a response to illness or fatigue.
- Mind: The main source for actions is the decision-making unit whose sole purpose is to produce actions to reach the agent's goal.

These actions are collected in a stack until the execution stage of the simulation cycle where they are processed by the execution processor:

- Inhibition: The mind may anticipate an automatic sensory reaction and needs the possibility to inhibit it. These actions are discarded from the stack without further influence.
- Energy: All remaining actions are considered for execution, but depending on the type of the action a certain amount of energy and stamina is consumed as long as there is enough.
- Mutual exclusions (Mutex): Actions can exclude themselves like moving left and right at the same time or picking-up and dropping something simultaneously. If combinations like these occur then these actions will also be discarded even though the energy was consumed.
- Constraints: Finally each action may have certain individual constraints. For example, the agent can only eat if something eatable is in range. Each action's constraints are checked and if they cannot be fulfilled the action will not be carried out
- Execution: All remaining actions will then be executed

The action processor also stores a history of actions from the previous round including information about if whether the action was performed or not. Since the software will probably be further enhanced and adapted, care was taken to use patterns which ensure the most flexibility. The action processor is encapsulated and only deals with a common "action" interface, so more actions can be added, and each can be changed independently without compromising the system. Actions that manipulate other entities do so through proxy-interfaces, so entities and actions can be developed independently and entities can be enhanced to allow an action when needed. An example for this is the "eat" action, which calls a generic function of a proxy-interface "eatable". When the command is dispatched, the processor searches for entities containing this interface in range, and when one is found, it calls the appropriate methods on the interface. The eaten entity reacts to this action in a way defined by the entity itself, for example return the food-equivalent of the bite and shrink in size or disappear from the environment. This follows the decoupling and separation-of-concerns rules, and allows entities to be available for the "eat"-action just by adding the interface and implementing the reaction.

As mentioned before, a list of requirements in the form of short stories or problem situations was created by the core team. From this, the implicit actions which need to be supported were derived and more than 20 individual actions were listed, e.g. move, turn, eat, kill, communicate, attack, change body colour, sleep. These actions can be grouped into three categories:

- "Stand-alone": These actions change internal states of the agent or it's state in the environment such as moving or changing the body-colour. They do not involve other entities.
- Binding actions: These actions bind entities to the agent, e.g. pick-up, move from/to the inventory. These actions need entities in range, which can be identified because they inherit a certain interface.
- Manipulating actions: Specialised actions, which involve certain interaction with other entities like attacking or communicating. Each of these actions interacts with the other entity only through a certain interface, for example, the "eatable" interface mentioned above. This allows identifying entities, which support a certain action, and also allows decoupling the function from the entity.

Following the feature driven development process, the features – here the actions – are listed, discussed in detail with the leading software developers and then implemented one-by-one allowing immediate usage after testing. The actions are also documented using a common description template that contains the purpose of the action, it's parameters and information on how to implement it. This results in a catalogue of actions, which can be found in Appendix II of this thesis and offers structured and exhaustive information for the developers who need to use them later.

2. Background

First the subject – artificial life simulations – is be presented by explaining the basic idea of such systems and the state-of-the-art in this field. The following section will then introduce the Artificial Recognition System (ARS) project, which this work is a part of, and lay out the concept of the simulator. This chapter then closes by summarizing the methods and tools used in the practical part of this work.

2.1 Artificial Life Simulations

After giving an introductionary overview, the state-of-the-art regarding artificial life simulations is described from different viewpoints such as the decision making architectures, the development process and the simulators.

2.1.1 Overview

The term "artificial life" was introduced by C. Langton in 1986 and meant the investigation of life as it could be [Bed2001]. This field of study is highly interdisciplinary and covers a whole range of areas from biology and the emergence of life as examined by [Coh2007], evolution itself as studied by [Cur2003] and even the study of social behaviour and interaction [Dau1999]. In the course of the Seventh International Conference on Artificial Life in 2001, a list of open questions in the field of Artificial Life was created. This list contains a total of 14 challenges that cover all these fields of expertise [Bed2001] and also suggests using simulations as the instrument for solving some of the problems. The term "artificial life simulator" is therefore used for several different applications. The shared aspect of all variations is, that they target at synthetically constructing a phenomena from basic units rather than analysing complex phenomena and breaking them down into their parts [Sip1995].

In this thesis an artificial life simulation is understood to consist of an environment and one or more agents. Agents are independent entities which can extract information from the environment with sensors and can perform certain actions to manipulate or explore the environment such as moving, or eating. The aim of the simulation is to study different decision-making strategies and the results of these schemes in game-like problem situations specially designed to focus on certain points of interest. Referring to the list of challenges mentioned above, the related questions from there are [Bed2001]:

- Explain how rules and symbols are generated from physical dynamics in living systems.
- Determine minimal conditions for evolutionary transitions from specific to generic response systems.
- Develop a theory of information processing, information flow, and information generation for evolving systems.
- Demonstrate the emergence of intelligence and mind in an artificial living system.

Simulations in general, are a way to model more complex systems in order to explore specific problems without having to deal with non-relevant details. It is therefore a simplified model for a real system and only valid within the scope it was designed for. A very common argument concerning simulations is therefore, that not all insights gained from them can automatically be declared valid just because some real world test cases worked [Pol2008]. In this project, the aim is to study the decision-making process itself and the principles involved. The part relevant for valid results is therefore not the simulation environment but the given problem scenario and this is of course valid in any domain (including any simulation which can represent the scenario).

The practical part and main focus of this thesis is the development of the possible actions an agent can perform in such a simulation. When it comes to computer systems a choice is made by applying some sort of algorithm to produce possible alternatives and then score them using all information available including a set of goals [Pom1996]. In artificial life simulations all information available to the agent is gathered via its sensors at the decision moment or earlier. The possible alternatives are the fixed list of actions the agent can perform and the decision is the choice which action should be executed at a given point in time. The number and variety of actions enlarges the number of possible decisions and allows for more complex scenarios. Therefore, creating many fine grained actions instead of only a few more powerful ones is important, e.g. actions for move, turn, eat instead of one “find-food-and-eat-it” action.

2.1.2 Decision Making Architectures

The first successful architecture described in detail for machine decision making which included a certain degree of reasoning rather than simple rule-based algorithms is the subsumption architecture developed by Rodney Brooks [Bro1986].

Rather than splitting the decision process into a set of serial tasks, this architecture builds control-layers stacked over one another. As sketched in Figure 3, each layer is realised as a single finite-state machine and adds further complexity – here called competence – to the model by manipulating the data flow of the layer below. In his paper, the zero-level competence for a mobile robot is just motion directed at moving it away from other objects. The first-level then adds a heuristic for planning ahead so that less evasive manoeuvres need to be carried out. The second-level adds a basic heading to the aimless movement by selecting potential targets and influencing the preferred direction towards this target.

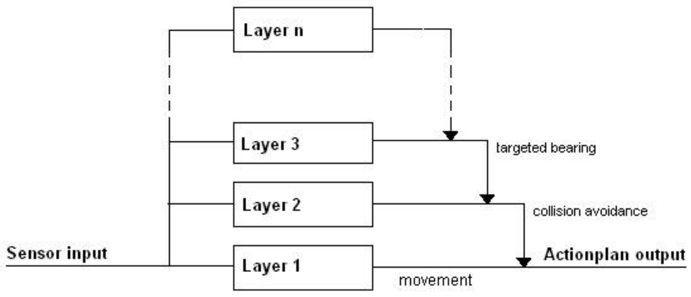


Figure 3: Subsumption Architecture [Bro1986]

This idea is elegant in an engineering sense, because performance can be increased just by adding well-encapsulated black-box units and stacking them without the need to change the lower layers. But in practice this approach has several flaws [Moc1995]:

- The construction of finite state machines requires great efforts for realising specific tasks.
- The system is not adaptable and has no learning components since the connections are hard-wired.
- The subsumed information is lost since behavioural arbitration is based on priority among the modules.

To overcome these difficulties several adaptations have been proposed such as neural networks for learning or emotional mechanisms.

Another well-studied and heavily applied concept is the belief-desire-intension (BDI) framework described in detail by [Rao1995]: One data-structure comprises all collected sensor-information and combines it to the agent's view of the world – the beliefs. A second data-structure represents the objectives and their priorities – the desires. And finally the third data-structure – the intensions – is the currently chosen course of action. Therefore the agent chooses an intended action that will satisfy it's desires – given the current beliefs.

The BDI framework was later enhanced with emotions. As shown in Figure 4, emotions come into play twice during the cycle belief->desire->intension: Primary emotions are generated immediately after updating the beliefs as a response to the changes in the view of the world. These emotions could be fear as a response to threats, sadness as response to the death of another agent, etc. These primary emotions are then used to adapt the intensions, which in turn generate secondary emotions. The process of adapting the intensions is carried out in a loop for a number of cycles and can produce changing emotions. For example, anger from sadness, which triggers other intensions, which again produce emotions and so forth. The performance of this decision making process was tested and seen to outperform existing BDI architectures by [Jia2007].

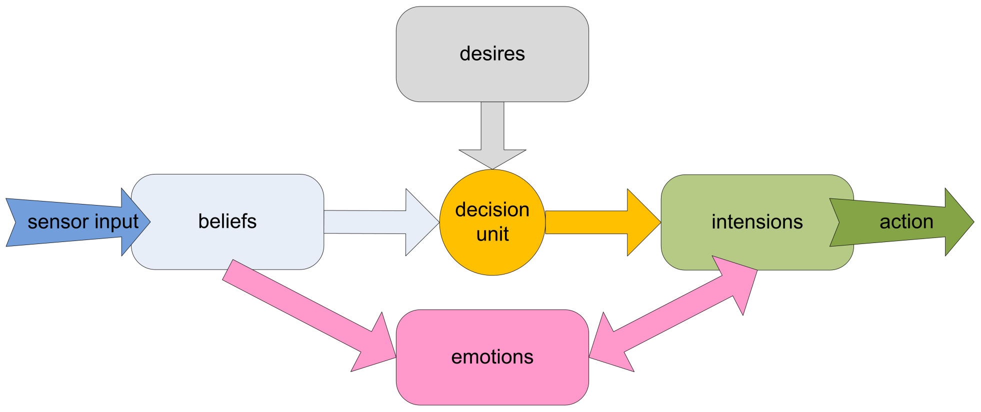


Figure 4: BDI Architecture [Jia2007]

Another adaptation to the concept has been made by [Ho2003], [Ho2004] through adding autobiographic memory to the agent. Both perceptions and actions are stored together as episodes in a way which allows to recapitulate past events both for adapting the agent's own behaviour as well as to communicate these episodes to other agents for their profit. To further explore this concept they developed a 3D simulation environment where agents can only survive by keeping separate internal values – energy, moisture and glucose – above a minimum threshold by searching for and consuming certain resources while every action they perform reduces these internal values. Through simulation, [Ho2008] found that the autobiographic memory and the possibility to communicate experiences could be used to ground the agent's decisions, i.e. the reasons for an agents actions can be traced back to significant experiences.

2.1.3 Development Processes

The BDI architecture is the basic idea behind most of the agent architectures and has even led to the development of domain-specific development processes.

From the engineering point of view, processes are needed to guide the development, allow for productive collaborative work and apply best practices. In the field of multi-agent programming the issue of autonomous software components must also to be taken into account and therefore new processes have been developed such as Tropos, Gaia, Prometheus or NUMAP which even comes with proprietary support tools to assist in the process [Hec2008].

A totally different approach to the design-process is the Behaviour-Oriented Design (BOD) proposed by [Bry2003], which focuses less on the programming task but more on the decomposition of the actual problem so more sophisticated agents can be developed without losing oversight. The method is similar to object-oriented design and works downwards from the basic "what should the agent do" description of a scenario. This behaviour is then split into atomic actions and sequences and sorted into a hierarchy with the root being the goal, and all possible actions and their constraints that can lead to this goal. The resulting tree can then be developed in separate parts, each in iterative cycles enhancing the specifications in the process. [Bry2006] have also developed a modelling suite using MASON as simulation.

2.1.4 Simulator Environments and Frameworks

A very old but still referenced environment for testing and comparing the performance of decision architectures is “Tileworld” developed back in 1990. The environment consists only of plains, holes and obstacles, and is generated randomly. Holes disappear after a certain timeout and reappear at a different location. The goal for the agent is to collect points by filling holes before they disappear. The simulation has only two phases: The act-phase in which the intended actions are executed and the reasoning-phase which contains the implementation of the decision which should be tested. As input information a whole world map is available including the timeouts and locations of all holes. The only actions available are moving and filling holes [Pol1990]. This test-architecture, although of course much simpler, shares the basic idea of the project at hand: a dynamic environment, a test scenario and agents following an input-reason-output cycle where the reasoning-part can be swapped to test different strategies.

Large numbers of models have been developed, which use genetic algorithms to evolve creatures and test the fitness by using an artificial life simulation to see how good the creature performs at a given task. Influential examples in this field are Tierra, Echo and Polyworld [Bor2006]. Later these systems became the basis for a set of more complex simulations, which added physics to the simulations and created creatures from basic parts, which were described in genetic code. An example for one of these more complex simulations is “framesticks”, which was developed by Poznan University of Technology in Poland and was introduced in the year 2000 [Kom2000, Kom2003]. This framework allowed creating stick-like “creatures” with neural networks as brains, which could evolve both in their bodily appearance and their mental abilities. The main focus here was still to study the power of the evolutionary process itself, but since the fitness of a creature was determined by it’s ability to reproduce they were also able to move and interact in a complex environment which featured different terrains, obstacles, lakes, etc.

A number of generic simulation frameworks has been developed which can potentially be used to create more complex simulations, e.g. SWARM [Ter1998], MUTANT [Cal1998], RePast [Tat2006], MASON [Luk2005]. In a comparison of popular multi-agent toolkits MASON [Rai2006], which is used for the practical part of this thesis, was recommended for calculation intensive simulations with many or complex agents, since it is faster than other frameworks and can be distributed over several machines. More detailed information on MASON and comparisons with other frameworks will be given later.

Another interesting idea is to use commercially available computer games, which already come with an environment and a framework to implement computer opponents. Some of these games like BioWare’s Neverwinter Nights even come with designers to develop scenarios. [Spr2004] uses adaptive scripting to generate more advanced computer opponents, but the basic idea could also easily be enhanced to produce an environment for artificial life simulations. A way to do this would be to add an abstraction layer, which provides the agents with restricted sensor information of their surroundings and passes on the decisions as actions back to the simulator. This way both the environment and all the already available out-of-the-box actions and features of the game can be used. Of course one would also be restricted to this fixed list of actions, but if the interface is designed properly, wrappers for several games could be developed while reusing the agent's decision unit and allow more possibilities this way.

2.1.5 Actuators and Actions in Existing Simulator Environments

Since artificial life simulations are developed to study and examine behavioural or evolutional theories, most publications focus on these topics and give little or no reference to the actuators and possible actions of the agents. The following enumeration will list the actions available in each of the simulation environments presented in the previous chapter as far as they have been published:

- Titleworld: The only available actions for the agent are moving from one square to an adjacent one and filling holes [Pol1990].
- Echo: Agents can move between different locations, eat and store resources but can also interact with other agents by fighting, trading resources and mating [Bor2006].
- Framesticks: The number of actions is very limited but the bodily appearance is taken into account and the speed, performance and abilities depended on the physical shape of the creature. The agents can move through different terrains (walk, swim, fly), consume energy and fight.
- Commercial games: When using commercial, off-the shelf games like [Spr2004], the available actions depend on which game is used. In the concrete case of Neverwiter Nights, which is a role-playing game, actions such as movement, picking up and dropping objects and fighting are available. Depending on the figure additional actions, like casting spells to attack opponents may also be available.

However all the system mentioned do not support the complexity needed to study decision-making processes. For this task a more complex environment and a larger number of available actions are needed to scale up the decision-space.

2.1.6 Related Work

Even though the technical possibilities have increased immensely, not many other projects can be found that can be compared to the ARS project this thesis is a part of.

A group at the University of Arizona are currently working on an enhanced BDI based simulation-framework with human-in-the-loop capabilities. The simulator in this case is a CAVE Automatic Virtual Environment (CAVE) where the environment is calculated from Google 3D images using Google SketchUp 3D. Using the CAVE technology a human can interact with the simulation in a cave-like room where its current view is presented as images on the walls. The goal of this project is Crowd Simulation where each human in the crowed is represented as an autonomous agent with a BDI model for decision making [Lee2008]. This is of course a totally different goal than the one of the project at hand, but the idea of human-in-the-loop simulations with real human-agent interaction is an idea, which could also be applied in artificial life simulations – both to add additional dynamics to the simulation as well as to compare the behaviour of agents relative to humans within the same environment. The latter could be done by comparing logs of the actions which would be analysed by behavioural experts [Nor2000].

The team working on autobiographic memory mentioned above have created and application called “FearNot!”, which tells stories by letting them emerge from the interactions of agents rather than fixed scripts. The story is therefore not implemented in the form of a script, but as intentions the single agents have. By communicating and trying to reach their individual goals the story unfolds and can be followed by human observers. The agents are also capable of summarizing the past events via their autobiographic memory and can – in contrast to a fixed script – react dynamically to changes in the environment or suggestions on what to do next, which can be given by observers at certain points in the story. This application is currently being evaluating at primary schools with children aged 9 to 11 who are supposed to judge the performance of the actors both with autobiographic memory and without it [Ho2007].

Jean-Daniel Kant and Samuel Thiriot from the Univeristé de Paris have proposed an alternative to the BDI decision-making model: The Cognitive Decision AGEnt (CODAGE) [Kan2006]. The main idea here is to distinguish between relevant and irrelevant information coming in from the sensor-layer by using experiences from the past to determine the priorities. The concept is a descriptive approach where decision-making is said to be a process comprised of three steps: intelligence, design and choice. Intelligence is understood here as the process of understanding what the decision is about by exploring the current context in respect to the constraints of the decision. In the following design phase possible different solutions to the problem are created. And in the choice-phase the system finally chooses one of these. This basic scheme has been applied by setting up a tree of alternatives, where the nodes represent different possible states of the world in past, present and future. The arcs between the nodes represent possible transitions of these states. This tree is then modified and enhanced by different agents:

- Agents which add information from perception and experience
- Agents that rate information with priorities or certainty
- Agents that connect information with reference-points

The final result is a tree, which shows different possible future states, how desirable they are, and how to get there. Another agent – the decision agent – works in parallel to select possibilities, which should be explored further. This decision-making architecture is very sophisticated and makes use of agents at a very low level of the process. It is also obviously very computationally intensive and therefore more useful for studying a single decision rather than a real-time decision process needed for an artificial life simulation. Currently the team is developing a simulation, which they want to construct as a generic framework where different decision heuristics could be incorporated including learning neural network decision units and also human-in-the-loop units so agents and operators can compete against each other in the simulation.

2.2 Artificial Recognition System – Project (ARS)

In modern office buildings more and more sophisticated automation systems are implemented. About 30 years ago the only few sensors in a building were usually thermometers for the heating and air-condition. Now building automation systems manage huge numbers of sensors and actuators for control, monitoring, surveillance and security purposes. The control-systems, which have to manage this ever-increasing data-flood, are becoming more complex as well. This leads to failures and expensive maintenance and installation work. But still there is no end in sight and even more applications and domains are put under the control of these systems, which also leads to a high dependency, making a possible failure have an increasingly large impact [Pra2005].

The software components developed in the course of this thesis are used for the "Bubble World" (BW) simulation, the artificial life simulation for the Artificial Recognition System (ARS) project. The primary goal of this project is to develop a more human like decision-unit which can cope with large amounts of input-data, filter out the relevant information, compare redundant sensor-information and use this for more fault-tolerant and robust applications in the fields of surveillance and building-automation [Pra2005].

The operation cycle for a decision in this system is perception, decision, action. The project is thus divided into two main parts: ARS-PC (Perception) where the systems responsible for gathering and interpreting sensor-inputs are developed and secondly ARS-PA (Psychoanalysis) where the decision making process is modelled [Deu2007].

Perception

As mentioned above, the decision unit has to be able to cope with hundreds, thousands or even millions of sensor-inputs and use this information in real-time for determining the course of action. Therefore this flood of data must be broken down to a volume that can be processed by a computer system. To achieve this, the concept of Neuro-Symbolic Networks was developed by [Vel2008] in the course of the ARS-PC project:

Neural networks are fault-tolerant, can deal with incomplete information and learn, but they cannot explain the decision process nor the output information. Symbolic systems on the other hand are very powerful when it comes to knowledge representation and can be explicitly combined by logic relations. Neuro-Symbolic Networks make use of the strengths of both approaches by creating symbols from layers of neural networks. The incoming sensor-information is condensed to more and more abstract symbols in each layer. These symbols therefore become more compact but also represent more meaningful information. In this way sensor-information from tactile sensors, light-barriers, cameras, motion sensors and others can be combined to symbols for "person entering the room", "person walking", "person standing" and "person leaving". This information is much more valuable than the large set of sensory information and also much more reliable than a state-of-the-art single motion sensor for surveillance would be. In later layers these basic symbols can be further combined to even more complex ones like whole sequences.

Due to the neural networks at the base of the system, operations can learn new symbols as they go along, even though the most basic symbols at the lowest layers have to be set in advance as a sort of reference from which all other information is derived. Another advantage is that the process can – in principle – be carried out in parallel, even though current computer technology has to simulate neural networks due to lack of specialized hardware. But the whole process can still be integrated in a hierarchical network of computers working in parallel at certain stages in the process.

The final result – the most abstract symbols – is scales smaller in volume than the basic inputs at the beginning of the process. They can therefore be handled by a decision unit, which they are passed on to, as the next step.

Decisions

The model developed in the ARS-PA project is highly interdisciplinary using knowledge from the fields of neurology, psychology, artificial intelligence and computational engineering. The development was carried out in a top-down design approach, starting off with Sigmund Freud's second topic model at the top and aiming at creating a coherent implementation, which can be simulated by computers. Compared to state-of-the-art architectures mentioned above, the approach taken in this project is a far more complex. But besides being one of the most thoroughly studied concepts in the field of psychology, this model can be split it in into individual components, namely the ID, Ego and Super-Ego. Each of which have certain properties, functions and interrelations, which can again be split into parts. This model is therefore accessible to a top-down design approach and can be tackled with common engineering practices [Zuc2008].

Simulation platform

The first simulation developed for verifying the model was created using a professional simulation tool, AnyLogic. This tool suited the requirements and was chosen primarily because it offers quick results with first prototypes and allows the usage of native Java, so code can later be ported to other platforms if need be.

The number and complexity of the test-scenarios needed for the evaluation of the model increased during further research work, and AnyLogic, offering many charts and extensive visualization of the simulation, suffers from performance loss when increasing the number of entities and the complexity. A second issue is the demand for automated tests and comparisons which need special logging are not offered by Anylogic [Deu2008]. Due to these problems the current project was launched, aiming at developing a new simulator to overcome these issues.

Simulation architecture

Figure 5 shows a screenshot of the actual simulator: the new simulation set-up has a 2 dimensional closed environment with fixed borders that cannot be overcome by entities of the simulation. Depending on the scenario, the environment can be equipped with objects such as the two stones, the cake or the can (blue dot) in the example. In a single simulation, several agents – represented by the three large coloured circles – can be added which can then move in the environment, manipulate objects and interact.

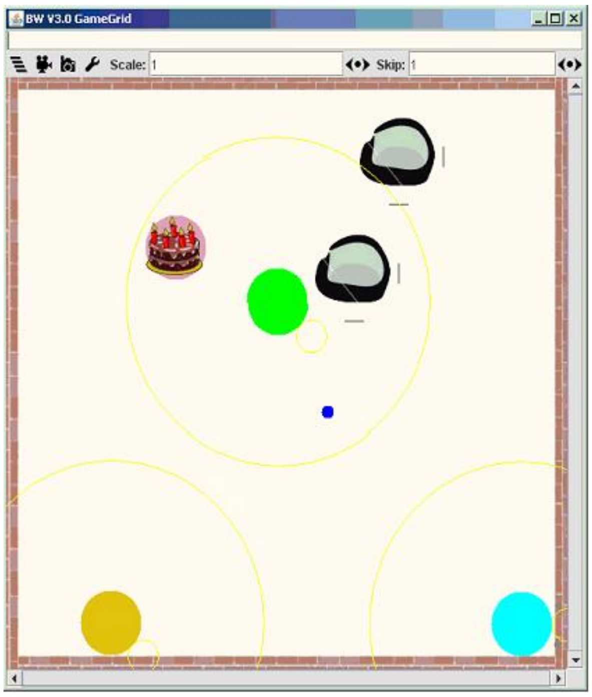


Figure 5: Screenshot of Simulator Visualization

Every agent is comprised of a mind and a body, which can differ in respect to the implementation, i.e. they have different physical abilities or minds. The mind is comprised of the decision unit and all components needed according to the model such as memory, and is only indirectly connected to the environment via the body. From this it gets input information in form of sensor-data and can execute actions by passing back appropriate commands. The body serves as a container for the internal state (health, energy or stamina) and also has several sensors. Internal sensors, representing the bodily states; and external sensors which represent information gathered from the environment, e.g. vision sensor, collision sensor, audio sensor. The body itself also has actuators with which it can manipulate both its internal states, for example setting a sleeping state, and also the environment, e.g. move, eat, attack.

The simulation is carried out in rounds, each comprised of a 5-stage cycle as can be seen in Figure 6:

1. Sensing: In this phase the sensor-data is updated by gathering information from the relevant sources.
2. Update internal body state: Status values like stamina or health are updated.
3. Process data: The decision unit receives the updated sensor-data as input and produces commands as output

4. Execution: Commands have been issued during all previous stages – reflex-like actions directly by the senses, representational actions like a change of body colour due to the internal body status, and of course the actual commands produced by the decision unit.
5. Advance Cycle: The actual simulation is carried out and the simulator-states are updated.

