
M.Sc. Information Technology
Faculty of Computer Science and Electrical Engineering
Kiel University of Applied Sciences

Master Thesis

Conception and Development of an Artificial Intelligence for an Online Multiplayer Game

Author Lars H. Engel
Matriculation Number

Supervising Professor
Second Examiner

Date, Place of Submission 30.08.2016, Kiel

Originality Statement

I hereby declare that

- this thesis and the work reported herein was composed by and originated entirely from me,
- information derived and verbatim from the published and unpublished work of others has been properly acknowledged and cited in the bibliography,
- this thesis has not been submitted for a higher degree at any other University or Institution.

Date, Name, Matriculate Number, Signature:

Abstract

InnoGames GmbH is a company developing browser based games and mobile games with more than 150 million registered players worldwide. One of their games is the 2009 published game Grepolis with 25 million registered players. Grepolis is a competitive strategy game in an Ancient Greek setting playable in browsers or on mobile devices (iOS & Android).

Artificial intelligence is an important topic for video games. It can be used to simulate human-like behavior for non-player characters (NPCs). NPCs can be used to create opponents in a game or to enhance the universe of a game. Introducing NPCs in a competitive strategy game can help to improve the long time play value of the game. NPCs could, for example, be used to implement in-game events, where players have to ally to defeat an NPC. Additionally, a computer program acting like a human player could be used to perform real-life server load tests to check if changes in, for example, database queries have an effect on the performance of the game, before they go into production mode.

This thesis discusses the conception and development of the first artificial intelligence for the game Grepolis. In this project, different strategies for artificial intelligence in video games, with a focus on decision making, are evaluated how good they fit to the game. Important questions concerning the implementation of artificial intelligence in video games are asked and answered for this project. A concept for an artificial intelligent NPC for the game Grepolis is built and implemented. Finally an evaluation is done to prove that the implemented solution corresponds to the requirements of the project.

After completing the project, InnoGames has a functional artificial intelligence for the game Grepolis that simulates human-like behavior. The developed solution uses the well known decision making techniques decision trees, state machines and goal oriented behavior to create a new way of decision making.

Contents

1	Introduction	1
2	Background	4
2.1	InnoGames GmbH	4
2.2	Grepolis	5
2.3	Project Management	9
2.3.1	Agile development	9
2.3.2	Versioning and branching using Git	11
2.3.3	Documentation	12
3	Artificial Intelligence in Video Games	13
3.1	Game AI versus Academic AI	13
3.2	Simplicity versus Complexity	14
3.3	Cheating in Game AI	15
3.4	Decision Making	17
3.4.1	Rule Based AI	18
3.4.2	Decision Trees	18
3.4.3	State Machines	19
3.4.4	Goal-Oriented Behavior	19
3.5	Learning in Game AI	20
3.6	Distinction of needed Techniques	22
4	Analysis	23
4.1	Grepolis in the Course of Time	23
4.2	Architecture of Grepolis	24
4.2.1	Frontend	24
4.2.2	Backend	25

4.2.3	Interface to the Frontend Implementations	26
4.2.4	Server architecture	26
4.3	Third Party Tools for Grepolis	28
4.3.1	Intelligent Agents versus Bots	28
4.3.2	Analysis of Grepolis Tools	28
4.3.3	Conclusion of Analysis of Tools for Grepolis	29
4.4	Requirement Analysis	30
4.5	Milestones	34
5	Conception	35
5.1	Programming Technology	35
5.2	Abilities of the AI	36
5.3	AI Model	38
5.4	Conception of Decision Making	40
5.4.1	Evaluation of Decision Making Techniques	40
5.4.2	The Decision Making Model	42
6	Implementation	44
6.1	Daemon	44
6.2	Design Patterns	47
6.2.1	Object Oriented Programming	47
6.2.2	Factory Pattern	50
6.2.3	Singleton Pattern	51
6.2.4	Transactions	52
6.3	The AI	54
6.3.1	Implementation in the Backend	54
6.3.2	Implementation of Decision Making	55
6.3.3	Learning and Adaption	57
6.3.4	Strategies	60
6.3.5	Daily Rhythm	64
6.4	Admin Tool	65
6.5	Quality Assurance	66
7	Conclusion	68
7.1	Evaluating the AI	68
7.1.1	Horizontal and vertical Robustness	68
7.1.2	Long-Term Evaluation	70

7.1.3	Simulation of Live Worlds	71
7.1.4	Individuality of Decisions	72
7.2	Results	73
7.3	Limitations	75
7.4	Outlook	76
A	Requirements Analysis Document	a
B	Learning the Fight System	k
C	Test cases	o

Chapter 1

Introduction

Chapter one introduces this master thesis. The motivation and goal of the project are described and the outline of the thesis is given.

Motivation One of the biggest challenges in developing an online free-to-play¹ game is keeping the churn rate of players as low as possible. Figure 1.1 shows that only about twelve percent of users that register in the game Grepolis log in a second time during the first week. After fourteen weeks this number decreases to a value under two percent. Grepolis is a game that is already seven years old. Yet it is still further developed by a team of nearly thirty employees every day. The continuous release of new content in the game is one of the reasons why the game is still successful after such a long time. Internal statistics show that releasing new features can increase the amount of daily active users (DAU) and can help to acquire new players. It is important to provide new content to the players very often so that the long term play value of the game improves to increase the player's / customer's retention.

Artificial intelligence (AI) is used in video games to imitate the behavior of a human player. This way non-player characters (NPCs) can be realized. NPCs can be used to tell the story of a game or to enhance the game world. An NPC can act as a companion of the player or even as an enemy that the player has to fight. In a competitive online strategy game like Grepolis an AI could be used to implement an opposing force that needs to be defeated. A use case could be, for example, an in-game event, where several players have to ally to be able to

¹see section 2.1

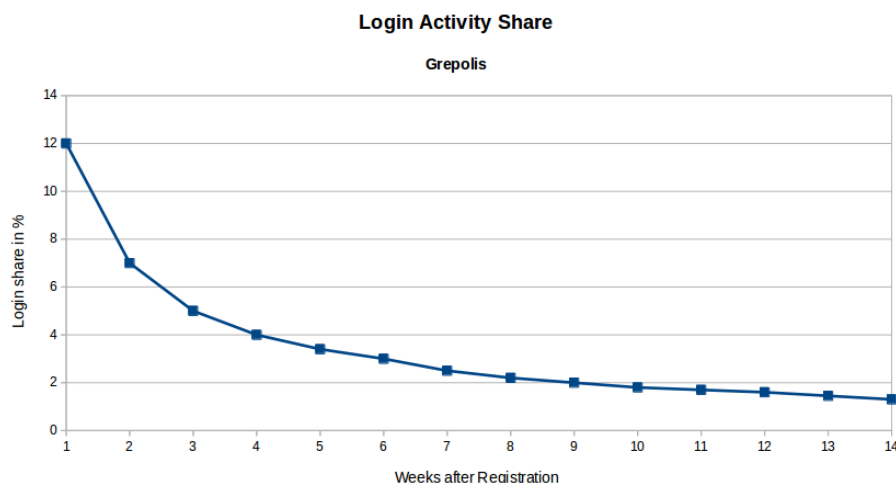


Figure 1.1: Percentage of login activity in weeks after registration

defeat the NPC. This can increase the activity of players and could also have a positive impact on the long time play value of the game.

But not only the players could benefit of an AI. Since the AI simulates the behavior of a human player, it could also be used to simulate the real world conditions for a server. Sometimes, for example, when fixing bugs, it can be useful to simulate a live world server to find issues that might be caused due to performance bottlenecks of the server architecture. It could be possible to set up a server with several thousand AI characters playing. This way, live world server conditions can be simulated. Also server stress tests could be performed using an AI.

Goal of this project Grepolis does not have an AI implemented yet. However, the game could benefit from the aspects mentioned above. Therefore, the goal of this project is to implement an artificial intelligence for the game Grepolis that simulates the behavior of a real player. It is important that the AI can be used for several different use cases. It should be possible to use the AI as an in-game feature, but also as a tool to analyze the server performance of the game. The implemented solution should be easy to maintain and extendable. Furthermore, the tool should be manageable by the community management of the game.

During the development, different techniques of artificial intelligence in video

games should be researched and applied to the game. Important questions concerning artificial intelligence in video games need to be answered, before a solution can be implemented.

Outline of the thesis The rest of this thesis is structured as follows: Chapter 2 describes the operational background of InnoGames and introduces the game Grepolis. In Chapter 3 topics of artificial intelligence in video games that are relevant for this project are addressed. Chapter 4 describes the analysis and, among other things, gives an insight into the requirement analysis. Chapter 5 explains the conception phase of the project describing the concept created for the AI. In Chapter 6 the implemented solution is documented and finally in Chapter 7 the evaluation process is described and the project is concluded.

Chapter 2

Background

In this chapter a brief introduction into the company InnoGames and the game Grepolis is given.

2.1 InnoGames GmbH

InnoGames GmbH is a developer and publisher of mobile and browser based online games. About 370 people are employed in the two locations in Hamburg and Düsseldorf.

History of InnoGames In the year 2003 the founders Eike and Hendrik Klindworth and Michael Zillmer started developing the browser game Tribal Wars as a hobby project. The following years the game got more and more popular. 2007, with already more than 50 thousand active users playing Tribal Wars, the decision was made to found a company to handle the operation and continuous development of the game. Nowadays, nine years after the foundation of the company, the six currently released games are played by more than 150 million users worldwide[1, 2].

Business Model of InnoGames The business model of InnoGames is based on the free-to-play (F2P) principle. In contrast to a pay-to-play game, where players have to spend money to get access to a game, a F2P game is playable for free. To achieve revenue a F2P game contains special features that can be bought with a premium currency, which can be obtained by spending money.

2.2 Grepolis

Grepolis is a browser based strategic online game that was released in December 2009. Despite the fact that the game is already seven years old, the game still has more than 160.000 daily active players.

Goal of the game The game is located in an ancient Greek setting. The player takes control over a small town located on an island in the Mediterranean Sea. Unfortunately, the island is also populated by towns controlled by other players. Players have to ally or fight with other players for the limited resources available on an island. The player has the task to build his own town up into a big city. Beside the construction of a town, players have to establish a military force to defend their town against attacks from other players or to conquer towns of other players, so that they can build up an empire of up to several hundreds of towns.



Figure 2.1: Screenshot of the game showing an island with several towns

Resources Nearly all actions taken in the game consume at least one of the three available resources, which are *Stone*, *Silver* and *Wood*. Resources are generated over time by resource production buildings in a town. Players can also trade resources with other players or can obtain resources as booty when attacking towns of other players.



Figure 2.2: Screenshot of the game showing an advanced town

Towns A town consists of several buildings that each have an influence on the game. The higher the level of a specific building is, the higher the effect of that building gets. The following is a description of the most important buildings that are relevant for this project. A detailed overview of all buildings can be found in the Wiki of Grepolis, which is available online[3]:

- Senate: The senate, as the administrative center of a city, is used to construct all other buildings. The higher the level of the senate is, the shorter the construction time for buildings gets.
- Farm: The farm determines how big the population of a town can get. Each level of the farm increases the amount of residents a town can have. Residents are needed to construct buildings or to recruit fighting units.
- Warehouse: In the warehouse of a city the resources are stored. The higher the level of the warehouse, the more resources it can store.
- Academy: The academy can be used to research technologies. Various features of the game need to be researched, before they can be used. With

each level the academy produces more research points, which can be used to research new technologies.

- Barracks / Harbor: The barracks and harbor are used to recruit new units for a military force. The higher the level of the barracks or the harbor is, the faster the recruitment of new units will get.
- Timber Camp / Quarry / Silver Mine: Timber Camp, Quarry and Silver Mine are the three resource production buildings and determine how much resources a town generates in a specific amount of time. The higher the level of one of the buildings is, the more resources of that type a town will generate.



Figure 2.3: Screenshot of the game showing the unit swordsman in the barracks

Units To attack other cities, or defend a city, players have to recruit units in the barracks or the harbor. In the barracks all land units can be recruited, while in the harbor naval units (ships) can be recruited. Recruiting a unit costs a specific amount of resources and population. Each unit is specialized either on attacking or defending. Land units additionally belong to one of the weapon

types distance, blunt or sharp. Additional parameters like movement speed or amount of booty a unit can carry, differ for each unit.

To reach cities that are located on a different island, players have to use transport boats that can be constructed in the harbor. Also attacking ships, like fire ships, or defending ships, like biremes, can be constructed in the harbor.

Gameplay The gameplay of Grepolis mainly concentrates on decision making. The player is not directly controlling a character’s position and does not need coordination skills. Players interact with the game using a usual computer mouse or via touching on a smartphone. By clicking on buttons features like upgrading a building or recruiting a specific unit can be triggered.

Premium Features As described in 2.1, Grepolis is a free-to-play game, which means that it can be played by anybody for free. The game contains premium features that can be bought with gold coins, which need to be bought with real money. Premium features are supposed to make the game more convenient for the users. One premium feature, for example, enables the player to schedule seven building construction orders in the building queue instead of only two.



Figure 2.4: The building queue as a standard (1) and premium (2) user

2.3 Project Management

2.3.1 Agile development

Grepolis is developed using an agile development method similar to Scrum. Two cross functional teams¹ so called *feature teams* work in sprints², that are always two weeks long, on user stories³. Since the agile development process is not relevant for this project, it is not described in detail here. For more information about agile development using Scrum see[4].