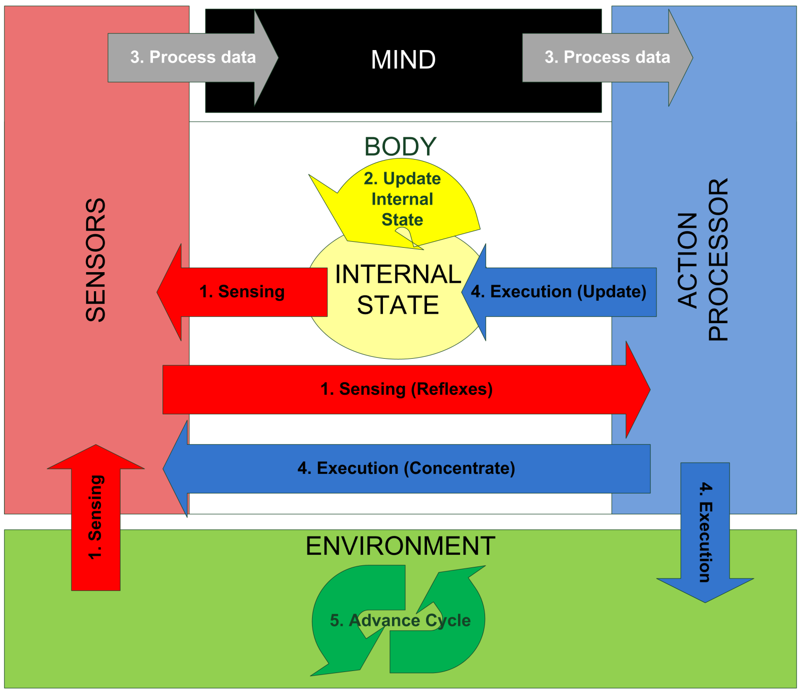


Figure 6: Simulation Architecture and Cycle

The actual object of study is the 3rd stage of the cycle. Taking only the sensor data as input, commands have to be produced as an output by the decision unit. These decisions are recorded and logged for later analysis. The interface for the decision unit is as universal as possible so different models can be implemented and compared without having to change other parts of the software. In addition, configurations can be applied so agents with different abilities can be constructed such as restricting the usage of certain actions or changing performance parameters.

Action execution

The practical part of this thesis is implementing the 4th stage of the simulation cycle. A means to collect the actions during the previous stages of the cycle has to be offered, and the actual execution of a list of about 20 actions has to be implemented. Due to the modularity of the concept this part of the simulation can be implemented without interfering with the rest of the project. A detailed analysis of the requirements will be given in Chapter 3.

2.3 Development Environment

This section introduces the methods and tools used in the course of the practical part of this thesis. Since the programming language used is Java, lots of Open Source tools exist and the first part will list the ones which were chosen by the project team for this simulator. The main non-generic library used for the simulation is the MASON multi-agent simulation toolkit which will also be presented here. Afterwards the basic development process will be explained and also some other possibilities will be shown for comparison. The last subsection will focus on patterns – both the basic idea developed by Gamma et. al, as well as some popular examples will be given.

2.3.1 Development Tools

The development team for the new simulator mainly makes use of open-source tools. The ones relevant for this project will be introduced briefly in this chapter.

Programming language

Java was chosen as the programming language and is used here in version 1.6. Java is an object-oriented language and similar to C in it's syntax. It is not compiled to machine code, which can be executed directly, but to an intermediate code which is interpreted by a just-in-time compiler. This makes it highly portable which is one of the reasons why it was so ideally suited for web-applications where the client can have any operating system or browser. Besides this java is also a fully qualified programming language and has a large community of developers, which contributed many open-source tools and libraries. This makes it uniquely attractive for non-commercial applications such as this one.

Integrated development environment (IDE)

The Eclipse Project was founded by IBM in 2001 and supported by industry leaders such as Borland, MERANT and others. In 2004 it was transformed into the non-profit corporation called the "Eclipse Foundation". Since then it is funded by members and has it's own managing staff, but the actual work is carried out by open source developers that volunteer their time to the project [Eclipse2009]. The main project of this corporation was and still is the Eclipse IDE (Integrated development environment), which is used in this project.

The Eclipse IDE is actually a multi-purpose platform for running plug-ins. These plug-ins provide the actual functionality. They can be easily updated and enhanced by an integrated update and installation wizard, which connects to databases with available tools via the Internet. The development IDE can be downloaded in pre-configured packages for several programming languages. Here the "Eclipse IDE for Java Developers" is used, which already contains all relevant tools: package browser, editor with syntax-highlighting, builder, debugger and also the JUnit test-suite for creating and running unit-tests.

Version-control

The project is carried out by a core-team and additional developers, which contribute components in the course of projects or diploma thesis such as this one. To allow the participants to work independently, but share their work within the team, a version-control system is used. For this project Subversion was chosen.

Subversion is a licence-free tool developed by CollabNet. It is available for several operating systems including Windows and Linux and already comes with a web-based access-scheme, so it can be used to allow connections even over the Internet. The system offers version control on file and directory basis with additional possibilities like branching and tagging for advanced repository management [CollabNet2009].

The client by CollabNet is only available as command-line tool, but plug-ins for the most common IDEs are also available. This includes the Eclipse IDE, so the Subversive SVN Team Connectors are used in this project for connecting to the archive.

Team forum and requirement repository

Subversion is used to keep track of the project source-code, all other information like requirements, how-tos, references, etc. are stored on a web-platform, created using DokuWiki.

DokuWiki is an Open-Source Wiki system developed by the German Andreas Gohr. The system was programmed in php and comes as a set of libraries, which offer built-in version control, WYSIWYG editing, indexing, and linking of content [DokuWiki2009].

2.3.2 MASON

According to [MASON2009], MASON stands for "Multi-Agent Simulator Of Neighbourhoods… or Networks... or something...". It is developed by the George Mason University in collaboration with the GMU Centre for Social Complexity and is a simulation toolkit for computationally demanding multi-agent systems. It offers a discrete event simulation core, visualization libraries for 2D and 3D simulations and other additional extensions such as the 2D Physics Simulator Package. It is written in Java and is the framework at the base of the simulation, which is currently being built.

Architecture

According to [Luk2005] the main goals for the design were:

- A small, fast, modifiable core with separate visualization in 2D and 3D
- Check-pointing any model to disk such that it can be resumed on any platform with or without visualization
- Efficient support for up to a million agents without visualization and as many as possible with visualization
- Easy embedding into existing libraries

One of the goals, which were explicitly not made for MASON, was to include parallelization across multiple processors, it is – by design – a single process simulator.

The main architecture consists of three layers:

- The utility layer: Contains general-purpose classes such as a random-number generator, GUI-widgets, etc.
- The model layer: Main classes of the simulator, responsible for scheduling and holding object states and associations.
- The visualization layer: GUI-based visualization and manipulation of the model. Each object of the model is referenced through a proxy for manipulation or inspection. The model and visualization layers were carefully separated so the simulation can be run with and without visualization or be stopped and continued with a different visualization model.

The framework is designed more like a library to allow excessive adaptations and use in a large variety of applications by adding additional layers on top of the model layer, such as extensions directly availably for MASON or custom-designed additions for domain specific problems [Bal2003].

The actual simulation is a scheduled execution cycle, where entities (agents) are registered at the scheduler for execution at a certain point of time in the future via a common interface. This simple behaviour can then be enhanced by using wrappers for ordering or grouping entities or running them multiple times or even in parallel threads [Luk2005].

Comparison

Table 1 shows an overview of two papers comparing different simulation frameworks. The left column shows results from the developers of MASON themselves [Luk2005], while the right column displays the conclusions of an independent group [Rai2006] concerning the most popular systems.

Table 1: Simulation Framework Comparison

Name	**Conclusion by [Luk2005]**	**Conclusion by [Rai2006]**
Objective-C Swarm	(+) Graphical visualization, inspection of simulation objects, stochastic event ordering (-) Objective-C is an uncommon programming language and it's key features and benefits are not used for the framework	(+) Stable, allows separation of graphical interface and model, allows hierarchies of swarms (-) lack of development tools, weak error handling, no garbage collection, bad documentation
Java Swarm	Special libraries which communicate with the objective-C version	(+) good trade-off between Objective-C version and Java enhancements (-) clumsy work-arounds for porting features from Objective-C to Java, difficult to debug due to Objective-C libraries used, slow

Repast	Re-Implementation of Swarm in Java or .NET; Large community, lots of functinality – neural networks, genetic algorithms, social modelling system, system dynamics modelling, logging, GIS and graphs and charts. (+) Graphical visualization, inspection of simulation objects, stochastic event ordering; Additionally from the same authors in [Bal2003]: (+) Import/Export Excel, charts, SimBuilder	(+) fast, implements most of Swarm's functions as well as good enhancements, classes for geographical and network functions (-) scheduler executes in randomised order, several incompatible collection classes, incomplete documentation
MASON	(+) fast, simulation and visualisation are separated, results can be reproduced; additional extensions for several domains	(+) fast simulation (fastest in the test), good choice for experienced programmers, clever innovations (-) only few tools, non-standard terminology, incompatible collection classes, lack of a terminal window for debugging
NetLogo	(+) Interpreted language = changes can be applied during runtime to tweak parameters, portable on different platforms (-) Slow, Visualisation constrained to a single window	(+) Easy to use, rapid prototyping, excellent documentation, good for simulating short-term, local interaction (-) only moderate complexity of agents can be modelled, no stoppable debugger, whole model in a single file, no direct access to design primitives
Breve	(+) Interpreted language = changes can be applied during runtime to tweak parameters, portable on different platforms; Physics engine (-) Slow, Visualisation constrained to a single place	
Ascape	Rule based simulations which react on certain environmental conditions (+) simplified modelling (-) Considerable constraints on simulation design due to simplification and the model	

For the project at hand, only Java Swarm or MASON can actually be considered, because according to both comparisons, these are the frameworks suited for complex, reproducible simulations. The lack of tools criticised by [Rai2006] is no problem when everything necessary for the actual project is available. And since both comparisons conclude that MASON's primary asset is its speed, it is an appropriate choice.

Usage in the project

In this project the core-libraries were used for the simulation itself. Agents and other entities are custom designed and are added to the scheduler with the appropriate interfaces. The environment is supported through the use of the 2D physics simulator package also available directly from the developers of the framework. For visualization, the 2D environment was used and integrated as an optional feature so the simulation can be run both with and without visualization.

2.3.3 Development Process

Every software project follows some kind of scheme to produce requirements, generate a design, allow cooperative development and implementation, evaluate the results and refine the design. Often the process itself is not written down and clearly formulated in guidelines, but evolves from initial work and experience to a kind of "how we do things" process as a general understanding among the team members. It can nevertheless be analysed and compared to well-known development processes or to the way similar projects were done.

Classifications for development processes

Although the number of published development processes is constantly rising, they can be grouped into more general categories. The differences between the models themselves are mainly in naming of the phases, roles of the actors and in documentation. Following categories have been described in [Bun2008]:

- Sequential processes: The steps of the development process are assigned to phases, which are then carried out in a sequential order. The most basic concept is the "phase" model, where the steps are simply requirement-analysis, design and implementation. But this process is not really in step with actual practice, because backtracking to earlier phases, which is not intended by the process, is actually very common in practice. Other models of this category therefore allow this to a certain extent, for example the waterfall-model, but still regard it as an exception. Nevertheless many software-projects are carried out in this way, since it is a very intuitive approach and works out for small projects [Bun2008].
- Prototypic processes: These models are basically sequential processes, but by-design demand to backtrack to earlier phases when a certain phase has been completed. For example, after the analysis phase a basic prototype is built, tested and then the analysis is refined to reflect the results of the test. The same goes for the design phase: here another prototype is built and from the tests both requirements and design may be changed [Bun2008].
- Incremental processes: These processes try to reduce complexity by creating the product in iterative cycles or spirals: The set of requirements is ordered and split into parts, then each

part goes through the complete sequential or prototypic process. When a part is complete more requirements are added and the enlarged system is build by following a sequential process. This is continued until the full system has been built. Depending on the specific definition of the process, phases of different cycles can be "pipe-lined" and carried out in parallel to save time. For example, the analysis and design teams can add new requirements and start creating a design-specification for the next phase while the implementation of the previous cycle is still being carried out [Bun2008, Lar2003].

- Re-use oriented processes: These processes are actually incremental processes, which very carefully select the order in which the system is developed. If this is done in a way that iterations represent abstract levels of a system, new products can re-use this completed work and start off from a basic product instead from nothing. For example, they could use the results of the prior project's 4th cycle and continue from there. Even though this sounds very sensible it is difficult to actually create these abstract levels for re-use. Often parts can only be re-used by adding wrappers or decorators to the existing code. In the last years the so-called model-driven development, which basically belongs to this category, has been developed. A domain-specific language is used to parameterise, configure. and enhance an existing base-system. The approach also allows separating the functionality developed in the domain-specific language from the bases system. This way, the produce of later stages of the development can be re-used when the base-system is changed as opposed to the opposite described above [Bun2008, Cza2005].

Agile methods

Agile methods were developed to reduce the growing amount of work imposed by fully-fledged development processes, which is not directed at actually producing software. These methods come in the same flavours as described above, but distinguish themselves from conservative processes by a higher degree of flexibility. The focal values hereby are [Abr2002]:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

This results in incremental processes with rapid cycles, many releases and a high involvement of the customer. Often the full requirements are not present at the beginning of the project, but are added or changed as the product evolves. Existing examples for agile methods are XP (eXtreme Programming) [Bec1999], SCRUM [Ris2000] or Feature driven development [FDD2009]. As the last is very similar to the applied process in this project, it will be explained in more detail:

Feature driven development is an agile method that can be put into the category of incremental development processes. The 5 phases of the process are [FDD2009]:

1. Develop an overall model: After a basic walk-through of the domain with a domain expert, independent teams propose models for parts of the overall system under guidance of a chief-architect. These are then combined to an overall model of the system.
2. Build a feature list: The domain is decomposed into subject areas, business activities within this area and steps within them to form a categorized feature list.
3. Plan by feature: The order in which the features have to be realized is planned depending on their dependencies. These are then assigned to chief-developers for realisation.
4. Design by feature: The chief developers receive packages of features. These are then designed, reviewed in a design inspection and then integrated into the overall model thereby extending it to add the new features.
5. Build by feature: The features are assigned to developers who implement them. The classes produced are "owned" by these developers, which makes them solely responsible for them. Before adding the features to the overall build, they are unit-tested and a code-review is done by the chief programmer.

Agent-Oriented development methodologies

Agents add complexity to all phases of the development process and therefore need additional practices or adaptations:

- Requirement-Analysis: New terms have to be introduced and defined which are not needed in other projects, e.g. agents, beliefs, goals, plans. In addition agents are active – as in oppose to objects, which are passive – and they can interact. To model theses concepts appropriate modelling-languages have to be used. [Bau2001] have proposed AUML (Agent Unified Modelling Language) as an extension to UML, which introduces additional schematics. [Cai2002] believed a completely different approach has to be taken and developed MESSAGE (methodology for engineering systems of software agents) which introduces a set of models to break down the requirements of the system to those for agents and their interactions rather than to class-diagrams.
- Design: Most methodologies split the design-phase into two main parts. The first part produces the agent-concept, i.e. which agents are needed and what goals do they have. The second part produces the realisation of these agents, the interaction and the environment. Examples for this are Tropos [Vre2004] or Prometheus [Pad2003].
- Implementation: The implementation process can also be seen as a combination of two parts: Firstly the environment must be created, only when the environment has reached a certain level of completeness the agents can be introduced, because they depend on a working environment which needs to offer at least the most basic functionality. To get results or test concepts early, existing frameworks or simulators are often used. This way agent-based prototypes can be developed, which are later ported to a new and more sophisticated environment.

Applied practices in the project

The aim of the project is to evaluate theories using the produced simulator. These theories and models are being enhanced over time and will probably also be changed or adapted to reflect results from the simulator as soon as they are available. It is therefore very difficult to specify all details of the system beforehand and it is also practically certain that new requirements will come up and the development will continue as soon as the simulator is put into use. This results in the need for agile methods, where dynamic response to changes is inherent to the process.

A second very influential property of the project is that it is carried out by a core team but also by students who contribute small parts in the course of practical lessons or diploma thesis such as this one. The project must therefore be divided into pieces, which can be developed independently under the supervision of members of the core team.

Both these arguments point towards the feature driven process described earlier, and even if the project team does not explicitly state it, the process is actually applied with very little adaptations and these are mainly due to the agent-orientation. In reference to the methodologies listed above, the variations are:

- Requirement analysis: The requirements which were created in cooperation with domain experts from the fields of neurology and psychology are either described as situations or stories the simulation must provide, or directly by the actual abilities the agent must posses. These can then be broken down as described in the feature driven model above.
- Design and Implementation: In accordance with the feature-driven approach, the system is being extended by adding feature after feature. But to cope with the agent-orientation this is done in two parallel processes – one process aims at developing the environment and the second at developing the decision model, which can be added by a plug-in like interface. Also, as mentioned above, a small prototype was created earlier to test certain theories and problems in regard to the agent-concept before starting the development of the actual simulator.

2.3.4 Patterns

Object oriented software development is a paradigm, which has had both great successes and large failures. On the one hand it reduces complexity of structural programming by creating independent units, which can be developed and tested independently before combining them to larger systems. On the other hand the number of possibilities for realizing a single objective is huge, but not all of them result in satisfactory results or differ in reference to scalability, performance or maintainability. To communicate good practices for solving certain design-problems and to have common names for referring to both a problem and the solution, [Gam1998] developed the concept of patterns. In general each pattern consists of following elements:

- Pattern name: The name of the pattern can be used as a reference and can thereby describe both the design problem and its solution. It allows communicating design-concepts on a more abstract level and in a more compact way.

- Problem: This describes when to apply the pattern by giving the context of the problem. This can be done be referring to certain algorithms, class structures or object structures and also conditions for when the pattern can be applied
- Solution: This part presents the elements that make up the solution, their relationships and collaborations. The solution does not describe a concrete design, but an abstract solution and how a set of classes or structure of objects can solve the given problem.
- Consequences: Here the results for using a specific pattern are given and also possible issues, which can come up when applying the pattern, are listed. By explicitly listing the consequences, the evaluation of possible alternatives is made easier.

The design-decisions made in the course of this thesis are described in terms of patterns, and the concrete patterns used will also be presented. As an example for this chapter, the singleton pattern will be given [Gam1998]. This pattern is also used in the project, but not in the actual components of this thesis:

Pattern name: Singleton

Problem: For some classes it is important to have only exactly one instance. The singleton pattern can be applied when

- there must be exactly one instance of a class, and it must be accessible to clients from a well known access-point
- when the sole instance should be extensible by sub-classing, and client should be able to use an extended instance without modifying their code.

Solution: Define a static method that lets clients access its unique instance. This operation has access to a static variable which holds the unique instance once it is created and returns this instance if it is already initialised or creates it by calling a private which cannot be called otherwise.

Consequences:

- Controlled access to a sole instance: Access can be restricted or controlled by the class itself
- Reduced name space: In contrast to global variables it avoids polluting the name space with global variables that store sole instances
- Permits refinement of operations and representation: The singleton class may be sub-classed and it is easy to configure the application with an instance of the class needed at run-time.
- Permits a variable number of instances: The pattern can be extended to allow the creation of a restricted number of instances.
- More flexible than class operations: Class operations cannot be enhanced to allow more than one instance, and – depending on the programming language – possibly cannot be overridden in subclasses.

In this project the Singleton pattern is uniformly applied instead of global variables, for example to get access to the simulator's scheduler.

3. Requirement Analysis

In the course of this thesis an artificial life simulator will be enhanced to allow the agent to call certain actions. The requirements were stated by the core team and several domain experts from the fields of neurology and psychology. In accordance with the agile development process they are mainly stated as short "stories" or scenes which indirectly imply what the agent needs to be able to do. They were collected in the project's wiki and are analysed in this chapter to produce the list of features which have to be implemented – in this case the list of actions the agent has to be able to perform and also all the infrastructural components needed for calling and managing them. In addition, the already existing architecture of the simulator has to be taken into account and the new components have to fit into the concept of the already existing framework. The intended use and nature of the project and it's design process also imposes several non-functional requirements like extensibility, performance parameters, which are listed in a separate section including the implications for the design. The final section lists further assumptions which are not explicitly stated in the requirements but which are crucial for the design and where agreed upon during a design meeting.

3.1 Use Cases

As mentioned above, most requirements are not explicitly listed as it would have been done in conservative projects, but were stated as small "stories". This comes more natural when people with different fields of expertise work together. The complete list of these stories can be found in Appendix I. As an example, one of these stories will be discussed in detail and the others will be listed in a table including the actions they imply.

As an example for the style of how requirements are stated, the use-case UC00 "The Lonely Life of a Hungry Bubble" is given:

"A bubble – a lonley one – roams around in a world. Whenever it stumbles across something eatable, and it is hungry, it will eat. Whenever necessary, the bubble rests or takes a short nap. The digestion system makes it necessary to get rid of the excrements. This will be done – similarly to eating and sleeping – wherever the bubble is at the moment of need. Other agents/bubbles are ignored – cooperation is not necessary in this sad and lonely world ..."

To allow an agent in the simulation to act as described in this description, the following must be possible:

- Movement: The agent must be able to move in the environment. This can be done by supplying two commands – one command for moving forwards or backwards and second command for turning left or right. The speed of movement and turning is not relevant in this context but it should at least be possible to override some default value.
- Eating: The agent must be able to find out if an object is eatable and then must be able to eat it. The search is done by the sensors, but a command for eating must be available. When it is called and an object is in range, the object or a part of it must be added to the digestion system so the hunger resides. In addition the object which is eaten must be able to react to the command, so it can, for example, shrink in size or disappear from the environment.
- Egesting excrements: For "getting rid of the excrements" a command must be supplied.
- Sleeping: The agent can get tired. This could be due to some arbitrary process, but with reference to the agent's already existing software-infrastructure, it is obvious that the agent's internal energy-level should be reduced by carrying out actions like moving, and can be refilled by sleeping. Therefore a reasonable energy-value must be calculated and consumed for each of the described actions and an action for sleeping must be supplied, which refills the internal energy levels.
- Cooperation: This description states that other agents are ignored and no cooperation should be possible. Since other use-cases state the opposite, it must be possible to enable and disable actions depending on the simulation at hand. This requirement is stated in the list of requirements concerning execution.

In this manner all of the use-cases where analysed. The complete sources can be found in Appendix I, a summary with the resulting conclusions is given in Table 1.

Table 2: Use Case Implications

UC	Use-case name	Implications
00	The Lonely Life of a Hungry Bubble	Action "eat" Action "sleep" Action "egest excrements" Actions for movement
01	Collecting and taking food home	Pick up/drop objects Carry objects Actions for movement
02	Planting energy sources	Pick up/drop objects Carry objects Action "plant" Action "cultivate" Actions for movement

03	Searching for type of Energy Source	Actions for movement Action “eat”
04	Transforming energy sources (cooking)	Action “cook” Action “eat”
05	Location Identification	No action-related requirements
06	Confronted with Harmful Energy Sources	No action-related requirements
07	Body Integrity	Respond to injuries by reducing functionality of the actuators
08	Reproduction	Agent can emit a certain odor Agent can mate and reproduce
09	Excretion	Action "egest excrements"
10	Motion Fatigue	Reduce speed or inhibit actions as response to lack of stamina
11	Different energy sources	
L1	Dancing	Dancing

3.2 Required Actions

The required actions come from two sources: Most come from the use-cases described above, but as a second source, an already completed feature-list of actions is available. This was developed during the course of a diploma thesis by [Koh2008]. It was then revised and compiled into a feature-list by the core team, but does not include all of the actions imposed by the use cases.

Since some actions need certain features which must first be available in the simulation and are not available yet some actions where also removed from the list. An example for this would be the “odour” mentioned in use-case “08 Reproduction”: this feature cannot be implemented yet, because a representation for scents is not available in the environment. Table 3 is the concluding list of extent of actions which need to be developed in the course of this thesis. For each action a short description and the source of the requirement is stated.

Table 3: Final List of Required Actions

Name	Description	Source
Bite	An agent can bite another entity when it is in range to harm it or kill and eat it.	Core team’s feature-list
Change body colour	The agent’s body colour must be able to be changed to reflect internal states such as illness.	Core team’s feature-list
Change facial expressions	The agent must be able to change: lens-size, lens-shape, left/right antenna position, eye-size.	Core team’s feature-list

Cultivate	Some objects should be able to be cultivated by an agent – for example plants supply more nutritional value if they are cultivated before they are eaten.	Use-case 02
Dance	An agent should be able to perform several different dances to express emotions or a certain state of mind: • salsa (happy) • tango (reproduce) • waltz (sad)	Use-case L1
Eat	The agent can eat certain objects when they are in range.	Use-case 00, 03, 04
Excrement	The agent can egest excrements after digesting the eaten food.	Use-case 00, 09
Kiss	An agent can try to kiss another agent triggering a response if the kiss is successful.	Core team’s feature-list
Lightning-attack	An agent can launch a long-range attack on another agent to harm or kill it.	Core team’s feature-list
Move to/from inventory	In addition to carrying a single object the agent can move objects to it’s inventory, thereby removing it from the environment.	Use-case 01, 02
Move, turn	Agents can move in a two dimensional environment.	Use-case 00, 01, 02, 03
Move to eatable area	An agent can move the entity it is currently carrying into the eatable area, i.e. the range of the “eat”-command.	Core team’s feature-list
Pick up/Drop	The agent can pick up objects in range which can then be carried when moving around.	Use-case 01, 02
Sleep	An agent can sleep to recharge it’s internal energy levels.	Use-case 00

3.3 Requirements Concerning Execution

The actions are one part of the project, the second part is to create the necessary infrastructure for calling, validating and executing them. Again a part of the requirements is derived from the use-cases like the need to disable commands for certain simulations. Some of the actions listed above also require additional infrastructure such as the inventory. On the other hand the scheme must fit into the existing architecture of the project which also imposes constraints on the design.