Kanban This project was done independent from the usual sprint cycles and was not part of the regular process of Grepolis. To keep track of the open tasks, a Kanban board was used to visualize the tasks. Just as Scrum, Kanban is an agile development method. But since some points of Scrum, like iterations using sprints, commitments or predefined roles, are not part of a Kanban process, it is more suited for a single software developer[5].



Figure 2.5: Example of a Kanban board

A Kanban board visualizes the work flow by dividing tasks into different columns that represent the current state of the task. A task is moved into another column when the state of the task changes. The columns of the Kanban board used in this project where *Backlog*, *To Do*, *In Progress*, *Feedback / Testing*, *Done*. In

¹A cross functional team combines people with different expertise (backend, frontend, art) to a team rather than having e.g. one team for the backend department.

²In agile development a sprint is a specific period of time during in which a defined amount of work has to be completed.

³In agile development a user story is the description of a requirement of the application.

the Backlog-Column all tasks were gathered. Even tasks that were not part of the initial scope of the project could be stacked here. The To Do-Column contains the tasks that needed to be done in the near future. The tasks, which were currently in development, were moved to the In Progress-Column. After completion of the development, tasks were moved to the Feedback / Testing column to leave room to test if a task was thoughtfully implemented. After completion of testing, tasks were moved to the Done-Column. In addition to visualizing the current state of a task, a Kanban board is also used to limit the amount of tasks that are worked on simultaneously. In this project only three tasks were allowed to be in progress at the same time.

The tool used to visualize the Kanban board was Trello. Trello is a free to use web-based project management software to create Kanban boards in a standard web browser[6]. The advantage of a digital Kanban board in contrast to a physical board in the office is that the board is always available online and not bound to a specific location.

Stakeholder involvement One core principle of agile development is a close collaboration with the stakeholders of the project. The advantage of this is that misunderstandings concerning the requirements of the project can be seen very quickly and fixed before it could be too late[7]. To follow this principle, every four weeks a meeting was set to inform important stakeholders about the current state of the project. Stakeholders that attended those meetings were in-house employees of InnoGames from different areas like software development, system administration, game design, product management or community management.

Continuous improvement Another principle that was followed was the principle of a continuous improvement process (CIP). CIP describes the process of improving a product in small incremental steps in several iterations. Each iteration follows four steps of the so called PDCA- or Deming-Cycle, which are Plan, Do, Check and Act (see Figure 2.6).

The tasks of each phase of the PDCA-Cycle are as follows [8]:

- Plan: Analyzing of the current status of the project, finding potential for improvement and development of new concepts.
- Do: Development of solutions how to implement concepts and implementation of the solutions.
- Check: Verify that the concepts were implemented.

- Act: If any deviation of planned tasks was found in the check state, measures can be created and taken to resolve issues with the implementation.

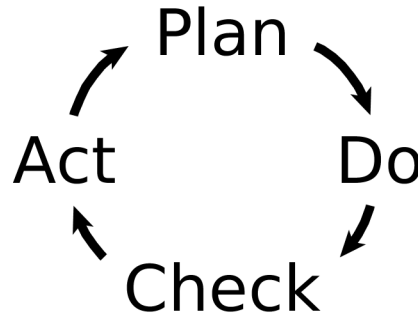


Figure 2.6: Plan-Do-Check-Act (PDCA) Cycle

Applied to this project continuous improvement meant improving the artificial intelligence that controls the non-player characters (NPC) step by step. Starting with a simple bot that could not make any decisions on its own in several iterations an artificial intelligence was created that could take actions based on decisions made on its own. While the first version executed decisions based on parameters that were predefined in classes, the final version was able to calculate parameters dynamically based on its current game situation.

2.3.2 Versioning and branching using Git

For code management InnoGames uses Git[9] repositories. Git is a standard for source code management to make versioning of a project's source code possible and to simplify the collaboration of several people on the same code. Git enables a team to easily develop on one project in different branches. This way features can be developed without the risk of influencing the main production code of the software.

“Branching means you diverge from the main line of development and continue to do work without messing with that main line.” [10]

After completing the development of a new feature the so called *feature branch* can be merged back into the main code. In this step the differences between the two branches are compared and merged together.

The main part of the Grepolis project is distributed in three repositories. One for backend, browser frontend and mobile frontend. The most important branches in the Grepolis Project are the master, beta and development branch. The master branch contains the current code which is running on the live servers, the beta branch contains the project code from the beta servers and the development branch contains the changes from the current active sprint work.

The work of this project was done in an own feature branch. Since the project lasted six months, the danger of diverging too much from the main Grepolis code existed. To mitigate this risk, the development branch was regularly merged into the feature branch. This way all the latest changes in the main code were also available in the feature branch, but the development done in this project did not influence the usual sprint work. At the end of the project the feature branch could be merged into the development branch.

2.3.3 Documentation

Documentation is an essential part of developing an application. A good documentation helps understanding the architecture of a system and gives an insight into design decisions. Especially for maintaining an application it is important to have a good documentation so that knowledge can be easily obtained. Since Grepolis is a software that is continuously improved by several developers a good documentation was needed for this project.

InnoGames uses a Wiki powered by Atlassian Confluence to maintain documentations about the products. Therefore, this project was also documented in the Wiki. A new section was created that was divided into the subsections *analysis*, *conception* and *implementation*. After the project was completed, each of the subsections contained all relevant information for the respective development phase. While the analysis subsection gives, among other things, an insight about technologies to implement an artificial intelligence in a game or about the requirements analysis, the subsection about the conception answers all questions about design decisions. Finally, the implementation subsection provides information concerning how the system was implemented. This includes the implementation of the artificial intelligence but also further used technologies.

Chapter 3

Artificial Intelligence in Video Games

Before it could be started to create a concept for the implementation, a research about artificial intelligence (AI) in video games needed to be done. In this chapter some important questions that can come up during the development of an AI for a game are answered applied to this project. Furthermore, this chapter gives an insight into techniques of AI that are relevant for this project.

3.1 Game AI versus Academic AI

Game AI, as it is seen in this project, needs to be differentiated from artificial intelligence that is developed for academic purposes such as researches. A big difference is the approach how problem solving is handled. The main focus of academic AI is to develop a rational thinking agent that is designed to solve a task or problem optimally[11]. This is the desired behavior in many use cases of AI. For example, an intelligent agent that controls a robotic arm in a factory needs to solve its task accurately every time.

In contrast to that, the goal in developing an AI for a game is not to create a solution that always finds the optimal solution to solve a task. It is not desirable that an AI will always beat the player because it knows the optimal strategy to win. Nobody wants to play a game that has an unbeatable AI implemented. It is not necessarily important that a task is solved hundred percent correctly, but that a task is solved in a similar way as a human player

would solve a task. In a first-person shooter game, for example, an NPC could have perfect aiming better than any human player could ever have. However, such an implementation would counteract the purpose of a game AI.

“Academic AI can be about a great many things. It can be about solving hard problems, recreating human intelligence, [...] Game AI should be about one thing and one thing only: enabling the developers to create a compelling experience for the player—an experience that will make the player want to spend time with the game[...].”[12]

The idea of game AI is to entertain the player and give him a believable and fun impression of an enemy. To achieve this, it is not absolutely necessary that the actions of the NPC are completely made up by the AI itself. Sometimes it can be a better solution to implement a simple rule based system that executes actions based on predefined conditions, than having a complex intelligent algorithm. The difficulty with an algorithm that handles task autonomous is that it could come to illogical decisions. Too much autonomy and unpredictability in the actions of the NPC could be undesirable[13]. Chapter 3.2 discusses when it is useful to implement real artificial intelligence and when to implement simple rule based behavior for a game AI.

3.2 Simplicity versus Complexity

When developing an artificial intelligence for a video game it is important to keep the balance between complexity and simplicity.

“It is a common mistake to think that the more complex the AI in a game, the better the characters will look to the player.

[...]

There have been countless examples of difficult to implement, complex AI that have come out looking stupid. Equally, a very simple technique, used well, can be perfect.”[14]

During the development, one has to keep thinking: *“Is there a simple way to implement the wished behavior?”*. The practice of implementing the simplest solution that works is also part of the twelve principles of the Agile Manifesto[15]. The player of a game does not see what happens “behind the curtain” of the game, so he does not care how a functionality is implemented but only cares

that the implementation works for him. In some situations, a simple rule based implementation of a functionality may be a better solution than a very complex learning algorithm, which may look worse than the simpler solution. As Ian Millington says in his book *Artificial Intelligence for Games*:

“Knowing when to be complex and when to stay simple is the most difficult element of the game AI programmer’s art. The best AI programmers are those who can use a very simple technique to give the illusion of complexity.”[14]

A problem of simple AI techniques is, however, that they can sometimes be outsmarted very easily. As described in section 3.4.1, if an NPC controlled by an AI will always react in exactly the same way in a specific situation, it is easily detectable for the players. This can, on the one side, make playing against the AI very easy for the players and, on the other side, can take away the fun of playing the game completely. If players are able to realize the pattern of decision making of an AI the illusion of an intelligent opponent vanishes completely. Especially actions of an NPC that are recognizable from the player’s perspective need to give the player the impression that the AI is acting reasonable.

In Grepolis players mainly interact with each other during fighting. Therefore, this part of the AI needed to be implemented very thoughtfully. Players might question the believability of the AI if it attacks in unusual situations or uses peculiar amounts of units like a usual human player would not do. Also, as mentioned above, if the decision making of this part of the AI is implemented too simple, players might be able to outsmart the AI. If the AI, for example, always uses the same amount of units to attack, or always attacks at the same point in time, players can prepare for such an attack very easily.

The decisions of the part of the AI that is responsible for the construction of the town are not clearly visible for the players. It is not easily detectable for a player how another player constructs his town. Therefore, it might not be as critical to use a simpler algorithm in this part of the AI as it could be for the so called “warfare tasks”.

3.3 Cheating in Game AI

During the development of an AI for a game an important question needs to be answered beforehand: *“Should the AI cheat or not?”*. With cheating an AI

would be able to take actions or gain information that are unavailable for a human player in the same situation[16].

The answer to the question, would have a fundamental impact on the way the AI can handle tasks. The advantages of letting the AI cheat are that the complexity of the implementation gets lower, because some game mechanics may not need to be implemented.

Jonny Ebert, the lead game designer of the game Dawn of War 2[17], made the following statement about implementing AI in video games:

“Cheat wherever you can. AIs are handicapped.

[...]

Never get caught cheating. Nothing ruins the illusion of a good AI like seeing how they’re cheating.”[18]

For features, where complex strategic thinking is needed for making a decision, it may be a better solution to implement a simple cheating mechanic than implementing a very complex AI. Secondly, the maintainer of the AI has more control over the decisions of the AI, when some game mechanics are bypassed. By setting for example the specific amount of units or level of buildings a town controlled by an AI has, you can influence the strategy of an AI directly. However, a cheating AI might not be well received by the community of players, because they could get the feeling of being treated unfair. A good example for a cheating AI is the AI in the games of the strategy game series Civilization[19], which are currently published by 2K Games[20]. The AI could, for example, bypass restrictions to build specific buildings or could obtain knowledge about the current progress of the human player to adapt his own strategy, which is not possible for normal players[21]. Although the Civilization games are very popular, the AI of the games was always disliked also because of the fact that the AI has an advantage towards a human player. Players of the games even created modifications, so called Mods, to exchange the AI with an own created one.

The AI in this project could cheat in several ways. One possibility would be, that the AI has the ability to access the database directly to see for example how big the army of an opposing town is, or how far the construction of such a town is. This is knowledge that cannot be easily obtained by a standard player and would give the AI an advantage in deciding how and when to attack an opposing town. Another way of cheating could be to bypass specific game mechanics to simplify the game for the AI by, for example, giving the AI an

infinite amount of resources. Without the need of determining how to spend the limited resources the decision making process for the AI could be simplified a lot. An even more drastic approach could be to bypass the recruitment and building process completely and set values like the amount of units of each type or the level of the buildings of a town directly in the database. All these examples would simplify the process needed for the AI to make a decision how to act.

For this project, the decision was made that the AI should not cheat and should only take actions and gain information that are available for usual human players. This decision may add more complexity to the implementation, but it secures that the AI has no unfair advantage towards the human players. Additionally, the point that one focus of this project was to simulate the behavior of a human player in the game with an AI, spoke against cheating so that the AI has to consider the same information as a human player to make the decisions how to act in the game.

3.4 Decision Making

In the context of game AI, the term decision making refers to the ability of an NPC to decide what to do. The process of decision making can be realized in several different ways. Yet the structure of the process is always similar. A decision maker processes knowledge as input and generates actions to be executed as output. The knowledge can be divided into external and internal knowledge, where internal knowledge is information about the character itself, like its current state or condition (e.g. health), and external knowledge is information the character has about the game environment around it.