Availability of Actions

The complete list of actions is a collection from several different and independent simulation setups. For some scenes it should not be possible to call certain actions. For example, communication actions should not be possible in the "The Lonely Life of a Hungry Bubble" szenario. On the other hand a simulation may include several different types of entites – some with certain abilities, some without them. A mechanism must therefor be provided to enable or disable actions. Since the simulator uses a parameter-tree to configure the setup of the simulation for other parameters, this existing infrastructure should also be used for configurating the availabilty of actions for a concrete instance.

In another situation a normally available action must be temporarily disabled, for example when actions are not available due to damage (see use case "07 body integrity"). This cannot be done using the normal configuration routine, because that would only happen once during the setup of the simulation. Therefore another mechanism for temporariliy disabling and re-enabling actions must also be supplied.

But even if two different entity-types support an action, the result may be different due to the type of the entity. For example, the bite of a lion and that of a cat will have a considerably different impact on the entity that suffers it. Parameters for scaling the input/output ratio of commands like this must therefor also be supplied and allowed to be configured using the parameter-tree.

Calling Actions

The simulation is carried out in rounds with 5 stages as presented in Chapter 2. It must be possible to call commands for the actions during the first three phases preceding the execution phase:

1. Sensing: Commands representing bodily reflexes are called.
2. Update of the internal body state: Internal body states such as illness or fatigue are expressed by changing the body-colour or facial expressions like eye-size. These changes are also performed by calling appropriate commands.
3. Data processing: Most commands come directly from the data processing unit, e.g. eat, move, attack.

The actions cannot be carried out the instant they are called, because they have to be validated first which is only possible after all the commands for the round are available. The interface for calling commands must therefore provide a possibility to call commands and collect them in a stack until the execution phase.

Actions called during the sensing phase cannot be influenced by the decision unit in the data processing phase. This is the intended situation for reflexes, but if the decision unit can anticipate a reflex-reaction beforehand it should be possible to inhibit this action. For each action which can be called, a mechanism for inhibiting it must therefore also be available. During the execution phase calls and inhibitions are then paired and cancel each other out.

To model recurring routines it should be possible to create an individual list of commands and call them as a whole instead of separate commands each round. During each consecutive round of the simulation the items should be taken from this list and executed as if they had just been called manually. But if one of the actions in the list fails – regardless of the reason – the rest of the list should be discarded. An example for such a sequence of actions would be the “dance” (see use-case L1). This is actually not a single action, but a list of commands such as move forward, turn, move backward, etc. Instead of calling all of the commands separately the whole dance-sequence is called and the executed command-by-command each round until the list is completed or one of the commands fails.

The architecture of the simulator is designed to support multiple decision units. Therefore, the simulator and the decision-units were implemented in two separate packages. A second reason for dividing the project into parts is, that the decision unit should not be able to directly access the simulation but should only be allowed prepared sensory data as input and also only a predefined list of actions as output. For this reason the decision-unit does not have a direct reference to the main simulator package. Instead a third package – the decision-unit interface –, which is referenced by both of the other packages, contains all common interfaces and classes. Since most of the commands come from the decision unit, but are then actually executed in the main simulator, the interface for calling actions or sequences and inhibiting actions must be made available for both using this third package.

To compare different decision units a log-file will be written, which logs the sensory input and the command-output of the decision unit. There is already existing functionality for creating the file, but the called actions and their parameters must be made available for this and methods for creating the content must be provided.

Constraints on Actions

When all the commands have been collected they need to be validated against a set of constraints before they can be executed.

The agent in the simulator has an energy and a stamina level. Both are constantly refilled by a fixed amount each round and additionally by eating or sleeping. But certain body functions and sensors consume energy, as do actions. The decision unit of the agent has to balance this out by eating and resting between other actions. The energy and stamina demand of a certain action depends on the type and also on the parameters of the action. For example, moving fast will need more energy than moving slowly. If a command was called which demands more energy or stamina than is available, the energy and stamina should be drained to a minimum value, but the command should be discarded and not executed. If the command was part of a sequence, the rest of the sequence should also be discarded.

The remaining actions from the stack must then be checked in reference to one another. It should be impossible to call actions like bite and eat in the same round while other combinations like move and eat could be possible. A way of defining these mutual exclusions must be found so they can be checked prior to the actual execution. If a conflict is found, one or both of the commands must be discarded. To distinguish between actions three priority levels are needed:

- Normal priority: If two actions with this priority come into conflict with one another, both should be discarded.
- Reflexes: When two reflexes come into conflict with one another both should be discarded, but if one of the two is just of normal priority, the reflex should remain and the other should be discarded.
- State updates: State updates have the highest priority in case of conflicts. These actions should never be discarded, except when two state updates conflict.

The constraints described up to now must be checked for any action, but individual actions also may have certain additional constraints. For example the "eat" command can only be executed if something eatable is actually in range. Therefore, before the actual execution these action-dependant constraints must also be checked.

Executing Actions

The fourth phase of the simulation cycle is the execution round. After the constraints have been checked, the remaining actions will be dispatched. The actions defined in Section 3.2 can be grouped into three categories in respect to their functionality:

- Actions which change the agent's physical representation or state. These actions have no immediate effect on other entities and either adjust internal states – like sleeping – or send commands to the environment like move or turn. The command can directly access the state values and manipulate them as required for the concrete action.
- Actions which manipulate other entities. These actions involve calling functions on other entities and additionally may need to identify if an action can be called using another entity ("is the other entity eatable?"). Since the development process is feature driven and new features are added incrementally, it is not known a priori which entities may be created in the future and which of these will need to be supported by the actions. The commands should therefore not deal with the concrete entities, because this would mean that the implementation of the command would need to be changed every time a new entity is created, but rather with an interface which is implemented by the entities that support the actions.
- Actions which concern the binding of other entities. These actions set both internal states to keep track of the bound entities but also deal with an opposing second entity. Both to search for "carryable" entities and to inform the entity what is happening to it an interface should be used similar to the previous category.

Managing Binding States

To manage the binding of entities, an additional component must be created which is not already available in the existing implementation. This "inventory" should be bound to the agent like a body-part and is used to keep track of objects that where picked up or moved to the inventory by the appropriate actions. When objects are carried they should be linked to the entity in the simulation environment, so when the entity moves, the bound object is moved too. The impact on the movement, i.e. slower movement with the same energy, is handled by the underlying physics engine and does not need any further attendance. Carrying other objects should also consistently consume energy proportional to the weight, but also proportional to a configurable parameter.

In addition, a carried object should be able to be moved to the inventory (and back out of the inventory) by special commands. When an object is in the inventory it is completely removed from the simulation environment and cannot be reached by any sensors or by other actions. The agent must therefore keep track of the inventory-contents in its memory. Keeping objects in the inventory also consumes energy, but less than carrying them. In addition the movement is not impaired and the agent can move more freely.

3.4 Non-Functional Requirements

Besides the functional requirements regarding the actions and the execution process presented in the previous chapters, additional needs exists which also have to be taken into account when designing the components. From all the possible parameters which can be discussed regarding non-functional requirements many are not relevant for this project, e.g. usability, security or configuration management. But the applied design process and the fact that this is only a part of a much larger project which involves many more people need special consideration.

Extensibility and Modifiability

The feature driven design process applied in the project is an incremental process as explained in Chapter 2. The whole project is split into parts, which are themselves split into features and added piece-by-piece. Additionally the simulator in this project is built to verify theories and additional requirements will surely come-up as soon as the software is put to use. Therefore efforts must be made to allow for later enhancements and adaptations. This need can be supported by taking care to encapsulate functionality as much as possible and keep the parts highly independent. The most probable changes and enhancements that may come-up, are new types of agents and new actions. New types of agents can be dealt with by configuring the available commands and their parameters as described earlier or – respectively – by using interfaces for interaction among entities also as described earlier. The main additional requirement is therefore to ensure that new actions can be added as easily as possible.

Documentation

The normal application of feature driven development would make the creator the “owner” of his classes and solely responsible for them. But due to the fact that the project team only has a small core team and other developers come and go in the course of internships or diploma thesis such as this one, it is highly likely that the created code will later have to be modified by someone else. The code should therefore be extensively documented, structured logically and written in the style of the existing software to allow fast comprehension.

The actions themselves will be both configured and called by other people, especially by the developers of the decision unit. A means of describing them in a compact but exhaustive way must therefore be found. It should be as easy as possible to select the appropriate command for a desired action and to find out how to use it.

Testability and Robustness

As described in the background chapter, debugging and testing agent-based software is not as easy as it is in other types of software. The simulator may run for hours and then exit due to some run-time-error which can only be reproduced by running the simulation for the same amount of time again. To start off with the highest possible robustness the components should be pre-tested individually via unit-tests. These have the additional advantage of being able to be called in automated tests after each build. The project team uses JUnit to create unit tests which is integrated into the development environment as mentioned in the background chapter. Even if many features cannot be tested in this way because they need the whole simulation environment up and running, the basic functionality of calling, validating and executing a command should be covered by extensive unit-tests.

Performance

Even if performance is always a key quality attribute for software, it is not of the highest importance in this project, since it is not a real-time system intended for frequent user-machine interaction, but will run independently until some exit criteria is met. Of course care must be taken not to waste resources, but given the decision between optimising for speed and keeping structures simple and easy to modify, the latter should be opted for.

3.5 Additional Assumptions

The design is based on the requirements stated above, but to further reduce the design space additional assumptions have to be made. These assumptions were discussed with the core-development team and it was agreed that the implications of these assumptions pose no problem to the intended use of the simulator. As a reference and to explain design-decisions made in the next chapter the most crucial assumptions will be presented here including their consequences.

Atomicity

To ease the processing of actions, the commands are treated as atomic operations which last exactly one simulator round. When they are called, they can be processed and checked for validity independently (except for mutual exclusions among the actions of the current simulation round). If the action cannot be performed due to lack of energy or because action-specific constraints are not met, the action is discarded without further ado. If it is valid, it is dispatched as a whole and then discarded. The benefit of this is, that only a single execution stack needs to be managed where actions for the current simulation round are collected and processed. There is no need to reference future or past actions and there is no need for further tracking the action after it has been processed this one time.

All actions are designed this way, so actions which should be carried out over several execution rounds must be called consecutively round-by-round. For example, sleeping for 10 rounds has to be done by calling the sleep command 10 times. An alternative is to use action-sequences which can be called a once, but may contain actions for several consecutive rounds.

Statelessness

In addition to atomicity, actions are stateless. An action can only inflict a change in the environment, an internal body-part, or a sensor, but it has no state of it's own. Actions which need to reference a state that was not previously available must be split into the action itself and an object which can hold the state and is bound to the agent's body.

An example for this would be the sleep command. This command should have several consequences, one of which is that when the agent is asleep certain functions are not available, e.g. sensory input or actions such as move, eat, attack. To determine if the agent is asleep one would need to check a sleep-state variable which cannot be provided by the command itself. In this situation, two solutions are possible:

- A Sleeping attribute is introduced as a property of the agent-body. A command "sleep" sets this attribute and any operation which is influenced by the sleep state must check this attribute before continuing with it's normal operation, and other commands which cannot be performed during sleep must check the variable when checking the other constraints. A second command "wake-up" would reset the sleeping attribute and all functions would be available again.
- The second possibility would be to abandon the state variable entirely and instead of forcing every influenced component to check a state variable, the command would directly call a function which deactivates the relevant components for the next simulation round. Actions which are not available when sleeping could be added to the list of mutual exclusions for the sleeping command and would not need any special treatment. Since commands are atomic, the sleep command would need to be called consecutively during the whole sleeping phase (which could be done creating a sequence).

No Feedback

By design, an actuator does not give feed-back about whether the action was performed or not. Commands are called and put on the stack, but the agent cannot track them and determine if they were actually performed or not. Instead it must use its sensors to indirectly check if the action was successful and had the desired result. For example when a move command is called the agent can determine the success by checking if the visual sensors reflect the desired new position. In some situations multiple sensory input could also be available. For example, a collision is sensed when hitting an obstacle and the visual sensors also provide the information that the position was not changed.

In most cases ready-available sensor-information can be used directly to determine the result of an action, but in other cases the information must be "injected" by the actuators. For example if the agent tries to eat when nothing is in range, no change in the environment would result. In this case the information will be returned by creating a fast messenger representing a painful sensation as a form of sensory input.

Handling of Opposing Entities

Some actions require an opposite entity. Eating, for instance, requires an entity which is eaten. It was agreed upon that when calling the command the opposite entity is not given as an argument. Instead the command will search a defined sensor area and use any valid entity in this range. If none or more than one entity is found in the action's range, the action will not be performed. The energy needed for the action will be consumed nevertheless, but depending on the action a response in the form of pain may result like when trying to bite "nothing".

This has influences in the pattern of commands when designing the agent's decision unit. When an agent sees something eatable it does not call a command like "eat entity X" directly referring to the entity, but instead must move in a way to ensure that the entity is in range and then call the command "eat" without further parameters.

Constraints on Binding Entities

Since actions are stateless, as mentioned above, actions which bind other entities in a persistent manner need a place to store this information. For this an inventory-object will be created which can be referenced through the agent's body. An agent can bind to another entity in two ways:

- Carrying: An agent can decide to pick up an object. It is then linked to the agent until dropped or put into the inventory, but an agent can only carry exactly one object.
- Inventory: An agent can put an object it is already carrying into the inventory. This removes it from the simulation and it cannot be used or seen by other agents until it is taken out again. The inventory can only hold a defined maximum number of objects and a certain total weight.

4. Model

Before dealing with the details, the basic requirements will be discussed on a more abstract level and patterns will be selected which promise the best possible solution. These will then be combined to create the model for the concrete requirements collected in Chapter 3. First the execution process will be discussed and the basic model will be presented and argued for with reference to the requirements. Afterwards the model for the actions will be given using the groups already defined in Chapter 3. Additional classes will also be defined which are needed to store persistent information for the actions such as binding states. Finally the non-functional requirements concerning documentation will be dealt with and the developed solution will be presented.

4.1 Applied Patterns

As stated in the background chapter, the concept of patterns was developed by [Gam1998] and is a way to document and communicate best-practice solutions for reoccurring problems. Instead of developing proprietary solutions, patterns will be applied wherever possible. Before combining them to the complete model the selected patterns will be presented and the decisions for choosing them will be given.

4.1.1 Command Pattern

As stated in the requirements, it must be possible to call actions during all phases of the simulation cycle. On the other hand, they cannot be executed instantaneously, because they have to be verified first and should only be executed in the execution phase of the cycle. Therefore, the commands must be collected on some kind of stack. The Command pattern is one of the behavioural patterns defined by [Gam1998], and it's intent is to "encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations" and later: "Use the Command pattern when you want to specify, queue, and execute requests at different time". Since this is an exact description of the problem at hand, further analysis of the proposed solution should be done.

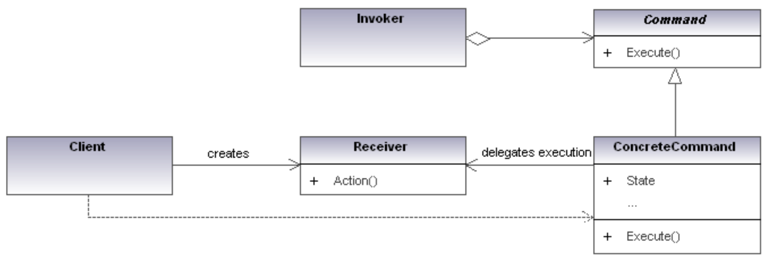


Figure 7: Command Pattern

The proposed solution contains following classes as can be seen in Figure 7:

- Command: The command is the interface for calling the actual execution of the operation.
- Concrete Command: Defines a binding between the command and the receiver. The execution is implemented by calling the appropriate action on the receiver.
- Client: The client creates a concrete command object and sets the receiver.
- Invoker: Calls the execution method of the command at the appropriate moment.
- Receiver: Knows how to perform the actual operation associated with the command.

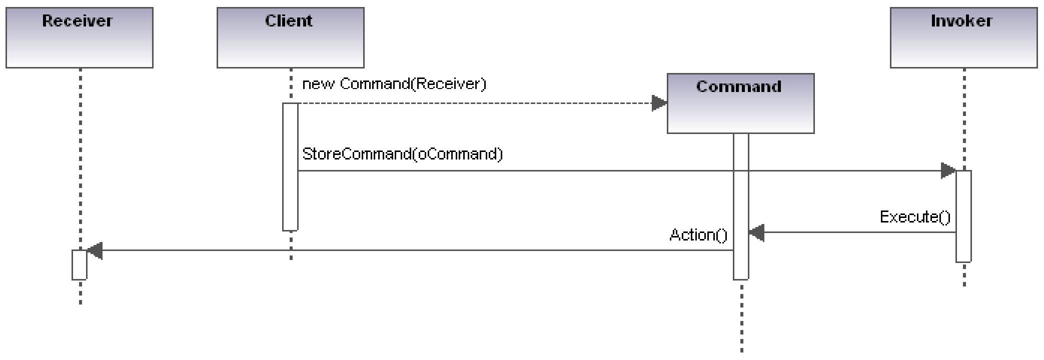


Figure 8: Command Pattern - Sequence diagram

The sequence-diagram in Figure 8 shows the interaction that takes place when an operation is called:

1. The client creates a concrete command object and specifies the receiver.
2. The concrete command object is then handed over to the invoker who stores it for later execution
3. At the appropriate time, the invoker calls the execute method on the command interface, which is implemented by the concrete command. This, in-turn, calls the action on the receiver.
4. The receiver then carries out the actual operation

With reference to the requirements, the application of this pattern can help solve several problems: First and most important is that the call and the actual execution of the command can take place at separate times. This solves the requirement that commands need to be called during the sensing, updating and processing phases of the cycle, but have to be executed later during the execution phase. Additionally, commands are encapsulated in separate and independent classes. New commands can therefore be added without changing existing code, which supports the requested simple extensibility.

4.1.2 Command Processor Pattern

Following the requirements, the commands – represented by the command objects – must be collected, validated, executed and logged. The Command pattern already offers a general interface for all commands. The Command Processor pattern described by [Bus2007] takes advantage of this fact and proposes a centralised, general processor class instead of the invoker in the command pattern. Since this generalized invoker only deals with the abstract command interface, it can handle any class implementing it if the interface supplies all the necessary parameters needed for the processors tasks.

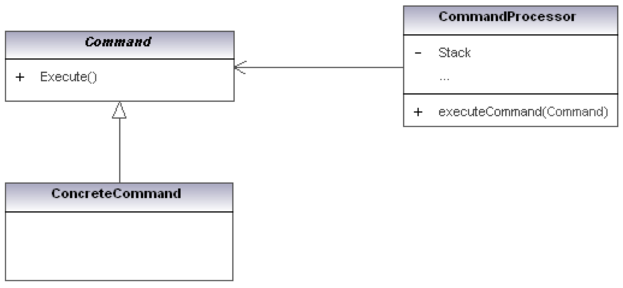


Figure 9: Command Processor

Figure 9 shows the structure of the solution and contains following classes:

- Command: This is the same interface as described in the command pattern and contains the execute – command and any other general attributes or methods needed by the processor
- Command Processor: This is the substitute for the invoker class in the command pattern. It incorporates a stack on which new commands can be laid and coordinates execution, logging and any other general operation needed.
- Concrete Command: This is the same class as in Figure 7 and contains the actual implementation of the operation.

The application of this pattern offers a general structure for the complete component. The commands can be collected in this centralised component until the execution phase. The processor only deals with the general interface, so the benefit of encapsulation gained by the command pattern is not lost. Using this interface, properties can be exposed to allow the processor to handle all the general constraints such as energy consumption, mutual exclusion, inhibition and logging in addition to invoking the actual execution.

4.1.3 Composite Pattern

Not only single commands but also whole sequences of commands should be available. The processor described in the previous chapter would therefore need to be able to handle two different interfaces – the command and the sequence. To overcome this double implementation, the Composite pattern can be applied, which was also defined by [Gam1998]: “Composite lets clients treat individual objects and compositions of objects uniformly.”

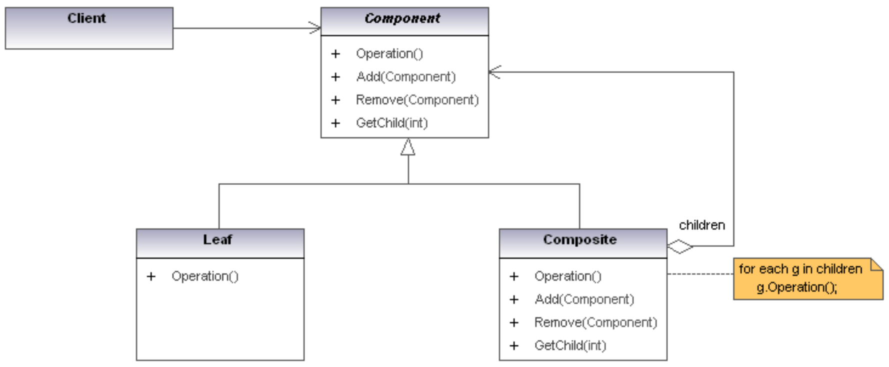


Figure 10: Composite pattern

The client in Figure 10 only deals with one unique interface – the “component”. From this abstract class both single objects (leaves) and composites inherit. These composites can –in turn – again contain leaves and composites at any arbitrary complexity because they also only reference them via the abstract component interface. When the client calls a method of the component, it is passed to all objects keeping the actual structure transparent to the client.

Even if the pattern allows handling hierarchical trees of objects, the problem at hands only requires dealing with single commands and lists of commands. But if later requirements arise and demand more complex structures than lists only a new composite class needs to be created and neither the commands nor the processor must be changed. Using this pattern therefore also backs up the requirement for extensibility.

4.1.4 Proxy Pattern

The actions themselves have been grouped into categories in Section 3.3. One group of these manipulate other entities, and it was argued that it is necessary for the interface to the opposing entity to be kept abstract and independent. This is to ensure that new entities can be created later which support certain actions without having to change the code of the execution itself. These actions must therefore not deal with the concrete classes, but with a generic interface which can be implemented by any entity which supports the action.

A similar problem is also imposed by the structure of the project. As mentioned in the requirements, the decision unit and the main simulation are divided into separate packages with no reference from the decision unit to the main simulator. The processor must be implemented in the main project so it can execute the commands, but these commands are, in part, called by the decision unit. This situation is similar to the one described before. The calling object may not deal with the actual class directly, but this time because the structure of the project makes it impossible.

This general problem can be overcome by applying the proxy pattern – again from the collection of [Gam1998].

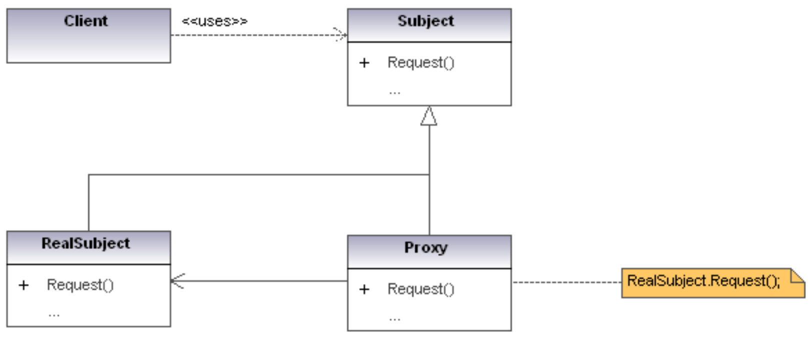


Figure 11: Proxy Pattern

A substitute – the proxy – is placed between the client and the real subject, as can be seen in Figure 11. The proxy can control access to the real subject and hide the actual implementation to the client. The simplest form of the proxy is an interface instead of a separate class. This interface is implemented by the actual subject. The client only accesses the real subject through this interface and therefore the implementation is hidden. If additional functions such as logging, access control or dynamic changes to the reference are needed, the simple interface can be replaced with a separate proxy class later without having to change the client or the real subject.

With reference to the concrete problems described above, each command which needs an opposing entity will access it only through a proxy-interface. This way new entities can be created and all that needs to be done to support a certain action is to implement the interface – no changes to existing code need to be made. Regarding the processor, a proxy class or an interface needs to be implemented in the decision unit interface. The decision unit can only access methods provided by this proxy and later enhancements such as logging or more complex access restrictions, can be easily implemented without having to change the processor or the decision unit. Using this concept it would also be possible to make a runtime separation of the decision unit and the simulation, because the proxy object can be used to transparently access the simulator through an Internet link or any other communication medium instead of a direct reference. Again, all that would need to be changed is the proxy and not the simulator or the decision unit.

4.2 Basic Concept for Calling and Executing Commands

In Section 4.1, patterns were presented which solve some of the issues imposed by the requirements. The main structure for the whole action-calling concept is based on the Command and Command Processor patterns. A problem which has not yet been discussed is, that the concrete commands have to be available in the decision unit interface, but the execution will need to reference objects in the simulator which are not available there. For this reason the receiver is not set by the client as suggested in the pattern, but will be set by the action processor and referenced as “executor” in later descriptions. The concrete model will therefore be comprised of following components as can be seen in Figure 12: the action command, sequence, processor and executor.