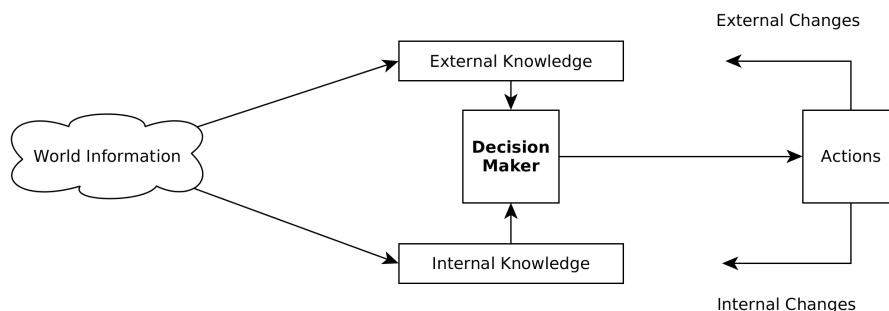


Figure 3.1: Schema of a general decision making process (adapted from [22])

The actions a decision maker generates can affect the character either internally by, for example, changing the state of a character or adapting the goals a character pursues, or can have external effects on the game environment itself. The following subsections shortly describe different techniques of decision making, which are used in this project.

3.4.1 Rule Based AI

Rule based systems consist of rules that state, when a specific action needs to be taken. The AI analyzes knowledge about the world surroundings and examines if the conditions of the rules are met. When the conditions of a rule are met, the defined action can be executed[23].

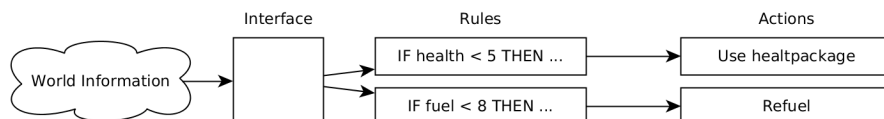


Figure 3.2: Example of a rule-based system

Rule based AI systems are very simple to implement and can be an effective method. The relation between rules and actions is always 1:1 which means that one condition (rule) will always result in the same action being triggered. This makes it easy to test the correct functionality of such a system. However, players can outsmart a rule based AI also very easily. A player might be able to exploit the AI if he understands the structure, when a specific rule triggers an action.

3.4.2 Decision Trees

Decision trees can be seen as an enhanced way of structuring rules for decision making by visualizing them in a tree like structure. The advantage of a tree structure is, on the one hand the easy visualization of the decision making process by providing an easy to understand hierarchical view of the system and, on the other hand, a tree structure can be easily and fast traversed to find a specific action[24].

The nodes of a decision tree can be divided into actions and decisions. A decision node can be followed by another decision node or an action node, while an action node should have no child nodes.

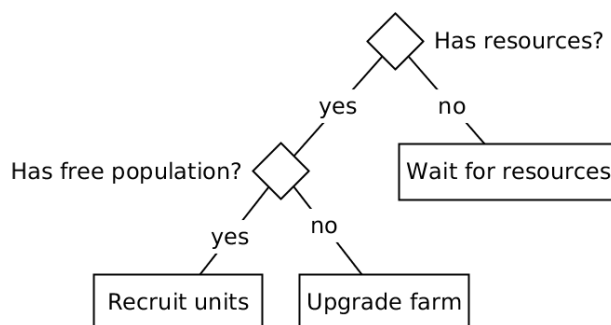


Figure 3.3: Example of a decision tree

3.4.3 State Machines

State machines¹ are a useful tool in game AI development. They provide an easy computational model to implement state driven applications. A state machine consists of several states, which define the current status of the machine. The machine can only be in one state at a time. States are connected by transitions. A transition describes the conditions that need to be met so that the machine transits from one state to another[25].

In game AI development state machines can be used to define the behavior of a character for a specific situation. A state of a character associates with specific actions or behavior. That means that a character will act in a specific way as long as he remains in the same state. When the state of a character changes, typically also his behavior changes. State machines can be visualized using a state diagram. Figure 3.4 shows an example state machine diagram for a game AI of a guard.

3.4.4 Goal-Oriented Behavior

The previous techniques for decision making focus on reacting. The decision maker processes a set of input information and produces an action as output based on the gathered information. That means that the decision maker depends on previous actions or happenings, before it can produce an action for the AI controlled character as result. For a convincing game AI this is not enough. What happens, for example, with a character that has nothing to react on?

¹In this context state machines refer to so called finite-state machines, which have a defined set of states, in contrast to infinite-state machines, which can have a varying amount of states and transitions.

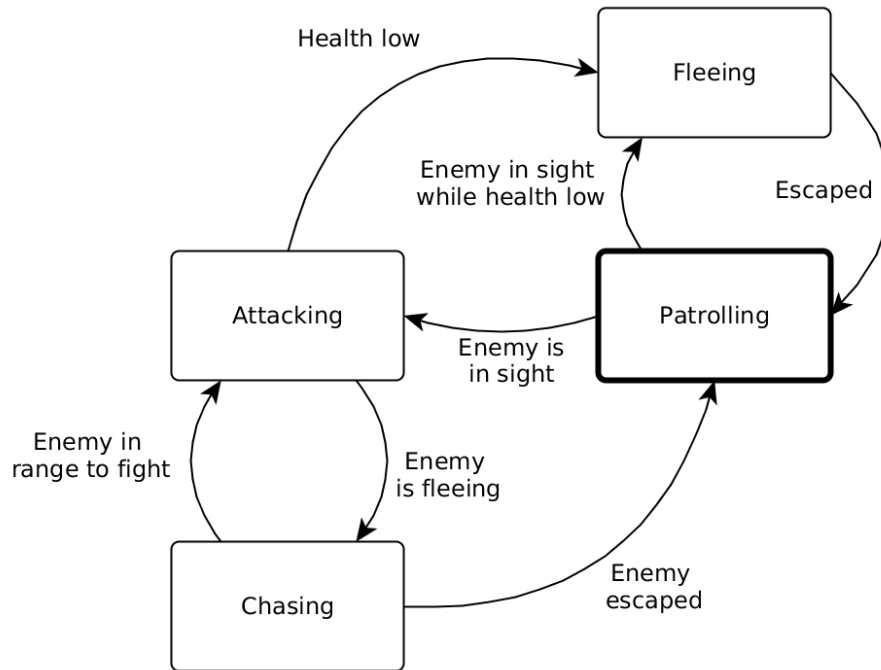


Figure 3.4: Example of a state machine for a game AI

A goal-oriented behavior (GOB) enables a game AI to come up with its own desires and goals to fulfill. A character that has the ability to choose a goal to reach by itself can act autonomously and does not rely on specific input. This can make the actions of the character less repetitive and less predictable and enables the character to adapt his actions to his specific situation [26].

A GOB consists of goals and actions. Goals define the motive a character has. They determine the actions a character takes to eventually achieve the goals. The goals of a character can have a level of insistence to signify the importance of a goal. A character should try achieve goals with a high level of insistence with a higher priority than goals with a lower level.

3.5 Learning in Game AI

Learning and Adaption Learning in the area of game development can be used to let an NPC controlled by an AI adapt to players or specific game situations. With learning two AI controlled characters may behave differently in

the same situation because they have learned different behavior from previous events in the past. Characters that can learn and behave differently under the same circumstances increase the impression of real intelligence significantly. Learning can also stand for the ability of a character to learn what action is best in which situation. This can reduce the effort that is needed to develop and implement the hard coded AI logic. Algorithms like ID3[27], for example, can be used to generate decision trees dynamically using observations from previous actions.