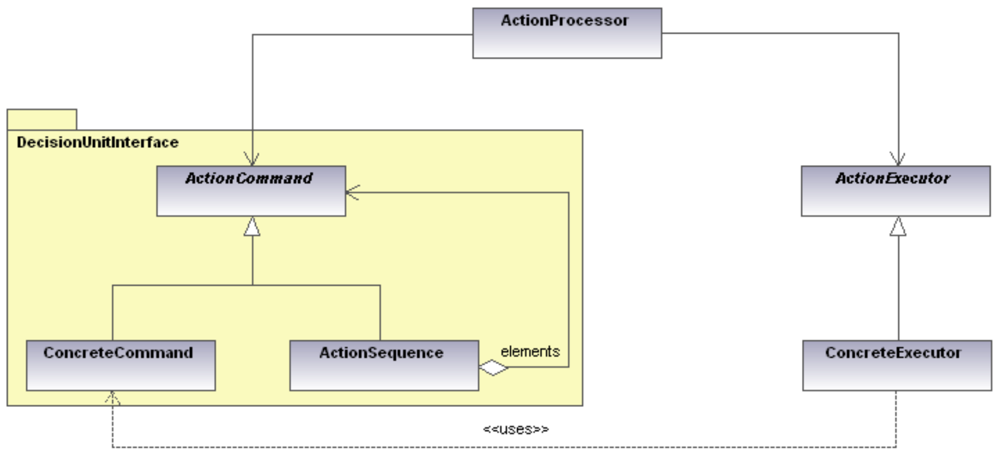


Figure 12: Class diagram of the model

4.2.1 Abstract Action Command Class

Each concrete action will be implemented as a single class extending an abstract action command as defined by command pattern. This interface will be the only one available to the action processor and must therefore provide all relevant functionality:

- Retrieve command: A single action is only comprised of one command, but the action sequence can contain several. Applying the composite pattern, a method to retrieve all commands will be implemented which hides the actual structure. The abstract class will implement this function with the default behaviour of returning just itself as the only command, so basic actions do not need to override it.
- Check if completed: Following the same argumentation as before, the processor will need to know if all actions have been performed or if it continues to the next round. For this a method must be available which accepts the current round as parameter and returns true if the whole composite has been completed. The abstract class will implement this function defaulting the command itself for the duration of one round.
- Logging information: Because all parameters concerning execution are implemented in the action executor, the abstract action command only has to provide a method, which returns the name and parameters of the command. The logging infrastructure creates an XML file, so the value returned must be an XML node representing this.

With reference to the vocabulary used in the Composite pattern, the default behaviour is that of a leaf, but it can be overridden to represent a composite. In both cases it hides the actual structure behind the interface so the client – in this case the action processor – can handle all of them in the same way.

In the requirements, the atomicity of actions was defined as an assumption, so only commands with the duration of exactly one round would need to be supported. But since the Composite pattern hides the actual structure behind the "retrieve command" method, this constraint could be loosened if the actual need arises.

This class must be implemented in the decision unit interface, so it can be referenced by the decision unit and also by the action processor in the main package.

4.2.2 Concrete Action Command

For each action, a concrete action command class must be created and implemented in the decision unit interface. The class must extend the abstract action command so it can be managed by the processor. The two methods "retrieve command" and "check if completed" can be used as implemented in the abstract class, only the logging-method must be overridden and return the command and it's parameters as an XML node.

The command will be passed to the executor by the processor and must therefore supply all the parameters for the concrete action in addition to the inherited interface, but it may not contain any state variables due to the statelessness property. Each type of command has its own executor, so these attributes do not need to be masked behind another interface, but can be provided directly as proprietary properties of the concrete command. For example, the "move" action may need attributes such as speed and direction, which can be implemented with appropriate member variables and getters/setters.

4.2.3 Action Sequence

According to the requirements, it must be possible to call whole lists of actions in addition to single ones. The action sequence will be represented by a single class and implemented in the decision unit interface. It must provide functions for adding and retrieving actions from a list. Actions can be added by passing a concrete command object and giving duration (number of cycles the action should be carried out), and the execution round it should start. The execution round is not the absolute number, but relative to the point in time when the sequence was initially called, i.e. 0 is the current round, 1 the next round, 2 the round after that. Following the Composite pattern, the concrete command added to the sequence could also be another sequence and thereby create a tree of commands. As already mentioned, this concept can be enhanced to even more complex constructions, but at the present time the requirements do not demand this.

The methods "retrieve command" and "check if completed" must be overridden so all commands for the current round are returned or – respectively – checked. The method for retrieving the logging information must also be overridden and return the concatenated list of all the information returned by it's commands list in XML format.

4.2.4 Action Executor

In the Command pattern, the "receiver" responsible for executing the actual command is normally set by the client. In this project the pattern had to be adapted because the client – the decision unit – cannot reference an object for execution because it needs to be implemented in the main simulator project. To cope with the situation each command is mapped to an executor – the receiver – and the relation is stored by the action processor and configured during the simulation set-up.

All concrete executors extend an abstract action executor, so they – like the command – have a common interface for the processor. The action processor needs this common interface to stay independent of the actual actions, but still all information necessary for carrying out the processor's general validation tasks must be available. For this, the interface must provide the following functionality:

- Mutual exclusions: A list of action command types which cannot be executed at the same time as this executor.
- Energy and stamina: Methods for determining the demand have to be implemented. They accept an instance of a concrete command of the appropriate type as parameter and return the energy and stamina demand. The command as a parameter is needed because its attributes may have an influence on the calculated demand. For example, moving fast is more costly than moving slowly.
- Execution method: The actual execution method will be called after all general validation work has been done. Since the processor can only check general constraints, the concrete executors must check any additional command-dependant constraints prior to execution. The method returns true or false depending on whether the action was successful. This result value is used for optional logging only and is not passed back, or made available to the decision unit as defined in the requirements.

The concrete action executor can be seen as a software representation of an actuator. It accepts commands of the appropriate type and executes them using the command's attributes like direction and speed in case of movement. But the executor itself can also be configured via its concrete interface in order to change the behaviour as requested in Section 3.3. In accordance with the rest of the software, these attributes are set via the parameter tree during start-up.

4.2.5 Action Processor

The central class of the executing component is the action processor. This class is responsible for solving several requirements: It must have an internal configuration interface to enable and disable actions, it must provide an interface for calling actions or sequences available to the decision unit, and finally it must validate and execute the commands.

Configuration Interface

As stated in the requirements it is necessary to define which actions are available. This must be possible during the set-up of the simulation and also later for dynamic changes. Since the Command pattern will be followed, all available commands are represented by a concrete command class inheriting the abstract command class. Using reflection, the list of available commands can therefore be represented by a list of types. Thus the action processor will supply the methods "enable" and "disable" which accept the reflection type for the concrete command as an argument and store these. These methods can be used both during start-up and also during execution of the simulator and therefore satisfy the requirements.

In addition to the en-/disable methods, all available commands must be mapped with an executor during start-up. For this, a registering method must be available which accepts the reflection class of a concrete command and a configured instance of a concrete executor. These must be stored in the processor so when a command is called, the appropriate executor can be found and used. By default the registering of an action also enables it, which means that if the opposite is required, the disable method must be called afterwards.

Calling Interface

The action processor provides an interface for inhibiting and calling actions. Both methods accept a concrete command object as a parameter and store them until the execution phase. Since all concrete commands and also the action sequence extend the abstract action command following the Composite pattern, only one method for calling and one for inhibiting is necessary. In accordance with the no-feedback definition, the methods do not return a value and it is not possible to track the command after it has been submitted. To be able to solve mutual exclusion conflicts as defined by the requirements, a second optional parameter sets the priority level of the call to be either normal, reflex or body update (default is normal).

To allow the decision unit to use the interface, the processor implements an interface stored in the decision unit interface to publish it's public methods. Since no references from the decision unit to the main simulator are allowed, no function not published by this interface can be used. Additionally, the concrete action processor can be substituted by a proxy implementing the same interface without having to change the implementation of any of the participating classes. Via this proxy, additional functionality could be provided as already mentioned in the pattern description, e.g. more sophisticated access control or transparent communication via Internet.

Execution Interface

The action processor must implement a single execution command without parameters, which will be called by the simulation framework during the appropriate phase.

All commands have been collected on a stack and can be accessed via the interface of the abstract command class. Since the atomic commands are masked by the interface, the processor must first get all actual commands by looping through the stack and calling the method for retrieving the commands for the current round. The disabled or inhibited commands are then immediately removed from the list. These actions can be ignored completely, so they will neither be executed, nor consume energy nor interrupt a running action sequence. All remaining actions are paired with the executor and thus represent the current execution stack.

The next step is responsible for energy and stamina consumption. The amount needed for the individual action can be determined by the methods provided by the command executors which are available through their common action executor interface. If the available energy or stamina is not sufficient for all the actions in the stack, the stamina will be drained to zero and the energy will be drained to a value proportional to the stamina used, i.e. if the demand was 100 stamina and 10 energy, and the stamina level is only 50, then 50 stamina and 5 energy will be consumed. The energy value is not checked and can therefore also drop below zero which will "kill" the agent.

All remaining actions then need to be checked against each other for mutual exclusions in a double loop. The abstract action command provides the list of mutual exclusions as a list of types of action commands which can be compared with the object instance's type via reflection. In the case of an exclusion, the priority with which the action was called determines which action should be discarded following the description in the requirements: Body update calls will always be executed, reflexes win over normal actions, and if both actions are of the same priority, they will both be discarded.

After reducing the list of commands, all remaining actions are executed by calling the appropriate function of the executor defined above, passing the command as a parameter. The return value (true or false) is saved for logging purposes only, as is the reason for not executing the action if it was dropped.

At the end of the execution phase all completed commands are dropped, which can be verified by using the "check if completed" method of the abstract action command class. Since all commands are stateless by definition, the command can be dropped without loosing any information.

Additional Functionality

In addition to the actual execution tasks, the action processor needs to provide further functionality to meet all requirements:

- Logging: The processor must provide logging information on all of the commands which were called in the current round. For this a method must be implemented which returns the list of all the command's logging information which is available through the corresponding method of the abstract action command.
- Execution history: The "no feedback" definition makes it unnecessary to provide information about the execution status. But for testing the processor it is essential to know when and why a command was discarded or if it was executed on schedule. For this, a method will be implemented which returns the list of commands that were on stack in the last round and the state of execution or non-execution including the reason for failing. This function will be hidden to the decision unit by not publishing it in the decision unit interface.

4.2.6 Execution Sequence Diagram

Figure 13 shows the sequence diagram for the execution of a single command. During the set-up phase the executor and command were registered at the action processor by creating an instance of the executor and passing the combination command-type and executor instance to the processor. Later, in the processing cycle, the decision unit creates an instance of a concrete command and sets the parameters. It then passes the command to the processor, who stores it on a stack. In the execution phase the processor gets the validation information for the commands via the executor (energy demand, mutual exclusions, etc.) and loops through them validating them. The remaining commands are then executed by calling the execute method of the corresponding executor, which performs the operation using the parameters of the command and returns its success back to the processor.

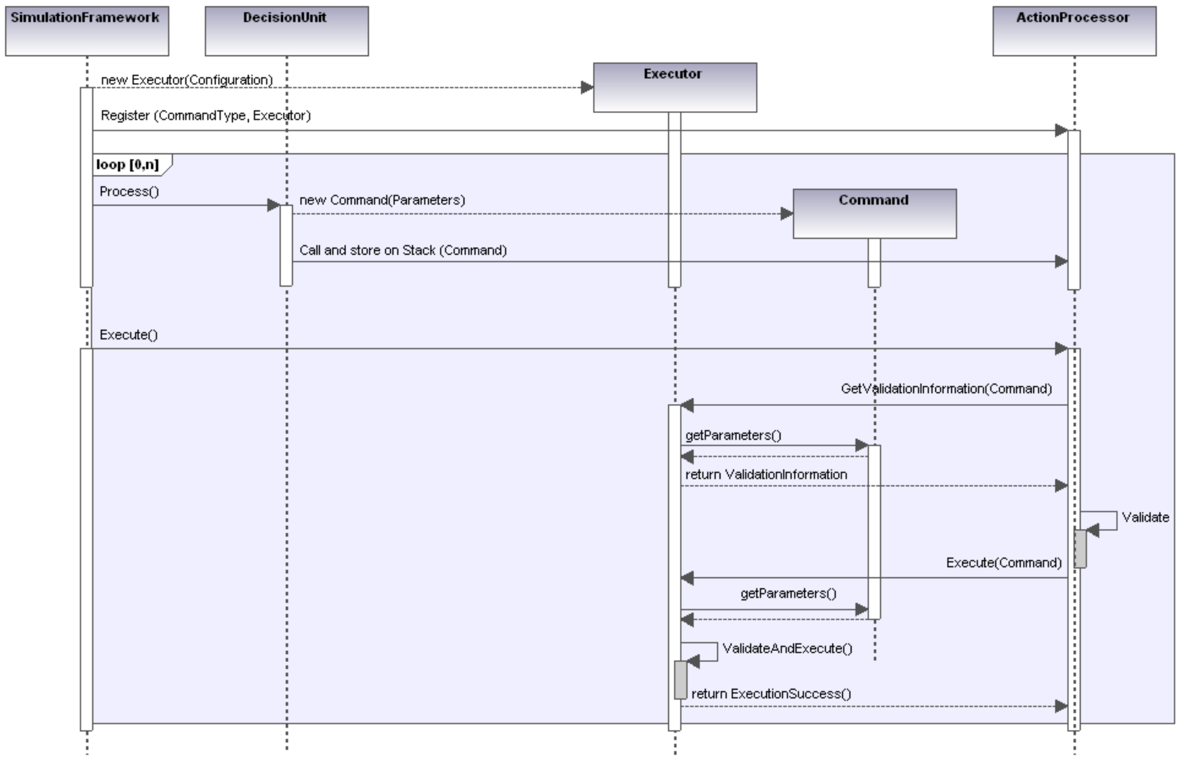


Figure 13: Execution Sequence diagram

4.3 Concrete Actions

In the requirements, the actions where divided into three categories:

- Actions which change the agent's physical representation or state
- Actions which manipulate other entities
- Actions which concern the binding of other entities

The first category of commands can directly access the appropriate objects and set the values or call the methods needed for the operation. All work can be done inside the execution method of the action executor without changes to other components, so no additional interfaces or masking needs to be done. For the other actions, proxies will be used to decouple the command and the entities as was already argued for during the description of the pattern. The third category additionally needs a new class to store the binding information. The first subsections present the logic applied for proxies and the new inventory class. Afterwards a table will be given containing all the actions listed in the requirements and the elements needed for the actual implementation.

4.3.1 Proxies

All actions besides those of the first category need to changes state-values or call methods of other entities. Since new entities may be created later, the action executor would need to be changed if the new class should support the command. To decouple the executor from the entity the proxy pattern will be applied, and for each action which needs to manipulate another entity an interface will be created: The executor deals only with methods provided by the interface and has no need to know more about the concrete class. This way new entities can be created and made to support a function by implementing the appropriate interface. This scheme can also be utilized to create test-containers where the executor deals with a dummy-object that only logs if the executor calls the functions properly.

The requirements state that the opposing entity is not passed as a parameter but should be selected automatically by looking for an appropriate entity in the agent's range. This can also be accomplished with the interface as described above. Using reflection, all entities in range can be checked to verify if they implement the interface needed for the concrete action. If they do, and if there is only one valid entity in range, the reference will be treated as the opposing entity and can be used directly.

4.3.2 Inventory Class

For actions which bind other entities an additional class must be created as argued for in the requirements. Every agent that supports the binding commands must provide a reference to an instance of this "inventory" class which needs to support the following functions:

- Pick up: As defined, one entity can be bound to the agent and is joined by a link so both move through the environment together. The inventory class must therefore provide a function to set the reference to this entity and establish the link. If another entity was "picked up" previously, the existing link will be dropped before the new one is established .
- Drop: This method drops the link to the currently carried entity.
- From/to inventory: One method moves the currently carried object to an internal inventory and removes it from the simulation environment. The second one moves an object from the inventory and binds it as the carried/picked-up object. The inventory can only be filled up to

a configurable maximum weight and number of objects. If the limit is reached, the methods throw an exception which must be handled by the calling action command.

- Energy and stamina demand: Carried objects, and objects in the inventory need a certain amount of energy and stamina each round. The value is calculated by the inventory class and the action processor retrieves and consumes that amount together with the total from the other actions.
- Move to area: This method is needed as a support function for the corresponding command. The default joint between the carried entity and the agent is changed so the carried object is "pulled" to the desired destination. The method needs the destination area as parameter.

Following the same logic as the other operations, an interface is needed with which these commands interact. This "carryable" interface must be implemented by any entity that can be picked up. The interface will be used both to identify "carryable" objects and to set the binding state of the entity through an appropriate method. This way the entity "knows" what is happening to it and can react accordingly. Plants, for instance, should stop to grow when they are carried. The interface also supplies a method to retrieve the weight of an object so energy demands can be calculated and the constraints can be checked. Additionally, a reference to the object representation in the simulator is needed so the joint can be established and the object can be removed from or returned back to simulator environment.

4.3.3 Table of Actions, Executors and Proxies

Using Table 3, a list of commands, executors and proxies can be derived following the model presented above. Table 4 shows the result and lists the commands needed for each action including their parameters, a description of then executor's functionality, and a definition if a proxy interface is necessary.

Table 4: Actions, Executors and Proxies

Action	Command(Parameters)	Executor	Proxy
Bite	AttackBite (Force)	Search for entity via the proxy, bite it and effect damage on it or the caller.	Yes
Change body colour	BodyColor (Red, Green, Blue) BodyColorRed (Red) BodyColorGreen (Green) BodyColor Blue (Blue)	Set values via the agent's internal body components.	No

Change facial expression	EyeSize(Size)
	LeftAntennaPosition(Position)
	LensShape(Shape)
	LensSize(Size)
	RightAntennaPosition(Position)
		Set values via the agent's internal body components.	No

Cultivate	Cultivate(Amount)	Search for entity via the proxy and cultivate it.	Yes
Dance	Sequence for Tango Sequence for Walz Sequence for Salsa	None (sequence of move and turn commands)	No
Eat	Eat()	Search for entity via the proxy, eat it. Inflict damage or pass nutritional value to stomach.	Yes
Excrement	Excrement (Intensity)	Create an excrement entity from the stomach system and drop it in the environment.	No
Kiss	Kiss(Intensity)	Search for entity via the proxy, try to kiss it and create the internal response if positive.	Yes
Lightning-attack	AttackLightning (Force, OpponentID)	Search for the named entity and attack it via the proxy function.	Yes

Move to/from inventory	FromInventory()
	ToInventory()
		Move the carried object to/from the inventory using the inventory class.	Yes , Carryable

Move, turn	Move(Direction, Speed) Turn (Direction, Angle)	Add the appropriate forces in the physics engine.	No
Move to eatable area	MoveToEatableArea()	Move the carried object to the destination area using the inventory class.	Yes , Carryable
Pick up/Drop	Drop () PickUp ()	Search for entity via the proxy and bind it to the entity via the inventory class.	Yes , Carryable
Sleep	Sleep(Intensity)	Notify a set of internal components about the sleeping state.	Yes

4.4 Documentation

In the requirements the necessity for a useful and extensive documentation was explained. Especially the actions must be documented in a way so other developers can use them and find the appropriate command easily, and without having to analyse the code.

Since all actions are based on a single strategy, the model allows describing them using following parameters:

- Description of the behaviour: The first and most relevant information is what the command actually does and what it should be used for.

- Stamina and energy: When implementing an action, the developer needs to know how much energy and stamina is needed. Since commands are not executed if the demand cannot be met, the decision unit must be designed in a way to balance out the demand.
- Command parameters: Every parameter available for the command must be described. If possible a default or recommended value should also be given.
- Executor parameters: When implementing new entities, the available commands must be mapped to an executor. These can also be configured to reflect differences between entities like the relation of force to invested energy so the bite of a lion is more severe than that of a cat.
- Mutual exclusions: Since some actions cannot be executed at the same time as others, these constraints must also be documented for the developers to consider.
- Constraints: Any proprietary constraints must also be known to the developer and must therefore be contained in the description.
- Proxy: If the command deals with an opposing entity via a proxy interface, the methods and parameters should be described, so when creating new entities, the developer knows what needs to be done to support the action.

To supply this information, a template will be developed and after implementing an action it will be filled out. This way a structured catalogue of actions results which can be made available to other developers. It is clear and easily understandable, but also exhaustive because all relevant parameters are contained.

4.5 Tests

The requirements showed clearly, that those parts which are accessible to unit testing should be covered extensively to allow automated tests and avoid problems during the actual simulation execution.

The action processor and the abstract classes are completely decoupled from the actual simulator. The whole process of registering, calling and executing commands can therefore be covered by white box unit tests. Table 5 lists the test cases giving a name, a description of the operation, and the list of functions that are covered by the test case.

Table 5: Execution Processor - Test Cases

Test Case	Description	Tested functionality
MutEx	1) Call two commands of the same priority which mutually exclude each other and check if both are blocked 2) Call two commands of different priority and check if the lower is blocked and the higher is executed	Calling commands, validation via mutual exclusions with different priorities, executing commands

Inhibition	Call two different commands and inhibit one of them. After the execution phase check if the correct command was inhibited and the second one was executed.	Calling, inhibiting and executing commands
Disabling	Call two different commands, disable both and re-enable one of them. After the execution phase check if the correct command was discarded and the second one executed.	Calling, disabling and executing commands
Sequences	Create a sequence containing at least two different commands which are called in a sequence where both are executed together and also separately. Check after each round if the correct commands where dispatched and if the sequence is discarded in the end.	Calling and executing sequences

The concrete actions cannot easily be tested separated from the actual simulation, because most functions need to access the actual environment or involve interaction of several entities. Unit-tests could therefore not cover most of the predictable problems. To make this part of the project available to automated testing, a whole test-environment and a separate test simulation for each action would need to be implemented:

- A test-configuration which is as simple as possible to ensure encapsulation of possible error-sources, for example only a plane environment and one agent to test basic movement.
- A test-decision unit with a simple command list for the test.
- A logging and verification component which logs the calls, the execution result and the state of the environment after each simulation cycle. Since MASON guarantees reproducibility of results as documented in the background chapter, the log of a test-run could be compared to a previously saved successful run. If the two files differ, something has changed and must be looked into manually to determine if this is an actual error or benign.

Even if this type of automated testing would be possible, it would be sensitive to changes in other parts of the simulation too. The feature driven development process is an incremental process, and changes to other parts are not only likely, but definite. This test would therefore need a lot of future attention and work, because most differences will be benign but still need manual examination. Following this argumentation, it is not justifiable to implement automated tests for the concrete actions, so these will be tested manually. The existing simulator fortunately already offers the tools for this. A special implementation of a decision unit can be used to "remote control" the agent by using cursor keys to move it. This "remote control" must therefore be enhanced, so all new actions can also be called and tested.

5. Implementation and Results

After a review in the course of a development meeting, the model was approved by the core team. Following the feature driven development process, the whole project is now split into several features which can be implemented piece-by-piece independently and do not interfere with the development of other components. For the model at hand, these parts are the inventory class, the execution-infrastructure and the individual actions which can be added separately anyway. This chapter is also organised in this order. After giving a short overview of the existing simulator, the implementation of the inventory class will be presented. Afterwards the action processor and the related classes will be explained in detail – including sequence diagrams, examples of how to use them and how the new components were connected to the existing model. After implementing this infrastructure, the actions are developed: The implementation of a single action will be presented in detail including all the classes and code changes needed and also the documentation template will be filled out. The details on all other actions are given as a table. In the final section of this chapter, the results will be presented by crosschecking the model and the implementation to the requirements and giving the results of all the tests.

5.1 Overview of the Existing Software

The main architecture and components of the Bubble World Simulation were already introduced in the background chapter. Here some technical details of the agent's design will be presented which are relevant for the interfaces and connections to the new components.

The simulation is agent based and therefore the main architecture is comprised of two main parts: The environment and the agent (the reasons for this were given in Chapter 2). Even if certain actions need to access the environment, the classes relevant for the execution process itself deal with the agent alone, so only this part of the software will be examined here. Since the actual implementation consists of several hundred classes, only the main ones relevant for the current project are contained in Figure 14 for better oversight.

The agent's main class is the abstract `clsMobile` which is later extended for the concrete agent, depending on the simulation. The class itself is an extension of the `clsEntity` which has a reference to an instance of `clsBody`, which is also an abstract class and is extended to resemble a certain type of body. The `clsBody`-extensions then provide references to the different body-parts of the concrete entity. All actual agent's bodies are instances of the `clsComplexBody` – other entities have no decision unit and resemble things like food or stones, etc. and are not relevant in this context.

The `clsComplexBody` hosts the hierarchical references to all internal body structures:

- `clsInternalSystem`: contains references to several body-internal components like the stomach or stamina system.
- `clsIntraBodySystem`: contains references to objects concerning internal states which also have an impact on the environment or can be seen by the environment, for example body-colour or facial expressions.
- `clsInterBodyWorldSystem`: mainly provides utility classes which involve the entities' effect on the environment or vice-versa, like damage infliction, creating excrements, etc.
- `clsInternalIO`: contains references to all internal sensors or actuators like sensors which detect stamina values or health status.
- `clsExternalIO`: contains references to all external sensors or actuators such as sensors for vision or audio.
- `clsBrainSocket`: gateway class which references the actual decision unit and calls the processing function converting and passing all relevant input and output data.

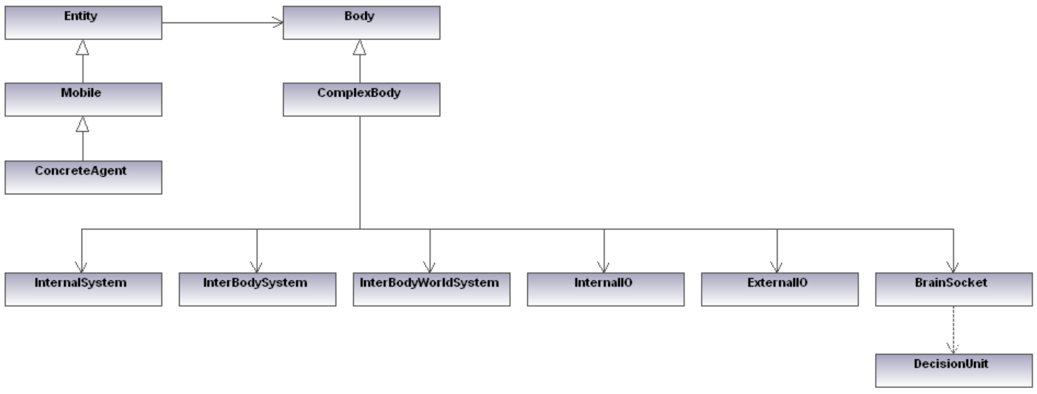


Figure 14: Simulator Class Diagram

In addition to the actual functionality of the classes, methods have to be provided so the software's infrastructure can handle them and provide some general functionality. The necessary methods are described here and are implemented in the appropriate classes without further mentioning in the text:

- Functions needed for storing and visualising the object-diagram and hierarchies:
    - `public String getName()`: returns the name of the class.
    - `public eBodyParts getBodyPartID()`: returns the type of the class referring to a general list of body parts represented by an enumeration type.
    - `public long getUniqueID()`: returns the unique ID of the instance.