Online and Offline Learning Learning can be performed either online or offline[28]. When learning is done while the player is playing, this is called *online learning*. Online learning allows to adapt the behavior of the AI dynamically to the actions taken by a player. The AI can continuously learn the style how a player plays the game and can adjust decisions based on this learning. This can, for example, be used to analyze how aggressive a player plays. If a player attacks an AI controlled character more often, the AI could react with attacking this specific player also more often. Beside online learning, one can also perform *offline learning* which refers to learning that is done during the development of the AI or while the game is not currently running. Offline learning is performed in an initial training phase and can be used to process data about the game and calculate strategies based on that. After the training phase the outcome of the learning will not change until the next training phase is performed.

Difficulties with Learning Using online learning in game AI development can be risky, because the decisions of an AI controlled character may be unpredictable and are hard to test. If the character reacts differently based on the behavior of the player it can be hard to reproduce a specific bug. With offline learning it is easier to test the behavior of an AI because the strategy of the AI learned during the initial training phase will not change.

“We normally want the learning AI to be able to generalize from the limited number of experiences it has to be able to cope with a wide range of new situations.”[29]

The quote above addresses another issue with learning: the problem of *overfitting*[30]. If an AI character is confronted with a limited amount of experiences for a specific situation and adapts its behavior based on that, this could lead

to the problem that the AI adapts only to those limited experiences and would not be able to handle different situations.

3.6 Distinction of needed Techniques

As written in section 2.2, the gameplay of Grepolis does not require skills to control a player or aim at an opponent, like for example, a first person shooter requires. The main task that an AI for the game needs to be able to perform is decision making. The execution of the decisions afterwards can be realized very easily by calling the APIs and models provided by the game backend logic. Therefore, specific topics of game AI are not needed for this project. The problem of movement and path finding, for example, can be neglected completely. Also simulating human senses like vision or hearing was not needed for the AI controlled NPCs.

Chapter 4

Analysis

In this chapter the analysis process of this project is described. This includes the analysis of the architecture of Grepolis, the requirement analysis and the milestone analysis.

4.1 Grepolis in the Course of Time

Grepolis is a browser game of the early generation. When Grepolis was released in 2009, browsers were not as powerful as modern browsers. The later introduced new web standard HTML5[31] enabled the native use of multimedia and graphical content (e.g. rendering of vector graphics) in the browser. Combined with the JavaScript library WebGL[32], which can be used to render interactive 2D or 3D graphics, games nowadays run in any modern standard web browser. Additionally, browser ad-dons, like the plugin for games made with the game engine Unity 3D[33], allow even more complex games to be played in modern browsers.

However, when Grepolis was designed, those technologies did not exist. Grepolis was designed completely using so called *Dynamic HTML* (DHTML) technologies, which are HTML, CSS, JavaScript and PHP. The browser version of Grepolis can be seen as an interactive website, since it is in the fundamental structure, a usual HTML website. Initially intended to be a classic browser game, in 2013 a mobile application was developed using the Adobe Air runtime[34] so that the game can also be played on smartphones, which became very popular during the lifetime of Grepolis.

4.2 Architecture of Grepolis

The architecture of Grepolis can be divided into frontend and backend. While in the backend all game logic and storage of data is handled, the frontend is responsible for visualizing the game. Players interact with the frontend to take actions in the game. This triggers game logic being handled in the backend. After the logic is computed, data, that might have changed, is sent to the frontend by the backend.

4.2.1 Frontend¹

As described in section 4.1, Grepolis can be played in a webbrowser on a computer or on a smartphone as an app downloadable in an app store. Therefore, the game consists of two different frontend implementations.

Browser Frontend The browser frontend uses standard web development technologies like HTML, CSS and JavaScript. HTML is the foundation of the game and describes all content of the elements, CSS is used to generate the “look and feel” by e.g. defining the colors of objects or defining sprite sheets² and JavaScript can be used to dynamically modify the game content. Additional libraries like jQuery[35] or Underscore[36] are used to extend and simplify the features of JavaScript.

Mobile Frontend The mobile frontend is developed with the Adobe Air runtime. Adobe Air is a cross-platform runtime system. The fact that Adobe Air can be used to implement cross-platform applications makes it easy to develop one application and release it for several platforms. The Grepolis App is developed for Android and iOS smartphones.

¹Since the frontend of Grepolis is not relevant for this project, it will not be described in great detail. This section should only give a short insight into the frontend.

²A sprite sheet combines several images into one file. This way the number of requests to the server can be reduced and bandwidth can be saved.

4.2.2 Backend

The backend consists of several components that are needed to run the game. The most important backend components are now described in closer detail.

Backend Game Logic The game logic written in PHP contains all the code that is needed to run the game. This includes calculation of the result of actions taken by the players like, for example, calculating the outcome of a fight between two armies. The backend logic runs using the Zend Framework[37], which is a PHP framework³ to enhance developing applications in PHP. Additionally, the tool Composer[38] is used to manage several further dependencies to external libraries that are used in the backend logic. One example is the framework PHPUnit[39], which is used for implementing unit and integration tests.

PostgreSQL Database In the databases in the backend all data that needs to be preserved for a longer time is stored. This includes information about the players, their towns and units and several further not necessarily game related data like, for example, login credentials. The used database is the object-relational database management system PostgreSQL[40]. The databases can be accessed by the backend game logic using the Zend Database Adapter Zend_Db provided by the Zend Framework. Zend_Db provides an easy interface to connect a PHP application with a relational SQL Database.

Daemon Many game functionalities need to run continuously or without the interaction of a user. The production of resources, for example, needs to be calculated every second. Also game actions that need to be called at a specific time need to be handled. Therefore, some part of the game logic is handled by daemons, which run a PHP instance continuously in the background and do not require direct user input to be started.

Additional Technologies The backend consists of several additional technologies like cronjobs⁴ or a redis⁵ queue. They are not described in greater detail due to the fact that they are not important for this project.

³A framework provides a universal and reusable software environment to simplify the development of an application.

⁴A cronjob is a task that needs to be handled at a specific point in time. Cronjobs are used in Grepolis to clean up the databases or calculate rankings.

⁵Redis is a in-memory data structure store. In Grepolis redis queues are used to store information which is needed only for a short amount of time.

4.2.3 Interface to the Frontend Implementations

The backend provides Application Programming Interfaces (API) for the frontend. Since Grepolis has two frontend implementations that use different technologies, it is important to have a unique interface for communicating to the backend. Therefore, all actions or information requested by one of the frontend systems need to go through an API. If one of the frontend systems calls a method provided by an API, the API will call logic in the backend. After computing, the API will return the result of the call as a JSON⁶ object back to the frontend.