- Structure for applying the configuration tree:
    - `public static clsBWProperties getDefaultProperties(String poPrefix)`: static method which returns the default configuration for the component and all it's internal components. The call is also passed on to all referenced classes, which results in a list containing all parameters as a configuration tree. Here each parameter is stored by a name=value line, the naming being a dot-separated representation of the hierarchical position in the agent's structure:

      `entitydefaults.HARE.body.sensorsext.actionavailable.ACTIONEX_MOVE=1`
    - `public clsClassName(String poPrefix, clsBWProperties poProp, references)`: the class's constructors accept the prefix and the list of properties for the configuration stored in the format above. Besides this, only relevant references to object instances are passed along side the configuration tree.
    - `private void applyProperties(String poPrefix, clsBWProperties poProp)`: private method that retrieves the values from the configuration tree passed via the parameter `poProp` and applies them (stores them in member variables or uses the values otherwise).

The naming of the new components, methods and variables is done following the project's coding conventions. For easier reading, the most important ones are given here:

- Files: classes begin with "`cls`", interfaces with "`itf`"
- Variables: The first letter defines the scope of the variable (p=parameter, m=member, s=static, blank=function), the second letter defines the type of the variable (b=boolean, e=enumeration, n=number, r=real number, o=object)
- Names are given in the camel-back notation, i.e. every new word starts with a capital letter but without spaces or blanks in-between, for example "`camelBack`" instead of "`camel_back`" or "`camelback`".

5.2 Inventory

Actions are stateless and will be discarded after execution, so any action which needs to keep a persistent state must do this by accessing components bound to the entity. The actions pick up, drop and move from/to inventory bind other entities to the agent in a persistent manner. Since the existing implementation does not offer a way to handle this, a new class – the inventory class – needs to be created and implemented.

Section 4.3.2 already described all the necessary functions and all of them were implemented in a single class. This new component does not fit directly into the existing tree of body-parts, but since it binds entities to one another, it was decided to reference it through the `clsMobile` as shown in Figure 15.

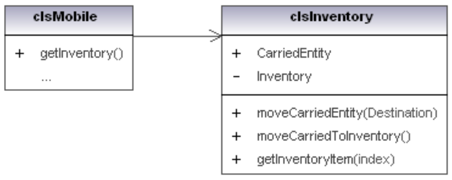

clsMobile
+ getInventory()
...
clsInventory
+ CarriedEntity
- Inventory
+ moveCarriedEntity(Destination)
+ moveCarriedToInventory()
+ getInventoryItem(index)

Figure 15: Inventory Class Diagram

5.3 Classes for Command Execution

The classes needed for the command execution infrastructure were developed in Section 4.2. Figure 16 shows an overview of these classes and a detailed description for each element is given in this section.

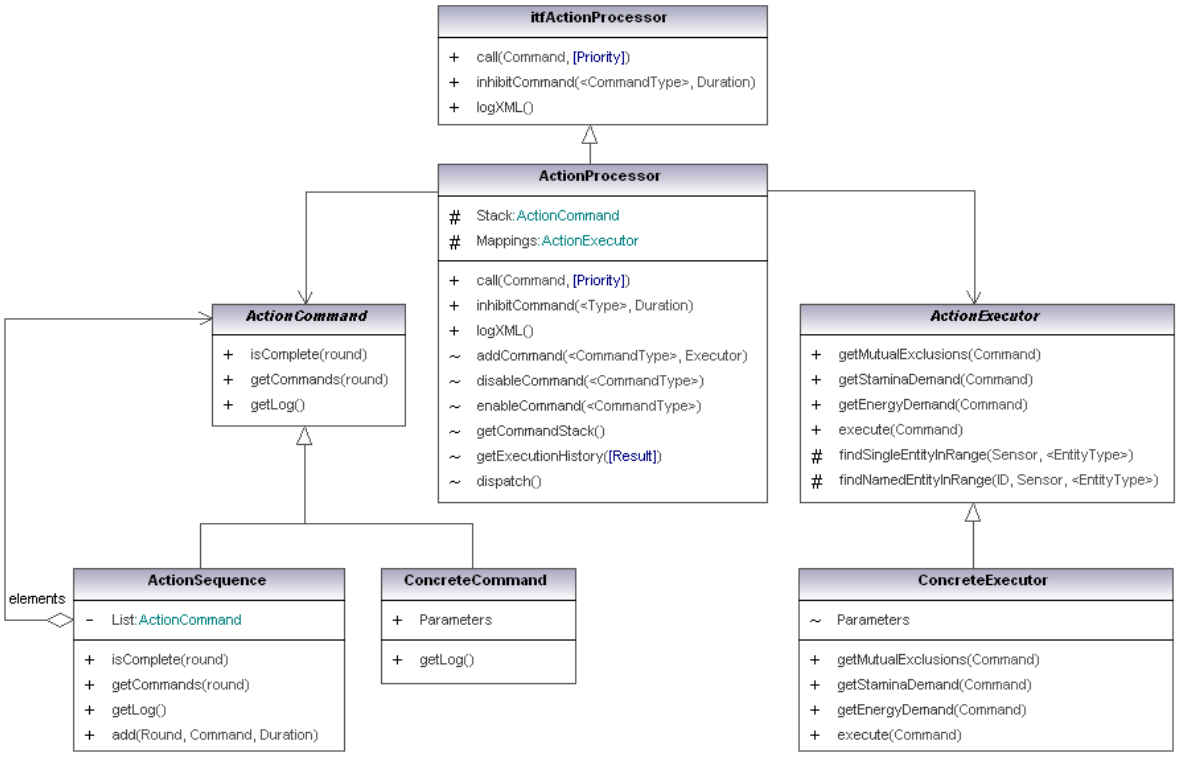

itfActionProcessor
+ call(Command, [Priority])
+ inhibitCommand(<CommandType>, Duration)
+ logXML()
ActionProcessor
# Stack:ActionCommand
# Mappings:ActionExecutor
+ call(Command, [Priority])
+ inhibitCommand(<Type>, Duration)
+ logXML()
~ addCommand(<CommandType>, Executor)
~ disableCommand(<CommandType>)
~ enableCommand(<CommandType>)
~ getCommandStack()
~ getExecutionHistory([Result])
~ dispatch()
ActionCommand
+ isComplete(round)
+ getCommands(round)
+ getLog()
ActionExecutor
+ getMutualExclusions(Command)
+ getStaminaDemand(Command)
+ getEnergyDemand(Command)
+ execute(Command)
# findSingleEntityInRange(Sensor, <EntityType>)
# findNamedEntityInRange(ID, Sensor, <EntityType>)
elements
ActionSequence
- List:ActionCommand
+ isComplete(round)
+ getCommands(round)
+ getLog()
+ add(Round, Command, Duration)
ConcreteCommand
+ Parameters
+ getLog()
ConcreteExecutor
~ Parameters
+ getMutualExclusions(Command)
+ getStaminaDemand(Command)
+ getEnergyDemand(Command)
+ execute(Command)

Figure 16: Implementation Class Diagram

5.3.1 Abstract Action Command Class “clsActionCommand”

The `clsActionCommand` is an abstract class which is later extended by the concrete commands and is implemented in the decision unit interface package. The class’s interface is designed following the Composite pattern to hide the internal structure and contains following methods:

- `boolean isComplete(int pnRound)`: returns if the command is completed, i.e. in the case of a composite command this means that all items have been executed. The default behaviour is a single-round command, so the method returns true when called with `pnRound` greater than 0.
- `ArrayList<clsActionCommand> getCommands(int pnRound)`: returns the list of individual commands for the current round. Default behaviour is again the single-round, single-instance command, so the method returns a list containing only itself.
- `abstract String getLog()`: returns the logging information for the command as an XML node with the format: `<CommandName>parameterlist</CommandName>`

5.3.2 Action Sequence Class "clsActionSequence"

This class is also implemented in the decision unit interface and is an extension of the `clsActionCommand`. It hides it's list structure to the processor through the interface, but supplies a method for adding new commands with it's extended interface:

- `add(int pnRound, clsActionCommand poAction, int pnDuration)`: adds the new command `poAction` to the sequence for a duration of `pnDuration` beginning in round `pnRound`, where `0` is the current round.
- `String getLog()`: overrides the method of the base-class and returns the concatenated string of all the commands' logging information by looping through it's command list.

5.3.3 Abstract Action Executor Class "clsActionExecutor"

The `clsActionExecutor` is an abstract class which is later extended by the concrete executors and is implemented in the main simulator package. It offers all functions needed by the action processor as was defined in the model:

- `ArrayList<Class<?>> getMutualExclusions(clsActionCommand poCommand)`: arraylist containing the types of action commands which cannot be performed at the same time. This will be checked by the processor in a double loop prior to execution. Commands of the same type automatically exclude themselves and need not be listed, i.e. no two commands of the same type can be executed in the same round. Default behaviour is to return an empty list.
- `abstract double getStaminaDemand(clsActionCommand poCommand)`: gets the amount of stamina needed to perform the action. Even if the action cannot be performed due to some constraint, this amount of stamina will be consumed.
- `double getEnergyDemand(clsActionCommand poCommand)`: gets the amount of energy needed per round to perform the action. Even if the action cannot be performed, this amount of energy will be consumed. Default behaviour is to return the amount of stamina scaled by the constant value `0.2`, i.e. a stamina value of `0.5` will result in an energy demand of `0.1`.

- `abstract boolean execute(clsActionCommand poCommand)`: executes the actual command and returns true if the command was successful or false otherwise.

Additionally, the class supplies utility functions as protected methods that are needed by several concrete executors:

- `clsEntity findSingleEntityInRange(clsEntity poSelfReference, clsComplexBody poBody, eSensorExtType peSensor, Class<?> poInterface)`: returns a reference to an entity that implements the `poInterface` if exactly one can be found in the range defined by `peSensor`. For example, this method is used by the eat action to find something "eatable" in range by passing the "eatable area sensor" and the `itfEatable` interface.
- `clsEntity findNamedEntityInRange(String EntityID, clsComplexBody poBody, eSensorExtType peSensor, Class<?> poInterface)`: returns a reference to an entity with `ID=EntityID`, if it carries the `poInterface` and can be found in the range of `peSensor`.

5.3.4 Action Processor Class "clsActionProcessor"

The calling interface of this class is published to the decision unit by implementing an `itfActionProcessor` interface which also decouples the two packages following the proxy pattern. It is implemented in the decision unit interface and supplies following methods:

- `void call(clsActionCommand poCommand, eCallPriority pePriority)`: calls a command `poCommand` and adds it to the execution stack with `pePriority` (reflex, body-update, or normal). The command can be any concrete implementation extending the abstract `clsActionCommand` including composites like sequences.
- `void call(clsActionCommand poCommand)`: calls the command using the default priority "normal".
- `void inhibitCommand(Class<?> poCommand, int pnDuration)`: inhibits commands of the type `poCommand` for `pnDuration` rounds.
- `public String logXML()`: returns the complete log of all commands called during the current execution round by returning the concatenated string of all the commands `getLog()` return values.

The concrete `clsActionProcessor` is implemented in the main simulator package and supplies following additional functions described in the configuration and execution interface of the model:

- `void addCommand(Class<?> poCommand, clsActionExecutor poExecutor)`: maps the command `poCommand` to the executor `poExecutor`. This method will be called during the set-up of the simulator. All commands must be registered in this manner, so the executor for a concrete command can be determined and validation and execution can take place.

- `void disableCommand(Class<?> poCommand)`: disables all commands of the type `poCommand` forthwith. Calls can be performed, but during execution phase the commands will be dropped without further notice.
- `void enableCommand(Class<?> poCommand)`: re-enables commands of the type `poCommand`.
- `ArrayList<clsActionCommand> getCommandStack()`: returns the list of commands currently on the stack. This method is for debug purposes only.
- `ArrayList<clsActionCommand> getExecutionHistory(eExecutionResult peResult)`: returns an array list of all commands that were on the stack during the previous round and where called resulting in `peResult` (execution or non execution due to constraint violation, disabling, inhibition, mutual exclusion, no stamina). This method is also for debug purposes only.
- `ArrayList<clsActionCommand> getExecutionHistory()`: returns all actually executed commands using the previous method with `peResult=eExecutionResult.Executed`. This method is also for debug purposes only.
- `void dispatch()`: Determines, validates and then executes all the commands on the stack as described in detail in the model.

The action processor is instantiated by – and can be referenced through – the `clsExternalIO` since this class is responsible for managing actuators to the agent's external world. Some actions actually don't relate to the environment, but to internal body states such as body colour or facial expressions. It would be possible to duplicate the execution infrastructure and implement a second processor in the `clsInternalIO` to differentiate between them. The action processor class could be re-used by instantiating it twice and actions would be registered at only one of the two instances. But there is no real benefit in doing this, because the developer of a decision unit would always have to be careful to dispatch a command to the correct processor. But since the commands themselves are instantiated without reference to an internal or external hierarchy, no compiler error or other warning would bring this to the attention of the developer and the simulation would just crash during runtime. Having only a single processor is therefore the easiest way to overcome this, even if the initial design seems to have anticipated a different implementation. But since the decision unit gets all sensor data in a single container instead of an internal and external one, it may also be that the differentiation of internal and external sensors will be given up later too.

To test the basic functionality of the simulator up to the point where the real actions are implemented, the development team integrated some simple temporary methods for movement. The development of the actual infrastructure and the new implementation of the actions can therefore be done without interfering with the other components. Following the feature driven development process, components of other developers are not interfered with due to code ownership. When all necessary work is done and the tests are completed, the old functions can be marked as deprecated and be replaced with the new ones by the individual developers. In practice this is not done in the current project because only two simple test-classes actually used the temporary methods. So after everything is prepared and tested, and after coordination with the responsible team members, the temporary implementation are removed and replaced right away.

5.3.5 Interface to the Decision Unit

All relevant classes and interfaces are implemented in the decision unit interface, so they can be referenced by all decision units without also referencing the main simulator. The processing phase of the cycle is initiated by the `clsBrainSocket` described in Section 5.1. This gateway class transforms all internal sensory data structures to the appropriate types known to the decision unit and also passes a reference to the action processor using the `itfActionProcessor` interface known to both packages. The decision unit can then create a command and call it as shown in following short example which calls the move command with the parameters for forward, and the default speed of 1:

```
oActionProcessor.call(new clsActionMove(eActionMoveDirection.MOVE_FORWARD,1.0));
```

5.4 Actions

After the execution infrastructure is implemented, the concrete actions can be developed piece-by-piece extending the functionality of the simulator without interfering with other development work as requested by the feature driven process. In this section, first a detailed example for implementing an action will be given. All other atomic actions will be described in a table enhancing the table created in the model. Some actions are not atomic and were implemented as sequences of other actions which can be created using a factory class as described in the final subsequence.

5.4.1 Implementing a new Action

An action can consist of up to three separate structures:

- Command: The command is a class extending the abstract `clsActionCommand` and implemented in the decision unit interface. It contains all of the parameters that can be set for a concrete command and is passed to the action processor upon calling and stored there on a stack.
- Executor: The executor is a class extending the abstract `clsActionExecutor` and implemented in the main simulator package. It is used by the action processor to validate and execute a command. An executor can handle several commands, but to encapsulate them and keep them independent only different variations of the same command should be combined in this way, e.g. change body-colour, change red component of the body-colour, change blue component of the body-colour, change green component of the body-colour.
- Proxy: When a command deals with an opposite entity – for example when attacking it – a proxy-interface is used to decouple the concrete entity from the action. The proxy is used both to search for an entity which can be used for the action and also to perform the action

itself. Not all commands need a proxy, the "change body-colour", for example, can directly access it's own entities' settings without needing any additional decoupling.

The implementation will be described in detail using the "attack/bite" action as an example because it is simple but still needs all three structures. The implementation will be done by creating the necessary components following the model, testing the implementation manually through the simulator's remote control and then documenting it using the template designed in the model. The single steps will show listings for the relevant parts of code matching. The complete, and combined source code for this command can be found in Appendix III.

Command class

Listing 1 shows the code for the concrete action command. Following the naming conventions, the class is called `clsActionAttackBite` and extends the `clsActionCommand`. The file is stored in the folder `decisionunit.itf.actions` of the decision unit interface.

Following the model, the only parameter necessary is the force of the attack, so this value will be stored in a member variable `mrForce` (double types are used rather than float to avoid conversions, since the physics engine uses doubles). For this parameter, get and set methods are implemented and the parameter is also added to the constructor to allow the complete creation and call of a command in a single line rather than having to set every property individually using the setters.

The `getLog()` method of the base class is overridden and returns the command's name and the force-parameter as an XML node to the caller. The methods `isComplete` and `getCommands` can be kept and need not be overridden, because the default behaviour of the abstract command is already that of an atomic command.

Listing 1: Concrete Action Command "clsActionAttackBite"

```
public class clsActionAttackBite extends clsActionCommand {
   private double mrForce;
   public clsActionAttackBite(double prForce) {
      mrForce=prForce;
   }
   @Override
   public String getLog() {
      return "<AttackBite>" + mrForce + "</AttackBite>";
   }
   public double getForce() {
      return mrForce;
   }
   public void setForce(double prForce) {
      mrForce=prForce;
   }
}
```

Proxy class

Following the model, the command uses an interface for the opposing entity. It is named `itfAPAttackableBite` and stored in the `bw.body.io.actuators.actionProxies` of the main simulator. The interface contains all methods needed by the executor.

Listing 2: Proxy Interface "itfAPAttackableBite"

```
public interface itfAPAttackableBite {
   double tryBite(double pfForce);
   void bite(double pfForce);
}
```

Listing 2 shows the source code for the `itfAPAttackable` interface which defines following methods:

- `double tryBite(double pfForce)`: This method will be called before the actual action. The opposing entity can determine if the force of the bite is high enough to actually damage the attacked entity rather than the biting entity. A zero value will be returned if ok, or otherwise a positive value representing the damage done to the calling entity.
- `void bite(double pfForce)`: This method will only be called after tryBite returned zero. The opposing entity must do all appropriate steps like register the damage, remove itself from the simulation, etc.

Executor class

The executor is named `clsExecutorAttackBite` and extends the `clsActionExecutor`. It is stored in the `bw.body.io.actuators.actionExecutors` folder of the main simulator.

The basic parts of this class are shown in Listing 3. Due to the configuration logic of the system, the constructor takes the prefix and configuration tree as parameters. The `applyProperties` method extracts the actual configuration values and stores the values in member variables. The defaults for are returned by the `getDefaultProperties` method as described in Section 5.1. For the Attack/Bite command, following configuration values are defined:

- A range sensor which returns all entities near enough to perform the action which is stored in the member variable `moRangeSensor` and defaulted to `eSensorExtType.EATABLE_AREA`.
- A scaling factor to set the ratio of energy to force (to distinguish between a cat's bite and a lion's bite) which is stored in the member variable `mrForceScalingFactor` and defaulted to `1`.

The concrete action additionally needs a reference to the agent itself to access the sensors, so this is also passed via the constructor and stored in the `moEntity` variable.

The constructor also creates the array-list of mutual exclusions and stores it in the member variable `moMutEx` which can be accessed through the overridden method `getMutualExclusions`. In this case the only command that mutually excludes the Attack/Bite command is the Eat command, so this command class type is added to the array-list. The construction of the array-list could be done in the getter itself, but because the validation for this is done in a double loop and the method will be called several times. For performance reasons it is therefore prudent to set up the list once and pass as a reference for each call.

For the infrastructure of the simulator, the executor then also needs to override methods for the name and body-part ID of the class. The latter returns an enumeration value that needs to added to the enumeration `bw.utils.enums.eBodyParts`.

Listing 3: Basic Parts of the Concrete Action Executor "clsExecutorAttackBite"

```
public class clsExecutorAttackBite extends clsActionExecutor{
   private double mrForceScalingFactor; //Set by the applyProperties method
   private ArrayList<Class<?>> moMutEx = new ArrayList<Class<?>>(); //Set by the
     constructor
   private clsEntity moEntity; //passed in the constructor
   private eSensorExtType moRangeSensor; //Set by the applyProperties method
   public static final String P_RANGESENSOR = "rangesensor";
   public static final String P_FORCECALINGFACTOR = "forcescalingfactor";
   public clsExecutorAttackBite(String poPrefix, clsBWProperties poProp, clsEntity
     poEntity) {
      moEntity=poEntity;
      moMutEx.add(clsActionEat.class);
      applyProperties(poPrefix,poProp);
   }
   public static clsBWProperties getDefaultProperties(String poPrefix) {
      String pre = clsBWProperties.addDot(poPrefix);
      clsBWProperties oProp = new clsBWProperties();
      oProp.setProperty(pre+P_RANGESENSOR, eSensorExtType.EATABLE_AREA.toString());
      oProp.setProperty(pre+P_FORCECALINGFACTOR, 1f);
      return oProp;
   }
   private void applyProperties(String poPrefix, clsBWProperties poProp) {
      String pre = clsBWProperties.addDot(poPrefix);
      moRangeSensor=eSensorExtType.
     valueOf(poProp.getPropertyString(pre+P_RANGESENSOR));
      mrForceScalingFactor=poProp.getPropertyFloat(pre+P_FORCECALINGFACTOR);
   }
   @Override
   protected void setBodyPartId() {
      mePartId = bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
   }
   @Override
   protected void setName() {
      moName="Attack/Bite executor";
   }
   @Override
   public ArrayList<Class<?>> getMutualExclusions(clsActionCommand poCommand) {
      return moMutEx;
   }
}
```

After having set-up the part of the class involved with the general infrastructure, the more action specific methods are added. The `getEnergyDemand` method is used in its default behaviour returning 20% of the calculated stamina. The `getStaminaDemand` method is overridden and returns an exponentially rising value depending on the force defined in the command. The parameters of the formula are stored in static variables and are not configured by the configuration tree because they have to be fine-tuned to balance out regeneration and consummation since they grow exponentially and should not be changed for different entity types. Listing 4 shows the code-snippet for this method including the static variables.

Listing 4: Attribute Functions of Concrete Action Executor "clsExecutorAttackBite"

```
static double srStaminaScalingFactor = 0.001;
static double srStaminaBase = 4f;
   @Override
   public double getStaminaDemand(clsActionCommand poCommand) {
      clsActionAttackBite oCommand =(clsActionAttackBite) poCommand;
      return srStaminaScalingFactor* Math.pow(srStaminaBase,oCommand.getForce()) ;
   }
```

Listing 5 shows the execution method, which is overridden to perform the actual operation. First an opponent is located using the utility-function `findSingleEntityInRange` of the executor. If none can be found the command is aborted, and a notification is sent to the brain using the Fast Messenger system. This needs to be done because actions do not provide direct feed back and indirect feed-back through sensors is not available here but must therefore be injected in this way.

If an entity was found, the reference is accessed via the `itfAPAttackableBite` interface. First the `tryBite` method is called which either returns a positive value which is interpreted as a damage to the caller and thus passed on to the health system, or zero which means the attack can be performed. The force defined in the command is herby scaled using the scaling factor from the configuration so an attack which needs the same percentage of the entity's energy and stamina represents a different force depending on the type of entity.

Listing 5: Execute Method of Concrete Action Executor "clsExecutorAttackBite"

```
@Override
public boolean execute(clsActionCommand poCommand) {
clsActionAttackBite oCommand =(clsActionAttackBite) poCommand;
clsComplexBody oBody = (clsComplexBody) ((itfGetBody)moEntity).getBody();
itfAPAttackableBite oOpponent = (itfAPAttackableBite)
    findSingleEntityInRange(moEntity, oBody, moRangeSensor,
    itfAPAttackableBite.class) ;
if (oOpponent==null) {
   clsFastMessengerSystem oFastMessengerSystem
    =oBody.getInternalSystem().getFastMessengerSystem();
   oFastMessengerSystem.addMessage(mePartId, eBodyParts.BRAIN, 1);
   return false;
}
double rDamage = oOpponent.tryBite(oCommand.getForce()*mrForceScalingFactor);
if (rDamage>0) {
   oBody.getInternalSystem().getHealthSystem().hurt(rDamage);
   return false;
}
oOpponent.bite(oCommand.getForce()*mrForceScalingFactor);
   return true;
}
```

Registering the command

Before using the command, it must be referenced at the processor and mapped to an appropriate executor. This is done during the set-up of the action processor in `bw.body.io.clsExternalIO`.

First the default property-tree is enhanced by adding a property to define if an action should be available. The list is also passed on to the executor so it can add it's default properties. The prefixes are created by combining fixed prefixes and a string representation of the body-part enumeration value of the executor. Following the code conventions, all fixed values – here the prefixes – are saved in static variables. Listing 6 shows the lines added to the `getDefaultProperties` method of `clsExternalIO` and the static variable definition.

Listing 6: Code-Snippets for Registering the Command AttackBite in the getDefaultProperties method

```
public static final String P_ACTIONAVAILABLE = "actionavailable";
public static final String P_ACTIONEX = "actionexecutor";

sKey= pre+P_ACTIONAVAILABLE+"."+bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
oProp.setProperty(sKey,1);
sKey= pre+P_ACTIONEX+"."+bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
oProp.putAll(clsExecutorAttackBite.getDefaultProperties(sKey));
```

The actual values are set in the `applyProperties` method. If – according to the configuration – the action is enabled, it will be registered at the processor by creating an instance of the executor and passing it to the processor together with the command's type. In Listing 7 the source-code for this is given.

Listing 7: Code-Snippet for Applying Properties

```
sKey= pre+P_ACTIONAVAILABLE+"."+bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
if (oProp.getPropertyInt(sKey)==1) {
   sKey=poPrefix +"." +P_ACTIONEX +"." +bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
   oExecutor = new clsExecutorAttackBite(sKey,poProp,moEntity)
   moProcessor.addCommand(clsActionAttackBite.class, oExecutor);
}
```

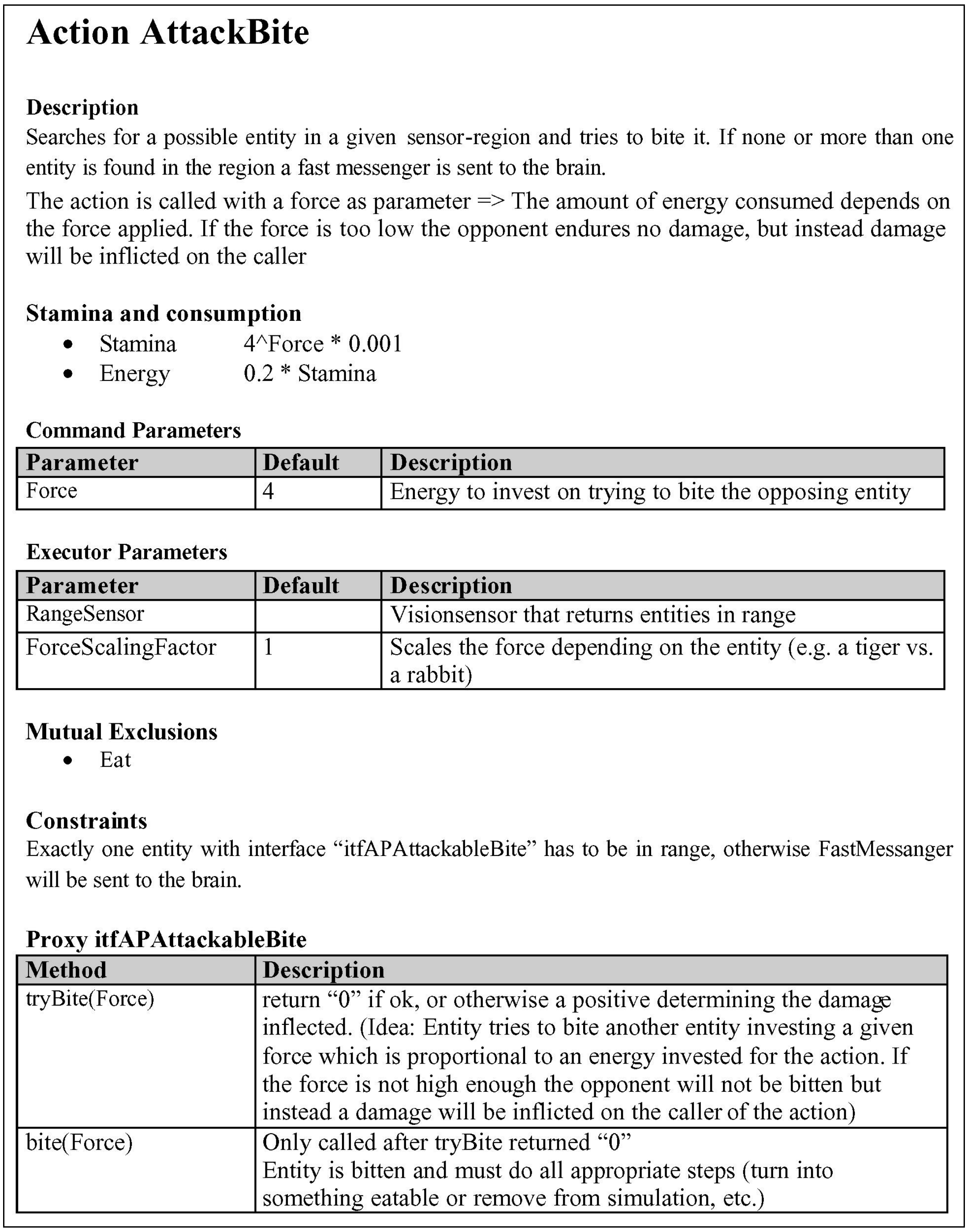
Documenting the command

After the command is implemented and all parameters are known it can be documented for other developers. For this, a general documentation template for actions was defined in Section 4.4 and is used consistently for all actions, so a general catalogue of actions can be produced. The template contains all the parameters which were already defined in the model:

- Description of the behaviour
- Stamina and energy demand
- Command parameters: name, default value and description
- Executor parameters: name, default value and description

- The list of mutual exclusions
- Constraints
- Proxy: name and list of methods including their description

For the example command at hand, the result is shown in Figure 17. The forms for all other implemented actions can be found in Appendix II.



Action AttackBite

Description

Searches for a possible entity in a given sensor-region and tries to bite it. If none or more than one entity is found in the region a fast messenger is sent to the brain.

The action is called with a force as parameter => The amount of energy consumed depends on the force applied. If the force is too low the opponent endures no damage, but instead damage will be inflicted on the caller

Stamina and consumption

• Stamina	4^Force * 0.001
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Force	4	Energy to invest on trying to bite the opposing entity

Executor Parameters

Parameter	Default	Description
RangeSensor		Visionsensor that returns entities in range
ForceScalingFactor	1	Scales the force depending on the entity (e.g. a tiger vs. a rabbit)

Mutual Exclusions

- Eat

Constraints

Exactly one entity with interface “itfAPAttackableBite” has to be in range, otherwise FastMessanger will be sent to the brain.

Proxy itfAPAttackableBite

Method	Description
tryBite(Force)	return “0” if ok, or otherwise a positive determining the damage inflected. (Idea: Entity tries to bite another entity investing a given force which is proportional to an energy invested for the action. If the force is not high enough the opponent will not be bitten but instead a damage will be inflicted on the caller of the action)
bite(Force)	Only called after tryBite returned “0” Entity is bitten and must do all appropriate steps (turn into something eatable or remove from simulation, etc.)

Figure 17: Filled out documentation template for the Attack/Bite action

Implementing the Proxy Interface

After the command is registered, the proxy interface has to be implemented by the code-owners of the entity-classes. They must decide which entities can be used for the function – in this case which can be bitten – and implement the interface following the documentation in the command's description. The interface `itfAttackableBite` is implanted by the classes `clsHare` and `clsCarrot`, which are extension of `clsMobile`. Therefore, these two types of entities are found by the executor and can be "bitten" by calling the Attack/Bite command.

Testing the Command

As argued in the model, the tests are performed manually by using a special simulation set-up. In this configuration a remote-control agent is available whose decision unit is a simple loop querying the keyboard and calling a function depending on the key the user presses. The `clsRemoteControl` from `decisionunits.simple.remotecontrol` must first be enhanced for the new command. To do this, a switch must be added for an unused key in the method `process`. For the Attack/Bite command the key "b" is chosen which has the integer representation of 66. When the key is pressed, the action command is created and passed to the processor with the default force of 4 as shown in Listing 8.

Listing 8: Code-Snippet Adding a New Command to the RemoteControl

```
case 66: //'B'
   poActionProcessor.call(new clsActionAttackBite(4));
   break;
```

After adding the new command to the remote control, the simulation can be started and tested by moving the remote controlled agent to an opposing entity that implements the `itfAttackableBite` interface and pressing "b" to bite it. The visualisation of the simulation and also the inspectors supply the necessary information to determine if the command was executed appropriately.

5.4.2 Implemented Actions

The commands, executors and proxies for the remaining actions are implemented in the same way as the example above. Table 4 lists the actions, executors, and proxies that have to be implemented. Table 6 shows all classes that were actually created for each of the actions, Details can be seen in Appendix II:

- For each command-class the parameters are listed
- For each executor-class the parameters and a short description of the functionality is given
- For proxy-interfaces the methods are stated.

Table 6: Implemented Actions

Action	**Type**	**Class Name**	**Description**
Bite	Command	`clsActionAttackBite`	*Parameter:* `force`
	Proxy	`itfAPAttackableBite`	*Methods:* `tryBite(force)=0` for OK `>0` for damage `bite(force)`
	Executor	`clsExecutorAttackBite`	*Parameters:* `Scaling factor`, `Range` *Execute:* Search for entity, try, bite
Change body colour	Command	`clsActionBodyColor`	*Parameters:* `red`, `green`, `blue`
	Command	`clsActionBodyColorBlue`	*Parameter:* `blue`
	Command	`clsActionBodyColorRed`	*Parameter:* `red`
	Command	`clsActionBodyColorGreen`	*Parameter:* `green`
	Executor	`clsExecutorBodyColor`	*Execute:* set values via the intra body system
Change facial expression	Command	`clsActionFacialExEyeSize`	*Parameter:* `size (enum)`
	Command	`clsActionFacialExLeftAntenna Position`	*Parameter:* `position (enum)`
	Command	`clsActionFacialExLensShape`	*Parameter:* `shape (enum)`
	Command	`clsActionFacialExLensSize`	*Parameter:* `size (enum)`
	Command	`clsActionFacialExRightAntenn aPosition`	*Parameter:* `position (enum)`
	Executor	`clsExecutorFacialExpressions`	*Parameter:* `Range` *Execute:* set values via the intra body system
Cultivate	Command	`clsActionCultivate`	*Parameter:* `amount`
	Proxy	`itfAPCultivatable`	*Method:* `cultivate(amount)`
	Executor	`clsExecutorCultivate`	*Execute:* Search for entity, cultivate
Eat	Command	`clsActionEat`	*Parameters*: `range, bitesize`
	Proxy	`itfAPEatable`	*Methods:* `tryEat()=0` for OK `>0` for damage `eat(biteSize)`
	Executor	`clsExecutorEat`	*Execute:* Search for entity, try, eat
Excrement	Command	`clsActionExcrement`	*Parameter:* `Intensity`
	Executor	`clsExecutorExcrement`	*Parameters:* `Scalingfactor` *Execute:* Create and drop excrement via inter body world system

Kiss	Command	`clsActionKiss`	*Parameter:* `Intensity (enum)` Execute: Search for entity, try, kiss
	Proxy	`itfAPKissable`	*Methods:* `tryKiss(intensity)` `kiss(intensity)`
	Executor	`clsExecutorKiss`	*Execute:* Search for entity, try, kiss
Lightning-attack	Command	`clsActionAttackLightning`	*Parameters:* `Force`, `OpponentID`
	Proxy	`itfAPAttackableLightning`	*Method:* `attack(force)`
	Executor	`clsExecutorAttackLightning`	*Parameters:* `Scalingfactor` *Execute:* Search for entity, attack
Move to/from inventory, Pickup, Drop, Move to eatable area	Command	`clsActionPickUp`	No Parameters
	Command	`clsActionDrop`	No Parameters
	Command	`clsActionToInventory`	No Parameters
	Command	`clsActionFromInventory`	*Parameter:* `Index`
	Command	`clsActionMoveToEatableArea`	No Parameters
	Proxy	`itfAPCarryable`	*Methods:* `getCarryableEntity` `setCarried-BindingState(State)`
	Executor	`clsExecutorPickUp`	*Parameters:* `Range`, `Scalingfactor` *Execute:* Set binding via inventory
	Executor	`clsExecutorDrop`	*Execute:* Set binding via inventory
	Executor	`clsExecutorToInventory`	*Execute:* Set binding via inventory
	Executor	`clsExecutorFromInventory`	*Execute:* Set binding via inventory
	Executor	`clsExecutorMoveToArea`	*Parameters:* `Destination Area` *Execute:* Change binding joint to add force pulling entity to eatable area via inventory
Move, turn	Command	`clsActionMove`	*Parameters:* `Direction (enum)`, `speed`
	Command	`clsActionTurn`	*Parameters:* `Direction (enum)`, `angle`
	Executor	`clsExecutorMove`	*Parameter:* `Scalingfactor` *Execute:* Add movement force
	Executor	`clsExecutorTurn`	*Execute:* Add force
Sleep	Command	`clsActionSleep`	*Parameters:* `Intensity (enum)`
	Proxy	`itfAPSleep`	*Method:* `Sleep()`
	Executor	`clsExecutorSleep`	*Parameter:* List of components to notify for light/intense sleep *Execute:* Notify components via proxy

5.4.3 Implemented Actions Sequences

The requirements specify “dancing” as a necessary action. As stated in the model, this action is not atomic and is therefore represented by an action sequence. To prepare and re-use these common action sequences, a factory class was added to the decision unit interface.

This `clsActionSequenceFactory` supplies static functions which create and return commonly used action sequences so they can be passed to the processor. In Table 7 the requested dance styles are listed including the method-name in `clsActionSequenceFactory` for creating the sequence and a description listing the atomic commands used.

Table 7: Action Sequences

Sequence	Method	Description
Salsa	`getSalsaSequence(Speed, Duration)`	Move in squares for <duration> times Round Command 1-15 `move (forward, <Speed>)` 20-30 `turn (right, 18)` 30-45 `move (forward, <Speed>)` 50-60 `turn (right, 18)` 60-75 `move (forward, <Speed>)` 80-90 `turn (right, 18)` 90-105 `move (forward, <Speed>)` 110-120 `turn (right, 18)`
Walz	`getWalzSequence(Speed, Duration)`	Move in circles for <duration> times Round Command 1-120 `move (forward, 2)` 1-60 `turn (left, 12*<Speed>)` 61-120 `turn (right, 12*<Speed>)`
Tango	`GetTangoSequence(Speed, Duration)`	Move in triangles for <duration> times Round Command 1-15 `move (forward, <Speed>)` 20-30 `turn (right, 24)` 30-45 `move (forward, <Speed>)` 50-60 `turn (right, 24)` 60-75 `move (forward, <Speed>)` 80-90 `turn (right, 24)`

5.5 Results

To validate the results of the project, the model will be crosschecked with the requirements making sure that everything is covered. Then the test-results for the infrastructure, hands-on tests for the actions and real simulation results will be given. To summarize the results a table with the state of all components is then presented in the last subsection.

5.5.1 Crosschecking Model and Requirements

The model defines the classes of the execution infrastructure and the basic strategy for implementing actions. To ensure that the requirements are met, a comparison is done.

For the actions a basic strategy and logic is applied that corresponds to the command pattern. The structure ensures that the list can be expanded later without having to change the existing code. For each of the actions in Table 3, the necessary command-, executor- and proxy classes are defined. The actual implementation can be done for each action separately in an order following the project's priorities.

The first demand stated in the requirements is, that it needs to be possible to enable and disable commands. This is solved by supplying appropriate functions in the command processor which can be called during set-up and also later during the simulation. The executors can be configured and registered at the command processor so the behaviour of an action can be adapted depending on the entity as requested.

Both actions and sequences can be called or inhibited during any phase of the simulation and the information is stored until the execution phase by applying the command and command processor patterns. The actions and also the calling interface of the execution processor are made available to the decision unit by implementing them in the decision unit interface package.

Subsection 3.3 demands that the commands shall be validated by checking energy demands, mutual exclusion and also additional individual constraints. This was solved by implementing all general checks in the action processor and all the individual ones directly in the executor classes. This way the action processor stays independent and the executors are kept free from redundant checking algorithms.

For the execution, Subsection 3.3 already suggest to use interfaces to identify entities and call the relevant methods. This was implemented following the proxy pattern and defining the list of classes needed by enhancing the table of actions with the command, executor, and proxy classes.

To manage the binding states an inventory class was defined which offers all the functionality needed to support the actual commands.

To ensure the extensibility of the components requested in Section 3.4 several measures were taken:

- The Proxy pattern is applied to decouple the executor from the entities. This way new entities can be created without having to change existing code.

- The Command pattern is used to decouple the command from the execution infrastructure. The commands are also independent of one another. New actions can be defined by creating a command and an executor and registering them at the processor – again no existing code needs to be changed.
- The Compound pattern is used so sequences and single actions supply the same interface. This can be used to develop new structures of commands encapsulating the structure and hiding it from the processor

Section 3.4 also describes the requirements concerning documentation. To fulfil these, a well-structured, single template that describes all actions is available and provides all relevant information for developers who need to call actions or develop new entities.

To enhance testability and promote robustness as requested, the action processor supplies additional logging functionality so test cases can be developed which automatically compare the actual result to the intended one. Additional the Proxy pattern allows to test commands and executors with unit tests, because the actual entity can be replaced with a stub without further changes to the components.

The performance could be estimated by calculating the amount of computation time that is needed when calling and executing commands. But since performance is only a second level priority and no concrete measurable values were given in Section 3.4, this examination is not undertaken.

Actions are atomic and stateless as defined, and calling them returns no feedback. The execution-result is available for testing and debugging only and is hidden from the decision unit because the methods are not published in the decision unit interface.

The Proxy pattern used to decouple the executor and the entities can also be used to identify and find applicable objects. As requested, it is not necessary to pass the entity as a parameter to the command because it can be found automatically.

The binding constraints defined in Section 3.5 are treated similarly to individual constraints of other actions and are checked by the executor or respectively the utility functions of the inventory class.

This crosscheck shows that all requirements are respected by the model and precautions have been taken for the non-functional ones. In accordance with the feature driven development process, the model was presented to the core development team by supplying a documentation of the proposed model and discussing it during a meeting prior to the implementation.

5.5.2 Unit-Testing the Execution Infrastructure

The test cases for the execution infrastructure are defined in Subsection 4.5. The development environment offers the JUnit Test Suite as mentioned in Chapter 2.3.1, so this framework is used to implement the test cases. Before creating the actual test, some utility classes need to be created:

- `tstTestCommand`: A concrete command extending the `clsActionCommand`. The command provides getters and setters for energy and stamina and an execution state flag. The command also hosts a mutual exclusion list which can be manipulated externally. This way a test can inject properties it wants to test without having to create separate commands.

- `tstTestCommand_A`, `tstTestCommand_B`: These two classes extend the `tstTestCommand` class, but do not change the actual behaviour of the class. The distinction is only necessary so two commands A+B can be called simultaneously as necessary according to the test cases (Calling the same command twice for the same round is not possible due to automatic mutual exclusion)
- `tstTestExecutor`: This is a concrete action executor extending the `clsActionExecutor`. The list of mutual exclusions is passed from the command and the execution function only sets the "executed" flag of the `tstTestCommand`. This way, by keeping a reference to the command, the test function can determine if the execute-function was actually called.

All test cases begin by creating an action processor and registering the `tstTextCommand_A` and `tstTextCommand_B` with the `tstTestExecutor`. Table 8 shows the implementation of the test cases. The names are take from Table 5 and the column "implementation" describes the actual realisation on the basis of the description in the model.

Table 8: Test Case Implementation

Test Case	Implementation
MutEx	1) Calling two commands with the same priority * Create new commands A and B of the type `tstTestCommand_A`, respectively `tstTestCommand_B` and call them both with priority normal. * Set the mutual exclusion of command B when calling command A * Call the action-processor's dispatch function * Assert that both commands were blocked by checking both the commands execution flag and also the execution history of the processor. 2) Calling two commands with different priority * Create new commands A and B of the type `tstTestCommand_A`, respectively `tstTestCommand_B` and call command A with priority reflex and command B with priority normal. * Set the mutual exclusion of command B when calling command A * Call the action-processor's dispatch function * Assert that command A was executed and command B was blocked by checking both the commands execution flag and also the execution history of the processor.
Inhibition	* Create new commands A and B of the type `tstTestCommand_A`, respectively `tstTestCommand_B` and call them both with default priority. * Inhibit command A for 1 round. * Call the action-processor's dispatch function * Assert that command A was blocked and command B was executed by checking both the commands execution flag and also the execution history of

	the processor.
Disabling	* Create new commands A and B of the type `tstTestCommand_A`, respectively `tstTestCommand_B` and call them both with default priority. * Disable both commands and re-enable command B. * Call the action-processor's dispatch function * Assert that command A was executed and command B was blocked by checking both the commands execution flag and also the execution history of the processor.
Sequences	* Create and call a sequence of four separate commands of type `tstTestCommand_A`, for execution in consecutive rounds, i.e. the first in round 1, the second in round 2, etc. * Create and call a second sequence containing one command of type `tstTestCommand_B` for a duration of four rounds. * Call the action-processor's dispatch function and assert that Command A1 and Command B were executed. * Call the action-processor's dispatch function and assert that Command A2 and Command B were executed. And afterwards twice more for command A3+B and A4+B. * Call the action-processor's dispatch function a fifth time and assert that none of the commands were called.

The tests were performed and passed. Since they can be called automatically during build or manually when needed, these tests offer permanent validation of the execution infrastructure.

5.5.3 Hands-on Test for Action Commands

As argued in Section 4.5, it is not justifiable to create automated tests for all of the actions, because they would need permanent manual attention to separate real errors from benign changes. The existing simulation already offers a possibility for manual hands-on tests through a special decision unit. This "Remote-Bot" registers keys pressed on the keyboard and can be used to create corresponding action commands which are then passed on to the action processor for execution. Table 9 shows the implemented keys and the commands they relate to.

Table 9: Remote Control Keys

Key	**Related command**
Up Cursor	Move forward
Down Cursor	Move backward
Left Cursor	Turn left
Right Cursor	Turn right
"e"	Eat
Num Pad "+"	Pick up

Num Pad “-“	Drop
Num Pad “/”	From inventory
Num Pad “*”	To inventory
“l”	Attack/Lighting
“b”	Attack/Bite
“x”	Excrement
“m”	Move to eatable area
“k”	Kiss
“c”	Cultivate
“s”	Sleep
“1”	Dance - Salsa
“2”	Dance - Tango
“3”	Dance – Walz
“f”	Change body colour: +10 red, -10 blue, -10 green

The commands concerning facial expressions can not be tested because functionality needed from the main simulator is not implemented yet. Since this is not part of this thesis but lies in the responsibility of other project members, the commands were prepared and call the appropriate functions which at the moment only consist of the method body.

During the tests some problems occurred that need further attention:

- Move to eatable area: The creation of the joint which pulls the object to the destination area handicaps the agent’s movement. The source of the problem lies in the coordinates supplied by the destination area, which is under the responsibility of another project member.
- Lightning attack: This command is the only one where the ID of the opposing entity is explicitly defined. During the conversion of sensory data when passing it to the decision unit, the entity IDs are lost, so the decision unit can’t set the parameters. This has to be looked into by the responsible developer.
- Exception when moving objects to the inventory and back to the simulation: The MASON physics engine has a flaw that throws an exception when deregistering and then reregistering objects in the environment. The deregistering method does not clean up all references to the object and so duplicate keys are created when the objects is re-registered. The problem has been analysed but must be resolved through an update of the physics engine.
- Slow Movement: Another problem of the physics engine was encountered when moving very slowly. Below a certain threshold the agent starts moving backwards instead of forwards. The problem is of minor concern because the normal speed of movement is much higher than this value, but it should also be looked into.
- The body-colour is not adequately represented in the visualisation of the simulation: The properties are correct and are also correctly changed by the command, but the colour of the agent is not updated on the screen. This may also be a flaw in the MASON framework’s

visualisation package, but should be looked into by a developer of the environment components.

5.5.4 Live-Test and Component Application

To verify the project, the model was already compared to the requirements, the execution infrastructure was unit-tested, and the commands were checked via hands-on tests. The final validation for the concept is therefore the actual application of the new functionality. In agreement with the feature driven development process, the model was implemented piece-by-piece. Therefore, after completing the basic infrastructure and the first few actions, the new features could already be used. So parallel to the development of the last actions two small simulation set-ups were made by other groups in the development team.

On the left side of Figure 18 the "hare vs. lynx" simulation is shown. This is a three stage predator/pray scenario where the lynx (the to larger yellow symbols) hunt and eat hares (the gray circles) who in turn search for and eat carrots (the small yellow dots). The three large stones in the screenshot are additional obstacles to hide behind and to make movement patterns more complicated. This simulation uses the commands: move, turn, eat and attack/bite.

The "Fungus Eater" is a scenario developed by Masanao Toda [Tod1982] and is shown on the right side of Figure 18. The environment contains one agent (green circle in the middle), several fungi (red circles) and uranium oar (yellow and black circles). The agent's goal is to mine as much uranium as possible while balancing out his energy level. Each operation consumes energy which can be refilled by eating fungi. The simulation makes use of the commands: move, turn, pick up, drop, to inventory, from inventory and eat.

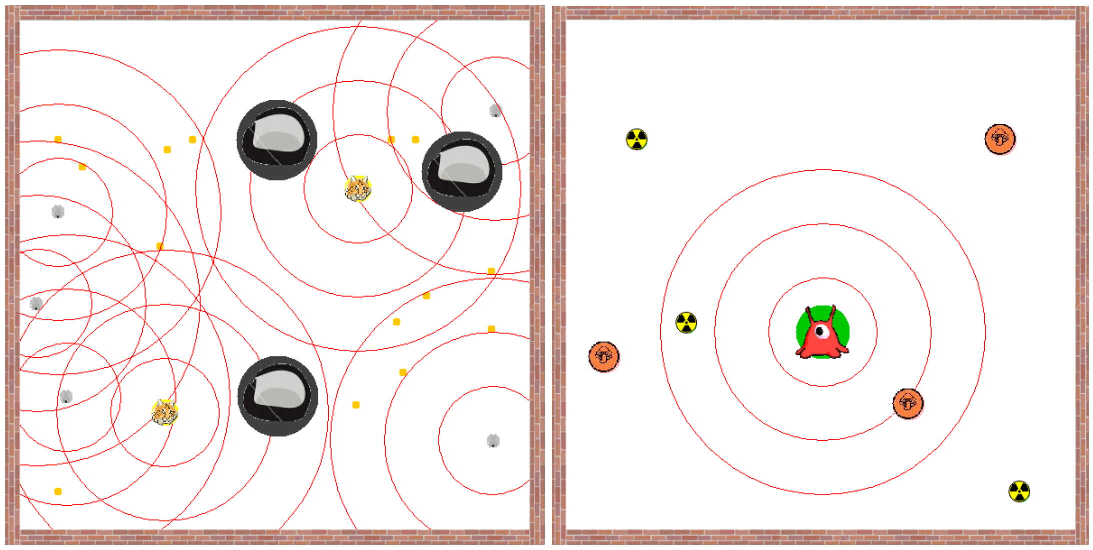


Figure 18: Simulation set-up: hare vs. lynx (left), fungus eater (right)

The basic concept has therefore been proven to be applicable and the fact that not one question was posed by the developers implementing the scenarios shows clearly that the documentation was also sufficient. The other precautions taken for ensuring extensibility and robustness will hopefully show their effect during the later course of the project and can only be verified over time.