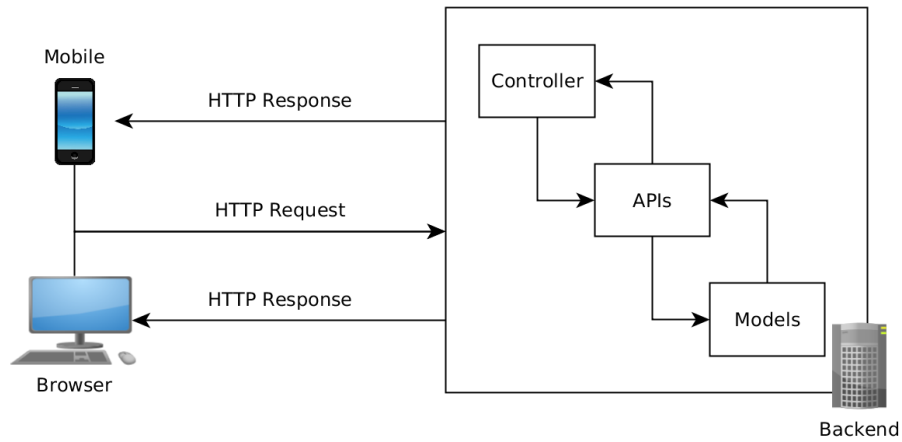


Figure 4.1: Overview of the call flow for an API

To call an API the frontend performs an asynchronous HTTP request⁷ to a specific URL using the POST method. A controller in the backend calls the correct API method and returns the result as the HTTP response back to the frontend. Listing 4.1 shows the relevant parameters for the API call to request the password of a specific user.

4.2.4 Server architecture

There are four different kinds of servers needed to run the game. For each market, where the game is available, there is one Master Database Server and

⁶JSON is a standard to transmit data objects consisting of a key-value pairs.

⁷In the HTTP protocol a client sends an HTTP request to a server requesting some kind of information or action to be performed server side. On the server this request is processed and the result is sent back to the client as an HTTP response.

Listing 4.1: Example of the parameters of an API call from frontend to backend

```

Request URL: 'http://[market].grepolis.com/api '
Request Method: 'POST'
Content : {
  'class_name' : 'ApiPlayer',
  'method_name' : 'requestPassword',
  'params' : {
    'username' : 'Peter Pan',
    'email' : 'peter.pan@neverland.com'
  }
}

```

two Master Web Server. Furthermore, for each game world there is one Game Database Server and two Game Web Server (see Figure 4.2). Each Server is responsible for a different task.

Master Database Server On the master database server the market database is located. In this PostgreSQL database all information is stored that is needed globally for the whole market. This includes, for example, the user credentials or the amount of premium currency a user has. The master database server also handles the master daemon and executes cronjobs that need to run globally.

Master Web Server The master web servers handle all market related requests. This includes API calls to register new players for a specific market or to login a player to a market. Each market has two master web servers that share the requests. This process is called load balancing and increases the availability and reliability of the game by adding redundancy. If, for example, one web server crashes there is still a second web server running. Furthermore, with load balancing the load for the servers is divided and the performance of the game can be increased. The load balancing is done by the used web server nginx[41].

Game Database Server Similar to the master database server, the game database server runs the PostgreSQL database that stores all world related data like, for example, information about the town of a specific player or the amount of units a player has. Additionally, the game database server handles the game daemons and the cronjobs for this specific world.

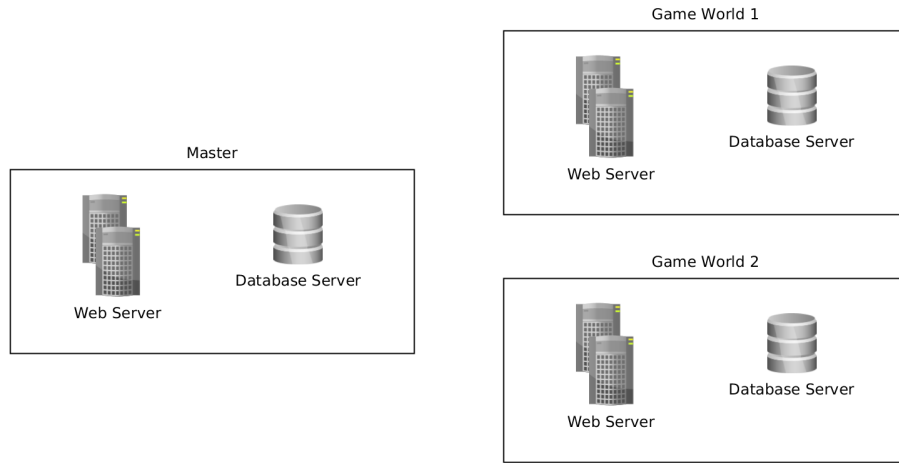


Figure 4.2: Scheme of the Server Architecture of a Grepolis market

Game Web Server The game web servers handle all game world related requests from the frontend coming to the backend. Similar to the master web servers, each world has two game web servers that share the requests that come from the players. The load balancing is also done by a nginx web server.

4.3 Third Party Tools for Grepolis

4.3.1 Intelligent Agents versus Bots

In game development the terms “intelligent agent” and “bot” are often mixed. However, the meaning of the words differs. A bot is a program that executes automated tasks defined by a user. Usually a bot cannot make own decisions and depends on the input of a user to tell him what to do. In contrast, an intelligent agent can perform actions based on own decisions. An intelligent agent does not need a user to tell him what to do, because it has own decision making algorithms, which make decisions based on parameters that might be predefined or are generated by analyzing the world surroundings of the agent.

4.3.2 Analysis of Grepolis Tools

An analysis was done to find out, what kind of tools for Grepolis exists and how they are implemented. The community of Grepolis players is very active in

creating their own scripts or tools to simplify the game. The variety goes from browser scripts to improve the user interface experience to desktop applications that can be used to automate tasks in the game, the latter being prohibited by the terms of the game.

Many tools aim on simplifying the process of recruiting units, upgrading buildings or gathering resources. The bot *GrepolisBot*[42] for example is a desktop application written in Java that can be used to automate the unit production and the construction of a town.