5.5.5 Project Results Summary

Table 10 shows the summary of all presented results. For each element the result of the three test-stages (unit test, hands-on test and live test) is given and also the resulting total status, which can be either:

- Completed: All applicable tests were passed
- Prepared: Live-test still open
- Bug: A bug was found and tests must be redone when resolved

Table 10: Summary of Test Results

Element	Status	Unit Tests	Hands-on Tests	Live-Tests
Execution Infrastructure	Completed	Passed	Passed	Passed
Bite	Completed	N/A	Passed	Passed
Change body colour	Bug	N/A	Visualisation problem	Open
Change facial expressions	Prepared	N/A	Command passed, Simulation support open	Open
Cultivate	Prepared	N/A	Passed	Open
Dance	Prepared	N/A	Passed	Open
Eat	Completed	N/A	Passed	Passed
Excrement	Prepared	N/A	Passed	Open
Kiss	Prepared	N/A	Passed	Open
Lightning-attack	Bug	N/A	Interface problem	Open
Move to inventory	Completed	N/A	Passed	Passed
Move from inventory	Bug	N/A	Physics engine problem	Physics engine problem
Move	Completed	N/A	Passed	Passed
Turn	Completed	N/A	Passed	Passed
Move to eatable area	Bug	N/A	Area sensor problem	Open
Pick up	Completed	N/A	Passed	Passed
Drop	Completed	N/A	Passed	Passed
Sleep	Prepared	N/A	Passed	Open

6. Conclusion and Outlook

To conclude this work, a short summary of the project will be given followed by an outlook containing possible enhancements and future applications.

6.1 Summary and Conclusion

Today's control systems use rule-based algorithms and are reaching their limits in respect to the complexity demanded of them. On the other hand the dependency on these systems is constantly rising and flaws can become very costly. Therefore new methods for managing and controlling are necessary. For this reason, the Department of Computational Engineering at the Technical University of Vienna has initiated the ARS project, which uses the human mind as a model for a new decision making unit. Here models and methods from the fields of psychoanalysis and neurology are combined with artificial intelligence techniques. To test their theories they need to simulate certain scenarios and challenges, and compare the performance of their strategy with other state-of-the-art decision-making models such as the BDI architecture.

An established way to verify concepts like this is to create an artificial life simulation. Here agents are set to the test in different scenarios and the performance of different agents can be compared. Even if this type of simulation has a long history in various fields of artificial intelligence, no ready-available simulator can provide the complexity needed for the project at hand. A team of doctoral candidates is therefore developing a new simulator, the "Bubble World", for this very purpose.

The software for the simulator is built following a feature driven process to cope with changing requirements and contributors. It uses the highly scalable simulation framework MASON at its core and also makes use of its physics engine to create an environment as realistic as possible. Around this, the actual simulation is built in Java and using various tools such as the Eclipse IDE, Subversion for version control and DokuWiki for keeping track of all the paperwork. The simulation is round-based and each round consists of sensing, updating, processing and execution phases. The main focus of the study is the processing phase, where an exchangeable decision making unit takes the gathered sensor data as input and produces actions as output. In the course of this thesis, the components necessary to call, manage, validate and execute these actions were built.

The requirements collected by a team of psychologists, neurologists and engineers are documented as short stories following the agile development methods. They are analysed to derive all the implicitly stated requirements to produce a table of actions. This is revised by adding individual commands that are mentioned in a separate list and also by removing actions which are not needed or cannot be provided at the present moment. A second set of requirements concerns the actual execution process like the need for enabling/disabling or validating actions. These are also mainly derived from the use cases but are also given by the current state of the project and the existing interfaces. Most of the non-functional requirements found to be necessary come from the development process itself. Having many changing contributors and lacking a complete finite list of requirements – as is the case in feature driven development and other agile processes – demands exhaustive documentation and prerequisites for later extensions. The project itself – being agent orientated – also contributes some requirements such as extensive automated tests, because debugging a running program of this type is difficult and time-consuming.

On the basis of these demands, additional assumptions are made to further decrease the design space. Actions are found to be atomic, stateless, and do not delivery any feedback. A basic scheme for dealing with opposing entities is also defined, so actions are now called without explicitly naming the opposing entity, but by using any applicable one in range. The binding of other entities, i.e. picking them up or putting them into the inventory, is also constrained because the original requirements do not give enough information on this topic.

On the basis of these specifications the model is developed. Before going into details, the main problem areas are identified and common, well-known solutions are found to address them in the form of patterns:

- To separate call and execution of the actions, the command and command processor patterns is examined.
- The composite pattern is found to be the solution for managing both single commands and sequences.
- Both to decouple the action and the opposing entity – eat" command and "eaten entity" – and also to mask the real action processor behind a public interface, the proxy pattern is studied.

Using these basic strategies as their foundation, the new components are designed:

- An abstract command that – using the composite pattern – can be extended to represent a single action but also a sequence of actions.
- An abstract executor that deals with the actual operation using the command as parameter list.
- A centralised processing component – the action processor – which deals only with abstract interfaces for commands and executors, but can use these to handle all general constraints and manage the execution process.
- An optional proxy for opposing entities for actions that need to deal with these.

Around this basic framework, all the functionality stated in the requirements is added and the table of actions is analysed and enhanced to resemble a list of commands, executors, and proxies needed for their realisation. To fulfil the documentation-demands a template is developed that is exhaustive and contains all necessary information, but is still easy to use and comprehend due to a uniform structure for all actions. Applying the design-for-testability rules, possible test methods are discussed before the actual implementation. For the basic execution scheme test cases are developed and for the actual actions a general procedure for manual tests is defined.

For actual implementation, first the existing software is analysed to find suitable connection points for the interfaces and new components. Then the list of features is developed, implemented, tested and documented piece-by-piece: the inventory, the basic execution infrastructure, the actions, and the sequences.

To verify the completeness of the model it is crosschecked by walking through the requirements and listing the relevant parts of the design. Actions are tested with a hands-on test using a remote controlled agent. In addition, practical live tests were done by another group who made use of the supplied commands to implement two real simulation set-ups. During theses tests some flaws were found in the physics engine which must be resolved by an update of the framework. Some work for the remaining team members also remains, who need to implement some last functions to support already implemented commands.

6.2 Discussion and Outlook

The components developed in the course of this thesis have already been put to work and found to suit the needs. Still, from a technical view, enhancements could be made. Besides the obvious one like adding new actions, also more sophisticated composites could be created. Small macros or even independent 2nd level decision units which can deal with small routine tasks in the style of the subsumption architecture would be possible. If these things are needed or are actually contra productive, because they undermine the decision unit, is up to the core team. But with reference to the input unit, where many delicate little pieces of information are combined to create powerful symbols, the output unit could also be constructed in a similar manner. Instead of combining small pieces to come up with complex information units such as “person X has entered the room”, an abstract action like “greet the person” could be split into parts like: turn towards the person, move forward until a certain distance has been reached, stretch out the hand. Which in turn can of course also be split into pieces until the level of the actual atomic actions is reached. Of course this scheme would need a learning mechanism similar to the one on the input side and it would not be sensible to hardcode these high level actions as fixed sequences. But the idea itself is pretty obvious considering the success it had on the input side and that humans also learn to do things by combining simple actions to casual routines by practicing. This job could be done by the decision unit, but following the same arguments as on the input side, it is sensible to keep the decision unit free of all unnecessary work and move it to a higher level of abstraction.

The general design of the simulator also allows several interesting opportunities. With a few fairly simple adaptations, the existing separation of the main project and the decision unit could be enhanced even further, so the two parts are completely independent and the dependencies are injected at runtime following the example of service oriented architectures. By doing this, two things become possible: On the one hand, an adapter could be created so decision units cannot only be used in this simulation, but also tested in other environments such as computer games like [Spr2004] did. To take this idea a little further, the scheme of letting a machine based decision architecture compete against and cooperate with human users to see if they are equal feels somewhat like a Turing test. Only here the odds are more just, because the humans are in machine territory and so both have exactly the same possibilities in respect to communication and actions.

The MASON simulation itself is actually a single process simulation which could not be distributed over several machines, but it offers to stop, save and continue a running simulation. Therefore it can be hosted in a cluster where several simulations run in parallel, and it is also accessible to the request/response logic of Internet applications. With this in mind, the second advantage of completely decoupling the simulation and the decision unit would be, that the interface could be supplied as a web service. This way decision units developed by other parties from other universities could be developed, tested and then compete against each other over the internet in predefined set-ups without any of the parties having to give up their source-code or even the compiled components. The concept would be somewhat like the football competitions in robotics, only the disciplines would be different simulation scenarios and the competitors would be the different implementations of the decision units.

Considering the project as a whole, the possibilities are hard to grasp. The work needed to create the simulation and thus be able to experiment with the ideas is huge. But it is a new frontier and many ideas and possibilities will arise only after experiments can finally be done.

Appendix I: Use Cases and Feature List

The list of use-cases was taken from the project's wiki as a reference for the requirement analysis on September 19, 2009.

UC00 The Lonely Life of a Hungry Bubble

Description

A bubble – a lonely one – roams around in a world. Whenever it stumbles across something eatable, and it is hungry, it will eat. whenever necessary, the bubble rests or takes a short nap. The digestion system makes it necessary to get rid of the excrements. This will be done – similarly to eating and sleeping – wherever the bubble is at the moment of need. Other agents/bubbles are ignored – cooperation is not necessary in this sad and lonely world …

Purpose

Fast and simple generation of the following three bodily needs and possibilities to satisfy them:

- Hunger
- Excretion
- Exhaustion

UC01 Collecting and Taking Food Home

Description

In this use case the agent has to search its world for food. It is sparsely distributed and regrows at a very slow rate. Hence, it is not sufficient to eat the food at the place it has been found, moreover it has to be carried home to guarantee enough supply in times of need.

To fulfill this task an agent has to have at least the following abilities: movement, orientation (at least to find back home), perception (identification of food and home), carrying objects, and a memory to remember that food has been stored at home.

Purpose

- collecting food
- storing food at a place near to the preferred rest-position (home-like) by lie down energy source

UC02 Planting Energy Sources

Description

This use case is an extension of “Collecting food and carrying it home” and “Different types of food”. If food is sparse, collecting and storing it is only one approach to this problem. Another solution could be to plant and harvest food. The agent’s task within this use case is to find special types of food (not every food type can be cultivated), carry it home, plant it, wait until enough food has grown, and then consume it. Alternatively, other areas with different soils are better for a certain type of food to be planted than the home of the agent.

This kind of behavior can be implemented by suppressing the primary-like wish ”eat now!” and to initiate a secondarylike wish procedure which includes the possibility: ”do not eat now; plant close to home; get more food”.

Purpose

- grow own energy fields in the neighborhood
- take care of the planted sources
- harvest new energysources

UC03 Searching for type of Energy Source

The bubbles have to search for a specific type of energy source.

UC04 Transforming energy sources (cooking)

As explained in “Different types of food”, agents need different types of food for survival. A simple solution for getting different types is searching and collecting. A more complex approach is the combination of two types which results in a third type (e.g. cooking). This ability needs conscious planning to be able to reason about whether it is preferable to search for this third type, or if it is possible not to eat such a food type at all (accepting minor problems with its own body).

UC05 Location Identification

To allow an agent to navigate through its world, it has to have the ability to identify certain locations using its own sensors. “Collecting food and carrying it home” can be extended by the knowledge of where to find food to ease the search for food once it has been found at a certain position. As the human mind does not have a localization mechanism like a particle filter (see @@@probabilistic robotics), this is not a trivial task. Existing approaches using episodic memory (e.g. [28]) are not totally compatible with the psychoanalytical memory traces. Hence, this use case is needed to enable us to work on the field of localization using human like methods.

UC06 Confronted with Harmful Energy Sources

Food is not necessarily a plant, it can also be an animal (or pushed to the extreme – other agents). While plants can be harmful by supplying the wrong diet, animals have the additionally possibility to self defend themselves or to attack or even hunt and eat the agents.

Next to the abilities movement and perception, the agents have to have the ability of thread assessment. This can be done using several approaches like affects, conscious planning, acting-as-if. Possible reactions could be fleeing, self defense, attacking, grouping with other agents, …

UC07 Body Integrity

If an agent is attacked (by an animal or an other agent) or hits an obstacle hard, it is very likely that it will be injured. If injured, the health of an agent is reduced and – eventually – a sensor or actuator has reduced functionality. The agent should learn that certain situations are to be avoided. If not, the agent risks to reach its original goal (e.g. a rare food type) at high costs.

UC08 Reproduction

The aim of this use case is to give bubbles the potential to reproduce themselves. Bubbles are sexual attracted to each other by the odor they emit.

The children of 2 Bubbles are again fully equiped mature Bubbles. Things inerhited by the parents are (see also inheritance):

- new generated kind of odor
- color of the new Bubble is a mixture of the parents
- size/weight etc. is a mixture of the parents

When two compatible bubbles meet by accident in the BFG they communicate and agree on mating. The act of reproduction is displayed by a cloud surrounding the two bubles and ends with a 3rd bubble appearing. The bubbles loose child-parent relationship direct after reproduction, no raising a family. The only connection is the odor building some kind of odor-clique.

UC09 Excretion

Some nutritions are more difficult to digest than others, some are even undigestible. This results in the need of the agent to get rid of them. The result are excrements which are omitting a bad odor. An agent has to search for a suitable place – far enough from its home and other agents, but close enough not to waste to much energy for this type of business. If an agent places its excrements to close to other agents, social reproach is the result.

UC10 Motion Fatigue

Next to energy gained from consuming food, an agent also need enough stamina to do a certain action like movement. Stamina is much faster reduced than energy. Stamina is regained by resting. This use case demands from an agent to fulfil e.g. “Collecting food and carrying it home” as fast as possible. Hence, not only the primary goal of finding food and carrying it home has to be considered, moreover, how fast the agent moves and where and when it takes a rest is of importance. Selecting the wrong place for a rest could result in an attack by an animal (see “Harmful food”).

UC11 Different energy sources

The agent needs to be able to differentiate between several food types using its sensors. Eating a ”good” or a ”bad” food results in affects towards the food type. Additionally, the current location, other agents nearby, etc. can be assigned with such affects. As a result, the agent is able to differentiate between several types of foods using its affects.

UCL1 Dancing

To enable social interactions among the agents next to more basic ones like hunting and reproduction, dancing is introduced. To dance, one agent has to search for another agent which is in the ”mood” to dance. If both are agreeing to dance, they start immediately until one of them decides to follow a different goal. As a result, both agents are ”trusting” each other more than before. In times of need, this may help to survive. For example, an agent is more likely to donate food to another agent if they have establishes a trust relationship before.

Feature List: Attack

a long range attack. each entity within the visionsensor range can be attacked. the decision returns the id string provided by the vision sensor (e.g. HARE_0). if an entity with this id exists in the vision sensor and this entity provides the lightning attack interface, an attack can be performed. the strength is passed similarly to movement params. energy and stamina are withdrawn even in the case that no matching entity is present. the stroke is always of the same strength – distance is ignored.

eating/killing/sleeping are mutually exclusive to lightning attack.

stamina = strength*strength_to_energy_factor + 5

energy = strength*strength_to_energy_factor + 1

Feature List: Body Color

Goal:

> Body Color is a actuator of the Bubble. While the basic body-color of a bubble is blue specific stimuli can change its color by increasing the green respectively the red content.

Parameters:

- RGB change value (+-R/+-G/+-B)

Effects:

- turn to red: (maybe same red values, for misleading perception of others)
    - body temperature
    - reciprocal stamina
    - sexual excitement
    - shame
- turn to green:
    - reciprocal health value
    - disgust
    - stomach upset

Return Values:

- nothing
- tbd if my sensors can detect my color

Actuator Costs:

- none (costs are created by the source of the change eg. stomach)

Implementation Details:

Except the bubble no agent type disposes of changing color.

changes the color value of the bubble

clsBodyColor in IntraBodySystems needs more implementation

Feature Liste: Cultivate Food

Goal:

- Abstracted action to water plants=food
- food (plants) grow faster when they are cultivated (watered)
- food-objects are effected for a predefined time after cultivate action (imlpementation in food-object)
- too much cultivation damages food

Parameters:

- amount of cultivation (water would be metaphoric because there is no physical water) –>
- 0 = no cultivation (unnecessary)
- double value = amount cultivation

Implementation details:

- target has to be in manipulate area
- target has to implement the itfCultivation
- target has to implement (that's obvious) what happens afterward (grow faster)

Effects:

- object changes re-grow rate

Return values:

- none
- sensor vision gives the necessary info after some time

Actuator costs:

- Energy cost = prop. to amount of cultivation
- Stamina cost = prop. to amount of cultivation

Feature List: Dance

Goal:

- follow a predefined motion scheme
- follow a predefined motion scheme with respect to another agent (has to be defined in more detail)

Parameters:

- type of dance –>
  - salsa (happy) (square)
  - tango (reproduce) (triangle)
  - waltz (sad) (circle left/right)
- speed of dance

Implementation details:

- This is the place where the ACTION-SEQUENCES (TM) will be used
- could be like this: forward (10x) - rot. left (10x) - goto begin

Effects:

- agent follows predefined motion pattern

Return values:

- none

- other sensors give the necessary info

Actuator costs:

- Energy cost = prop. to speed of dance (just like move)
- Stamina cost = prop. to speed of dance (just like move)

Feature List: Excrement

agent decides to get rid of some of its indigestible nutritions. nutrition is removed from the stomach up to a maximum amount (different per entity) and stored into a new created excrement object (clsSmartExcrement). the intensity of the action is a multiplicator of the maximum amount (ceiled by the available amount). Stamina = intensisty^2 * 0.01; Energy = 0.2 * stamina. no expulsion is performed in case of lack of energy or stamina (energy and stamina should be decreased).

most of this functionality should be implemented in a class located in bw.body.interbodyworldsystems.

body-color, odor, and facial expression are allowed. every other action is incompatible.

Feature List: Facial Expression

Each bubble can change its facial expression in five dimensions:

- lens size (small, medium, large)
- eye size (small, medium, large)
- lens shape (round, lenticular, oval, dash)
- left antenna (upright, intermediate, horizontal, down)
- right antenna (upright, intermediate, horizontal, down)

the params are enums. an interface should be used to distinguish between entities with and entities without facial expression.

only while being sound asleep not facial control is available. stamina = 0.0001, energy = stamina * 0.01

Feature List: Kiss

Goal:

Create a emotional reaction in the other agent. Fiendly physical interaction between 2 Bubbles.

Parameters:

Intensity of the kiss (low, middle, strong)

Effects:

- if sucessful: trigger a (tbd) slow messenger in the kisser and one in the kissed, reduce some energy (small amount)
- if unsucessful: reduce some energy (small amount)

Return Values:

- nothing
- sensors need to sens if the kiss was accepted or rejected

Actuator Costs:

- increased energy consumption (low, middle, strong)

Implementation Details:

- only bubbles can kiss, and be kissed
- maybe side effects when kissing someone of the same sex

Feature List: Move object to eatable area

Goal:

- Move object (topmost) from manipulate area to eatable area

Parameters:

- applied force –>
- 0 = no force (unnecessary)
- double value (depends on decision unit but corresponds to physics engines forces)

Implementation details (MASON physics engine):

- create pinJoint between Objects
- reduce length of pin joint
- stop when object is in range

Effects:

- object additionally appears in eatable area (as long as the eatable area is a subset of the manipulate area)

Return values:

- none
- sensor eatable area gives the necessary feedback

Actuator costs:

- Energy cost = prop. to applied force

- Stamina cost = prop. to applied force

Feature List: Sleep

Goal:

- Agent regains stamina and health
- Energy consume decreases - selected consumers are switched off

Parameters:

- Sleep intensity –>
- 0 = no sleep (default)
- 1 = switch off first phalanx of consumers
- 2 = switch off all non-life-supporting consumers

Implementation Details:

- Agent Body supports and interface (itfSleep) that defines a
- parametring function that finally switches of the sensors/act/brain act/etc
- There, the list of consumers has to be defined

Effects:

- switch off a predefined list of consumers
- consumers can be sensors / actuators? / brain activities

Return values:

- none
- agent has to check if the sensors are down by it's own indirectly
- (reduced energy consumption)
- (faster stamina regain)

Actuator costs:

- Energy cost = 0
- Stamina cost = 0

Appendix II: Action Descriptions

The following pages show the descriptions for the actions which were developed in the course of the thesis using the documentation template:

- Move
- Turn
- Eat
- AttackBite
- PickUp
- Drop
- ToInventory
- FromInventory
- AttackLightning
- BodyColor
- FacialExEyeSize
- FacialExLeftAntennaPosition, FacialExRightAntennaPosition
- FacialExLensShape
- FacialExLensSize
- Kiss
- Cultivate
- Excrement
- MoveToEatableArea
- Sleep

Action Move

Description
The entity is moved forward or backward in the simulation environment at a given speed

Stamina and consumption

• Stamina	2^Speed * 0.01
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Direction	FORWARD	FORWARD or BACKWARD
Speed	4	Is multiplied by a scaling factor (Only applies when moving forward)

Executor Parameters

Parameter	Default	Description
SpeedScalingFactor	10	Scale the speed given by the command parameter without increasing the energy demand. Use for determining the relationship of speed to energy demand (e.g. turtle vs. rab-bit)

Mutual Exclusions
None

Constraints
None

Proxy
None

Action Turn

Description

The entity is turned either left or right for a given angle

Stamina and consumption

• Stamina	2^ * 0.001
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Direction		LEFT or RIGHT
Angle	2	In degrees

Executor Parameters

None

Mutual Exclusions

None

Constraints

None

Proxy

None

Action Eat

Description

Searches for an eatable entity in a given sensor-region and tries to eat it. If none or more than one entity is found in the region a fast messenger is sent to the brain. The eaten entity can cause both damage (e.g. stone) or have nutritional value (e.g. cake).

Stamina and consumption

• Stamina	0.5
• Energy	0.2 * Stamina

Command Parameters

None

Executor Parameters

Parameter	**Default**	**Description**
RangeSensor		Visionsensor that returns entities in range
BiteSize	1	Size of the bite the entity can eat at a time

Mutual Exclusions

- Move
- Turn

Constraints

Exactly one entity with interface "itfAPEatable" has to be in range, otherwise FastMessanger will be sent to the brain.

Proxy itfAPEatable

Method	**Description**
TryEat	return "0" if ok, or otherwise a positive determining the damage inflected.
Eat(BiteSize)	Only called after tryEat returned "0" Entity must return a clsFood object and do all appropriate steps (shrink in size, remove from simulation, etc.)

Action AttackBite

Description
Searches for a possible entity in a given sensor-region and tries to bite it. If none or more than one entity is found in the region a fast messenger is sent to the brain.
The action is called with a force as parameter => The amount of energy consumed depends on the force applied. If the force is too low the opponent endures no damage, but instead damage will be inflicted on the caller

Stamina and consumption

• Stamina	4^Force * 0.001
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Force	4	Energy to invest on trying to bite the opposing entity

Executor Parameters

Parameter	Default	Description
RangeSensor		Visionsensor that returns entities in range
ForceScalingFactor	1	Scales the force depending on the entity (e.g. a tiger vs. a rabbit)

Mutual Exclusions

- Eat

Constraints
Exactly one entity with interface “itfAPAttackableBite” has to be in range, otherwise FastMessanger will be sent to the brain.

Proxy itfAPAttackableBite

Method	Description
tryBite(Force)	return “0” if ok, or otherwise a positive determining the damage inflected. (Idea: Entity tries to bite another entity investing a given force which is proportional to an energy invested for the action. If the force is not high enough the opponent will not be bitten but instead a damage will be inflicted on the caller of the action)
bite(Force)	Only called after tryBite returned “0” Entity is bitten and must do all appropriate steps (turn into something eatable or remove from simulation, etc.)

Action PickUp

Description

Searches for a carryable entity in a given sensor-region and tries to pick it up. If none or more than one entity is found in the region nothing happens.
The item will be linked to the entity via a PinJoint and the reference is stored in the clsMobile's inventory.
Picking-up entities needs energy, but holding on to them does not. Due to the link between the agent and the carried entity movement is more difficult and slower – so consider putting it into the inventory.

Stamina and consumption

• Stamina	<Picked-up Entity's mass> * MassScalingFactor
• Energy	0.2 * Stamina

Command Parameters

None

Executor Parameters

Parameter	Default	Description
MassScalingFactor	0.01	Stamina need per unit of mass

Mutual Exclusions

- Drop
- ToInventory
- FromInventory

Constraints

Exactly one entity with interface "itfAPCarryable" has to be in range so the function can be executed

Proxy itfAPCarryable

Method	Description
getCarryableEntity()	return a reference to the clsMobile interface of the entity if carrying is possible, otherwise null
setCarriedBindingState(BindingState)	Is called when the binding state changes. Enum-values NONE CARRIED INVENTORY

Action Drop

Description
If an item is currently being carried the item will be dropped and the link to the entity will be removed.

Stamina and consumption

• Stamina	0
• Energy	0

Command Parameters
None

Executor Parameters
None

Mutual Exclusions
- PickUp
- ToInventory
- FromInventory

Constraints
An item must currently be carried (reference in the clsMobile's inventory must be set)

Proxy itfAPCarryable

Method	Description
getCarryableEntity()	return a reference to the clsMobile interface of the entity if carrying is possible, otherwise null
setCarriedBindingState(BindingState)	Is called when the binding state changes. Enum-values NONE CARRIED INVENTORY

Action ToInventory

Description
The item currently being carried will be stored in the inventory and hence removed from the simulation environment.
Moving an entity to the inventory does not consume energy, but keeping it there does. If stamina is drained to zero - items can still be moved from the inventory and be dropped, because these actions do not need energy by themselves. (so there's your way out …)

Stamina and consumption

• Stamina	for the action	0
	while in the inventory	0.25 * Mass/<Inventory's max. mass>
• Energy	0	
	while in the inventory	0.2 * Stamina

Command Parameters
None

Executor Parameters
None

Mutual Exclusions
- PickUp
- Drop
- FromInventory

Constraints
An item must currently be carried (reference in the clsMobile's inventory must be set). The inventory must have enough space left (maximum number of items and maximum weight)

Proxy itfAPCarryable

Method	Description
getCarryableEntity()	return a reference to the clsMobile interface of the entity if carrying is possible, otherwise null
setCarriedBindingState(BindingState)	Is called when the binding state changes. Enum-values NONE CARRIED INVENTORY

Action FromInventory

Description
An item from the inventory is taken, re-registered in the simulation and linked to the entity (=back to state “carried”). To rid yourself of an item in the inventory entirely first call “FromInventory” and then “Drop”.

Stamina and consumption

• Stamina	for the action	0
• Energy	0	

Command Parameters

Parameter	Default	Description
Index	0	0-based index of the item in the inventory

Executor Parameters
None

Mutual Exclusions
- PickUp
- Drop
- ToInventory

Constraints
No item is currently be carried (reference in the clsMobile’s inventory is null)

Proxy itfAPCarryable

Method	Description
getCarryableEntity()	return a reference to the clsMobile interface of the entity if carrying is possible, otherwise null
setCarriedBindingState(BindingState)	Is called when the binding state changes. Enum-values NONE CARRIED INVENTORY

Action AttackLightning

Description

Searches for an entity defined by it's ID in a given sensor-region (default: vision sensor) and attacks it. If the entity cannot be found, energy and stamina will be wasted but nothing else will happen. The action is called with a force as parameter => The amount of energy consumed depends on the force applied.

Stamina and consumption

Stamina	1 * Force * 0.05
Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
OpponentID		ID of the opponent to attack
Force	1	Energy to invest for the attack

Executor Parameters

Parameter	Default	Description
RangeSensor		Visionsensor that returns entities in range
ForceScalingFactor	1	Scales the force depending on the entity

Mutual Exclusions

- Eat
- AttackBite
- Kiss
- Cultivate

Constraints

The Opponent with the given ID must be in Range.

Proxy itfAPAttackLightning

Method	Description
attackLightning(Force)	Entity is attacked with the given force and must do all appropriate steps (damage or more)

Action BodyColor (And –Red, Green, - Blue)

Description
Set the red, green and blue component of the entities body-color. There is an action to set all values at once and actions to set the three components independently. The parameter passed refers to a change in comparison to the default color, not to the current color, i.e. setting a component to zero sets it back to default

Stamina and consumption

- Stamina 0
- Energy 0

Command Parameters

Parameter	Default	Description
Red	0	Change the value of the component +/- in regard to the default color
Green	0	Change the value of the component +/- in regard to the default color
Blue	0	Change the value of the component +/- in regard to the default color

Executor Parameters
None

Mutual Exclusions
None

Constraints
None

Proxy
None

Action FacialExEyeSize

Description
Set the eye size in the facial expression of the entity

Stamina and consumption

• Stamina	0.01
• Energy	0.2 * stamina

Command Parameters

Parameter	Default	Description
Size		Small, medium or large

Executor Parameters
None

Mutual Exclusions
None

Constraints
None

Proxy
None

Action FacialExLeftAntennaPosition FacialExRightAntennaPosition

Description
Set the position of the left/right antenna of the entity

Stamina and consumption

• Stamina	0.01
• Energy	0.2 * stamina

Command Parameters

Parameter	Default	Description
Position		Upright, intermediate, horizontal, down

Executor Parameters
None

Mutual Exclusions
None

Constraints
None

Proxy
None

Action FacialExLensShape

Description

Set the lens shape of the entity

Stamina and consumption

• Stamina	0.01
• Energy	0.2 * stamina

Command Parameters

Parameter	Default	Description
Shape		Round, lenticular, oval, dash

Executor Parameters

None

Mutual Exclusions

None

Constraints

None

Proxy

None

Action FacialExLensSize

Description
Set the lens size of the entity

Stamina and consumption

• Stamina	0.01
• Energy	0.2 * stamina

Command Parameters

Parameter	Default	Description
Size		Small, medium, large

Executor Parameters
None

Mutual Exclusions
None

Constraints
None

Proxy
None

Action Kiss

Description

Searches for a kissable entity in a given sensor-region and kisses it. If none or more than one entity is found in the region the energy is consumed, but nothing happens. If the kiss is successful then a kiss-effect is launched which sends a fast messenger corresponding to the intensity.

Stamina and consumption

• Stamina	2*(1…3=Low…Strong)*0.001
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Intensity		Low, Middle, Strong

Executor Parameters

Parameter	Default	Description
RangeSensor		Visionsensor that returns entities in range

Mutual Exclusions

- Move
- Turn
- Eat
- AttackBite
- AttackLightning
- Cultivate

Constraints

Exactly one entity with interface “itfAPKissable” has to be in range, otherwise energy will be wasted but nothing happens.

Proxy itfAPKissable

Method	Description
tryKiss(Intensity)	return true if ok, otherwise false
kiss(Intensity)	Only called after tryKiss returned true Entity is kissed and must do all appropriate steps (sent it’s own Fast Messenger, etc.)

Action Cultivate

Description

Searches for a cultivatable entity in a given sensor-region and "cultivates" it. Cultivation can be seen as synonymous to watering a plant or something similar, e.g. it causes a plant to grow faster, but if too much cultivation is done it can actually cause harm aswell. If none or more than one entity is found in the region the energy is consumed, but nothing happens.

Stamina and consumption

Stamina	2*Amount^Amount * 0.001
Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Amount	1	"Amount" of cultivation.

Executor Parameters

Parameter	Default	Description
RangeSensor		Visionsensor that returns entities in range

Mutual Exclusions

- Move
- Turn
- Eat
- Attack/Bite
- AttackLightning
- Kiss

Constraints

Exactly one entity with interface "itfAPCultivatable" has to be in range, otherwise energy will be wasted but nothing happens.

Proxy itfAPCultivatable

Method	Description
Cultivate(Amount)	Entity must do all appropriate steps (determine if amount is too much or ok and grow faster or be damaged, etc.)

Action Excrement

Description

Removes indigestible from the stomach and drops an appropriate smartExcrement at the current location. A parameter for the Intensity can be given which determines both the amount removed from the stomach and the energy consumed

Stamina and consumption

• Stamina	0.1*Intensity
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Intensity	1	Determines amount removed from the stomach and energy consumed.

Executor Parameters

Parameter	Default	Description
IntensityScalingFactor		Scales the amount removed from the stomach in reference to the invested energy (e.g. for cows vs. rabbits)

Mutual Exclusions

None

Constraints

None

Proxy

None

Action MoveToEatableArea

Description

Action searches for a "carryable" entity the manipulatable area and then moves it towards the center of the eatable area with a given force. If none or more than one carryable entity is in the manipulatable area no action will be performed but energy and stamina will be wasted. If the object is already in the eatable area no action will be performed either, but energy and stamina will be wasted.

Stamina and consumption

• Stamina	2^Force*0.01
• Energy	0.2 * Stamina

Command Parameters

Parameter	Default	Description
Force	4	Force to move the object with

Executor Parameters

Parameter	Default	Description
ForceScalingFactor	1	Scale the force given by the command parameter without increasing the energy demand. Use for differentiating between strong/weak entities
RangeSource		Visionsensor of the source-range
RangeDest		Visionsensor of the destination-range

Mutual Exclusions

- Move, Turn
- Cultivate
- Drop, PickUp, FromInventory, ToInventory

Constraints

Exactly one object inheriting the itfAPCarryable interface must be in the source-range. This entity may not be in the destination range.

Proxy itfAPCarryable (Will only be used to find applicable entities, but no method will be called)

Method	Description
getCarryableEntity()	return a reference to the clsMobile interface of the entity if carrying is possible, otherwise null
setCarriedBindingState(BindingState)	Is called when the binding state changes. Enum-values NONE CARRIED INVENTORY

Action Sleep

Description

Informs a number of body-parts about the sleep-state. These should react by turning off for one simulation round which results in lower or zero energy consumption but also means that their function will not be available or impaired. Two sleep states (light and deep) are available for each of which a list of objects can be defined which will be informed when the action is called.

Stamina and consumption

• Stamina	0
• Energy	0

Command Parameters

Parameter	Default	Description
Intensity		Light or Deep

Executor Parameters

Parameter	Default	Description
NotifyLight		Arraylist of itfSleep – inheriting objects which should be informed in case of light sleep
NotifyDeep		Arraylist of itfSleep – inheriting objects which should be informed in case of deep sleep

Mutual Exclusions

- Move, Turn
- Eat
- AttackBite, AttackLightning
- Kiss
- Cultivate
- Drop, PickUp, FromInventory, ToInventory, MoveToEatableArea

Constraints

None

Proxy itfAPSleep

Method	Description
Sleep()	Bodypart must do all appropriate steps (reduce energy consumption, deactivate features, …)

Appendix III: Source-Code for Action Implementation Example

In Section 5.4.1 the action Attack/Bite was implemented step-by-step as an example. The source-code was split into parts so explanations could be added. Here the complete listing for this example is presented in a single piece.

clsActionAttackBite

```
package decisionunit.itf.actions;

public class clsActionAttackBite extends clsActionCommand {
   private double mrForce;
   public clsActionAttackBite(double prForce) {
      mrForce=prForce;
   }
   @Override
   public String getLog() {
      return "<AttackBite>" + mrForce + "</AttackBite>";
   }
   public double getForce() {
      return mrForce;
   }
   public void setForce(double prForce) {
      mrForce=prForce;
   }
}
```

itfAPAttackableBite

```
package bw.body.io.actuators.actionProxies;

public interface itfAPAttackableBite {
   double tryBite(double pfForce);
   void bite(double pfForce);
}
```

clsExecutorAttackBite

```
package bw.body.io.actuators.actionExecutors;
import config.clsBWProperties;
import java.util.ArrayList;
import bw.body.clsComplexBody;
import bw.body.internalSystems.clsFastMessengerSystem;
import bw.body.io.actuators.clsActionExecutor;
import bw.entities.clsEntity;
import bw.utils.enums.eBodyParts;
import bw.body.io.actuators.actionProxies.*;
import bw.body.itfget.itfGetBody;
import decisionunit.itf.actions.*;
import enums.eSensorExtType;
public class clsExecutorAttackBite extends clsActionExecutor{
   static double srStaminaBase = 4f;
   static double srStaminaScalingFactor = 0; //0.001f;
   private ArrayList<Class<?>> moMutEx = new ArrayList<Class<?>>();
   private clsEntity moEntity;
   private eSensorExtType moRangeSensor;
   private double mrForceScalingFactor;
   public static final String P_RANGESENSOR = "rangesensor";
   public static final String P_FORCECALINGFACTOR = "forcescalingfactor";
   public clsExecutorAttackBite(String poPrefix, clsBWProperties poProp, clsEntity
    poEntity) {
      super(poPrefix, poProp);
      moEntity=poEntity;
      moMutEx.add(clsActionEat.class);
      applyProperties(poPrefix,poProp);
   }
   public static clsBWProperties getDefaultProperties(String poPrefix) {
      String pre = clsBWProperties.addDot(poPrefix);
      clsBWProperties oProp = clsActionExecutor.getDefaultProperties(pre);
      oProp.setProperty(pre+P_RANGESENSOR, eSensorExtType.EATABLE_AREA.toString());
      oProp.setProperty(pre+P_FORCECALINGFACTOR, 1f);
      return oProp;
   }
   private void applyProperties(String poPrefix, clsBWProperties poProp) {
      String pre = clsBWProperties.addDot(poPrefix);
      moRangeSensor=
    eSensorExtType.valueOf(poProp.getPropertyString(pre+P_RANGESENSOR));
      mrForceScalingFactor=poProp.getPropertyFloat(pre+P_FORCECALINGFACTOR);
   }
   @Override
   protected void setBodyPartId() {
      mePartId = bw.utils.enums.eBodyParts.ACTIONEX_ATTACKBITE;
   }
   @Override
   protected void setName() {
      moName="Attack/Bite executor";
   }
   @Override
   public ArrayList<Class<?>> getMutualExclusions(clsActionCommand poCommand) {
      return moMutEx;
   }
   @Override
   public double getEnergyDemand(clsActionCommand poCommand) {
      return getStaminaDemand(poCommand)*srEnergyRelation;
   }
   @Override
   public double getStaminaDemand(clsActionCommand poCommand) {
      clsActionAttackBite oCommand =(clsActionAttackBite) poCommand;
      return srStaminaScalingFactor* Math.pow(srStaminaBase,oCommand.getForce()) ;
   }
   @Override
   public boolean execute(clsActionCommand poCommand) {
      clsActionAttackBite oCommand =(clsActionAttackBite) poCommand;
      clsComplexBody oBody = (clsComplexBody) ((itfGetBody)moEntity).getBody();
      itfAPAttackableBite oOpponent = (itfAPAttackableBite)
    findSingleEntityInRange(moEntity, oBody, moRangeSensor
    ,itfAPAttackableBite.class) ;
      if (oOpponent==null) {
         clsFastMessengerSystem oFastMessengerSystem =
    oBody.getInternalSystem().getFastMessengerSystem();
         oFastMessengerSystem.addMessage(mePartId, eBodyParts.BRAIN, 1);
         return false;
      }
      double rDamage = oOpponent.tryBite(oCommand.getForce()*mrForceScalingFactor);
      if (rDamage>0) {
         oBody.getInternalSystem().getHealthSystem().hurt(rDamage);
         return false;
      }
      oOpponent.bite(oCommand.getForce()*mrForceScalingFactor);
      return true;
   }
}
```

References

[Abr2002] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, Juhani Warsta: “Agile software development methods”. VTT Technical Research Centre of Finland, 2002. Pages 9-17

[Bal2003] G.C.Balan, C.Cioffi-Revilla, S.Luke, L.Panait, S.Paus: “MASON: A Java Multi-Agent Simulation Library”. Proceedings of the Agent 2003 Conference, 2003. Pages 49-64

[Bau2001] Bernhard Bauer, Jörg P. Müller, James Odell: “Agent UML: A Formalism for Specifying Multiagent Software Systems”. Lecture Notes in Computer Science, Volume 1957/2001. Pages 109-120

[Bec1999] Kent Beck: “Embracing Change with Extreme Programming”. Computer, Vol. 32, Issue 10, Oct 1999. Pages 70-77

[Bed2001] Mark A. Bedau, John S. MacCaskill, Normann H. Packard, Steen Rasmussen, Chris Adam, David G. Green, Takashi Ikegami, Kunihiko Kaneko, Thomas S. Ray: „Open Problems in Artificial Life“. Artificial Life, Vol. 6, No. 4, 2000. Pages 363-376

[Bor2006] Stefan Bornhofen, Claude Lattaud: “Outlines of Artificial Life: A Brief History of Evolutionary Individual Based Models”. Lecture Notes in Computer Science, Vol. 3871/2006, Pages 226-237

[Bro1986] Rodney A. Brooks: “A Robust Layered Control System For A Mobile Robot”. IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, 1986. Pages 14-23

[Bry2003] Joanna J. Bryson: “The Behavior-Oriented Design of Modular Agent Intelligence”. Lecture Notes in Computer Science, Volume 2592/2003, Pages 61-76

[Bry2006] Joanna J. Bryson, Tristan J. Caulfield, Jan Drugowitsch: “Integrating Life-Like Action Selection into Cycle-Based Agent Simulation Environments”. Proceedings of Agent, 2006. Pages 67-81

[Bun2008] Christian Bunse, Antje von Knethen: „Vorgehensmodelle kompakt“, 2. Auflage. Spektrum Akademischer Verlag Heidelberg, 2008. Pages 4-18

[Bus2007] Frank Buschmann, Kevin Henney, Douglas C. Schmidt: “Pattern Oriented Software Architecture. On Patterns and Pattern Languages”. Wiley, 2007. Pages 155-158

[Cai2002] Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gomez, Juan Pavon, Francisco Leal, Paulo Chainho, Paul Kearney, Jamie Stark, Richard Evans, Philippe Massonet: “Agent Oriented Analysis Using Message/UML”. Lecture Notes in Computer Science, Vol. 2222/2002. Pages 119-135

[Cal1998]	Stéphane Calderoni, Pierre Marcenac: “MUTANT: a MultiAgent Toolkit for Artificial Life” Simulation. Technology of Object-Oriented Languages and Systems, 1998. Pages 218-229
[Coh2007]	Irun R. Cohen, David Harel: “Explaining a complex living system: dynamics, multi-scaling and emergence”. Journal of the Royal Society Interface, Vol. 4, November 2006. Pages 175-182
[Cur2003]	Dara Curran, Colm O’Riordan: “On the Design of an Artificial Life Simulator”. Lecture Notes in Computer Science, Vol. 2773/2003. Pages 549-555
[Cza2005]	Krzysztof Ctarnecki: “Overview of Generative Software Development”. Lecture Notes in Computer Science, Volume 3566/2005. Pages 326-341
[Dau1999]	Kerstin Dautenhahn: “Embodied and Interaction in Socially Intelligent Life-Like Agents”. Lecture Notes in Computer Science, Vol. 1562/1999. Pages 102-141
[Den2008]	Louise A. Dennis, Berndt Farwer, Rafael H. Bordini, Michael Fisher, Michael Wooldridge: „A Common Semantic Basis for BDI Languages“.Lecture Notes in Computer Science, Vol. 4908/2008. Pages 124-139
[Deu2007]	Tobias Deutsch, Heimo Zeilinger, Roland Lang: “Simulation Results for the ARS-PA Model”. 5th IEEE International Conference on Industrial Informatics 2007, Vol. 2, June 2007. Pages 995-1000
[Deu2008]	Tobias Deutsch, Tehseen Zia, Roland Lang, Heimo Zeilinger: “A Simulation Platform for Cognitive Agents”. INDIN 2008. 6th IEEE International Conference on Industrial Informatics, July 2008. Pages 1086-1091
[Gam1998]	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley Professional, 1995. (CD-Version) Files: pat4cfs.htm, pat3e.htm, pat4gfs.htm, pat5bfs.htm
[Hec2008]	Aaron Hector, Frans Henskens, Michael Hannaford: “A Software Engineering Process for BDI Agents”. Lecture Notes in Computer Science, Volume 4953/2008. Pages 132-141
[Ho2003]	Wan Ching Ho, Kerstin Dautenhahn, Chrystopher L. Nehaniv: “Comparing Different Control Architectures for Autobiographic Agents in Static Virtual Environments”. Intelligent Agents, 4th International Workshop, IVA 2003, September 2003. Pages 182-191
[Ho2004]	Wan Ching Ho, Kerstin Dautenhahn, Chrystopher L. Nehaniv, René Te Beoekhorst: „Sharing Memories: An Experimental Investigation with Multiple Autonomous Autobiographic Agents“. IAS-8, 8th Conference on Intelligent Autonomous Systems, 2004. Pages 361-370
[Ho2007]	Wan Ching Ho, Joao Dias, Rui Figueiredo, Ana Paiva: “Agents that remember can tell stories: Integrating Autobiographic Memory into Emotional Agents”. AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, ACM, 2007, Pages 1-3
[Ho2008]	Wan Ching Ho, Kerstin Dautenhahn, Chrystopher L. Nehaniv: „Computational memory architectures for autobiographic agents interacting in a complex virtual environment: a working model“. Connection Science, Vol. 20 Issue 1, March 2008. Pages 21-65
[Hun2002]	John Hunt: “Java and Object Orientation: An Introduction”, 2nd Edition. Springer, 2002. Pages 39-43

[Jia2007] Hong Jiang, Jose M. Vidal, Michael N. Huhns: “EBDI: An Architecture for Emotional Agents”. Proc of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems, 2007. Pages 38-40

[Kan2006] Jean-Daniel Kant, Samuel Thiriot: “Modeling one Human Decision Maker with a Multi-Agent System: the CODAGE Approach”. International Conference on Autonomous Agents: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, 2006. Pages 50-57

[Koh2008] Stefan Kohlhauser: Diploma thesis on „Requirement Analysis for a Psychoanalytically Inspired Agent Based Social System”. Submitted a the Faculty of Electronic Engineering, Vienna University of Technology, September 2008. Pages 62-70

[Kom2000] Maciej Komosinski: “The World of Framesticks: Simulation, Evolution, Interaction”. Proceedings of the 2nd International Conference on Virtual Worlds, July 2000. Pages 214-224

[Kom2003] Maciej Komosinski: “The Framesticks system: versatile simulator of 3D agents and their evolution”. Kybernetes, Vol. 32 No 1/2, 2003. Pages 156-173

[Lar2003] Craig Larman, Victor R. Basli: “Iterative and Incremental Development: A Brief History”. Computer, Vol. 36, No. 6, June 2003. Pages 47-56

[Lee2008] Seungho Lee, Young-Jun Son: “Integrated Human Decision Making Model Under Belief-Desire-Intention Framework For Crowd Simulation”. Proceedings of the 2008 Winter Simulation Conference, 2008. Pages 886-894

[Luk2005] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, Gabriel Balan: “MASON: A Multiagent Simulation Environment”. SIMULATION, Vol. 81, Issue 7, July 2005. Pages 517-527

[Moc1995] Tetsuji Mochida, Akio Ishiguro, Takeshi Aoki, Yoshiki Uchikawa: “Behaviour Arbitration for Autonomous Mobile Robots Using Emotion Mechanisms”. International Conference on Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings, Vol. 3, 1995, Pages 516-521

[Nor2000] Emma Norling, Liz Sonenberg, Ralph Rönnquist: “Enhancing Multi-Agent Based Simulation with Human-Like Decision Making Strategies”. Lecture Notes in Computer Science, Volume 1979/2001. Pages 349-357

[Pad2003] Lin Padgham, Michael Winikoff : Prometheus: “A Methodology for Developing Intelligent Agents”. Lecture Notes in Computer Science, Volume 2585/2003. Pages 174-185

[Pol1990] Martha E. Pollack, Marc Ringuette: “Introducing the Tileworld: Experimentally Evaluating Agent Architectures”. AAAI-90 Proceedings, 1990. Pages 183-189

[Pol2008] Fiona A. C. Polack, Tim Hoverd, Adam T. Sampson, Susan Stepney, Jon Timmis: “Complex System Models: Engineering Simulations”. Artificial Life, Vol. 11, 2008. Pages 482-489

[Pom1996] Jean-Charles Pomerol: “Artificial intelligence and human decision making”. European Journal of Operational Research 99/1997. Pages 3-25

[Pra2005] Gerhard Pratl, Peter Palensky: “Project ARS – The next step towards an intelligent environment”. The IEE International Workshop on Intelligent Environments 2005, June 2005. Pages 55-62

[Rai2006] Steven F. Railsback, Steven L. Lytinen, Stephen K. Jackson: “Agent-based Simulation Platforms: Review and Development Recommendations”. SIMULATION, Vol. 82, No. 9, 2006. Pages 609-623

[Rao1995] Anand S. Rao, Michael P. Georgeff: “BDI Agents: From Theory to Practice”. Proceedings of the First International Conference on Multi-Agent Systems, June 1995. Pages 312-319

[Ris2000] Linda Rising, Norman S. Janoff: “The SCRUM Software Development Process for Small Teams”, Software, IEEE , Vol. 17, No. 4, Jul/Aug 2000. Pages 26-32

[Sip1995] Moshe Sipper: “An Introduction To Artificial Life”. Explorations in Artificial Life (special issue of AI Expert), Sep 1995. Pages 4-8

[Spr2004] Pieter Spronck, Ida Sprinkhuizen-Kuyper, Eric Postma: “Online Adaption of Game Opponent AI with Dynamic Scripting”. International Journal of Intelligent Games and Simulation, March/April 2005, Vol. 3, No. 1. Pages 45-53

[Tat2006] Tatara, E., M.J. North, T.R. Howe, N.T. Collier, and J.R. Vos: “An Introduction to Repast Modeling by Using a Simple Predator-Prey Example”. Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects, 2006. Pages 83-94

[Ter1998] Terna Pietro: “Simulation tools for social scientists: Building agent based models with swarm”. Journal of Artificial Societies and Social Simulation, Vol. 1 No 2, 1998. Pages 1-12

[Tod1982] Masanao Toda: “Man, Robot, Society: Models and Speculations”. Martinus Nijhoff Publishing, 1982. Pages 100-129

[Vel2008] Rosemarie Velik, Dietmar Bruckner: “Neuro-Symbolic Networks: Introduction to a New Information Processing Principle”. INDIN 2008. 6th IEEE International Conference on Industrial Informatics 2008, July 2008. Pages 1042-1047

[Vre2004] Paolo Vresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos: “Tropos: An Agent-Oriented Software Development Methodology”. Autonomous Agents and Multi-Agent Systems, Vol. 8, No. 3, Mai 2004. Pages 203-236

[Zuc2008] Gerhard Zucker, Brit Müller, Tobias Deutsch: “Way to go for AI”. Proceedings of IT Revolutions, 2008. Pages 6-11

Internet references

[CollabNet2009]	http://www.open.collab.net
[Dokuwiki2009]	http://www.dokuwiki.org/dokuwiki
[Eclipse2009]	http://www.eclipse.org/org/
[FDD2009]	http://www.nebulon.com/articles/fdd/download/fddprocessesA4.pdf
[Java2009]	http://java.sun.com/
[MASON2009]	http://cs.gmu.edu/~eclab/projects/mason/

(All references were checked for availability on September 28, 2009)