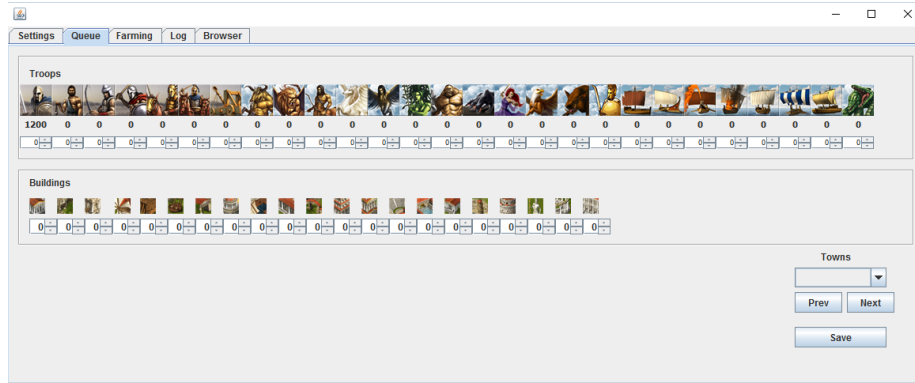


Figure 4.3: Screenshot of the tool GrepolisBot

Users can set a specific amount of units that should be produced. After the goal is set, the tool will try to recruit as many units of each kind as defined by the user. Similar to the unit production a user can define the desired level of each building. The tool will check the building level of the buildings and will try to upgrade the buildings as long as the maximal desired level is not reached.

4.3.3 Conclusion of Analysis of Tools for Grepolis

The tool described in 4.3.2 is a good example to differentiate between a bot and an intelligent agent. The tool cannot make any own decisions. It depends fully on the input of a user to tell him what to do. If a user, for example, does not enter any units to produce or sets specific building levels, the tool will not perform any actions.

All found tools can be categorized as bots, because all of them depend on user input to tell them what to do. This behavior is in contrast to the desired implementation of this project which needed to be able to make own decisions.

4.4 Requirement Analysis

A requirement analysis was done to find out what are the most relevant requirements that the final product needs to have.

Stakeholder analysis Before a requirements analysis could be performed, the relevant stakeholders⁸ for the project needed to be found, because the stakeholders are the most important source for gathering requirements. The final outcome of the stakeholder analysis was a stakeholder matrix, which visualizes all important stakeholders and their corresponding value of influence (power) and interest for the project. Depending on the interest and power a stakeholder has for the project, one can decide how to interact with the stakeholder (see Figure 4.4).

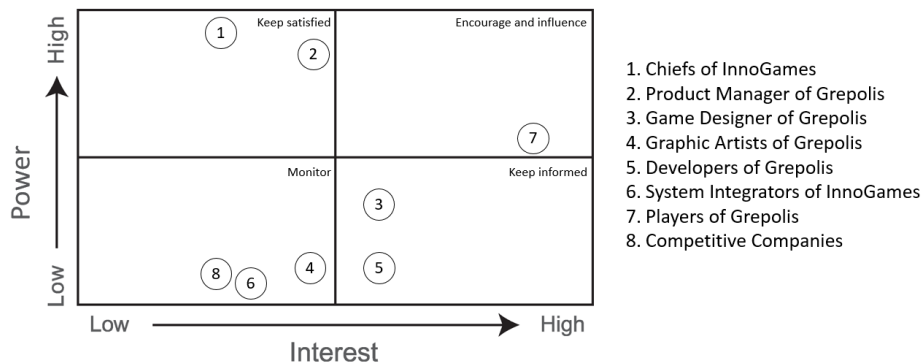


Figure 4.4: Stakeholder Matrix

Involvement of Players During the requirements analysis the question, if the community of active Grepolis players should be involved in the development process, arose. It is obvious that the players, as one of the most important stakeholders, play a key role in accepting a new feature. By involving them in the development process one can be certain that the development goes into a direction that is desired by the players. A player knows best what kind of functionality he expects from a new game feature. The difficulty with this project was, however, the long development time of nearly six months and the uncertain outcome of the project. Letting the players know very early of the

⁸A stakeholder is any person or group of people that is in any way interested in the outcome of a project[43].

development of a new feature can lead to expectations that cannot be fulfilled. This is why the decision was made to keep the development of the new feature as a secret to the players. But since some of the members of the Grepolis development team are also active players of the game, they could act as the stakeholder group “players”.

Requirements in Agile Development Since this project was developed in an agile environment, the requirement analysis did not result in a fixed amount of features that needed to be developed to accomplish the goal of the project. Unlike in traditional software development, where the amount of features to be implemented (requirements) is fixed at the beginning of the project and should not change until the end of the project, in agile development the requirements stay flexible and are likely to change during the development process. The value or quality driven approach of agile development fixes the available resources and the date to finish the project and leaves the scope of the project flexible[44]. The resources of this project was the work one developer can do. The date or time for this project was six months.

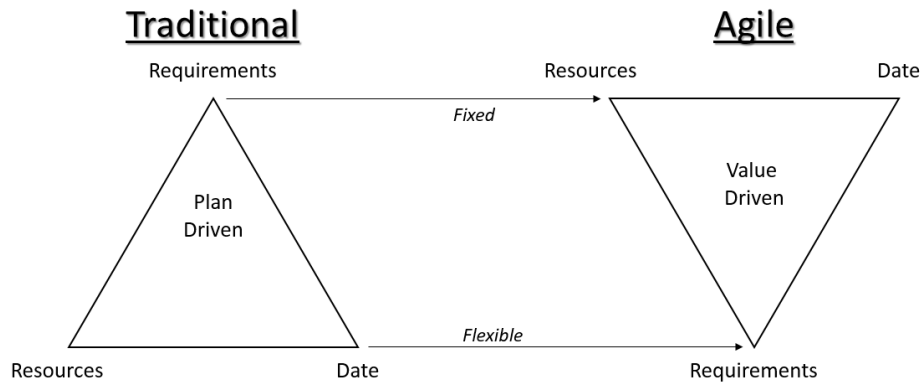


Figure 4.5: Flipping of the iron triangle with agile development (adapted from [44])

Requirements gathering In interviews with different stakeholders requirements were gathered. The initial requirement analysis resulted in one main user story that states the main goal of this project.

The main story was:

“As a player I want to be able to interact with a non-player character so that I have a better game play experience.”

The main story could be divided into smaller sub stories, which described special requirements in greater detail. Furthermore, the sub stories could be divided into functional and non-functional requirements, where non-functional requirements define the overall qualities or attributes of the system while functional requirements refer to specific functions of the system. The requirements analysis was stated in a requirements analysis document that can be found in Appendix A.

The analysis resulted in the following list of requirements:

Functional Requirements

- F-P1 As a player I want the NPC to do researches in the academy so that I have a worthy enemy.
- F-P2 As a player I want the NPC to upgrade his towns buildings so that I have a worthy enemy.
- F-P3 As a player I want the NPC to defend itself on an upcoming attack so that I have a worthy enemy.
- F-P4 As a player I want the NPC to attack other players so that I have a better game player experience.
- F-P5 As a player I want the NPC to recruit units so that I have a worthy enemy.
- F-P6 As a player I want the NPC to use heroes so that I have a worthy enemy.
- F-P7 As a player I want the NPC to trade resources so that I have a worthy enemy.
- F-P8 As a player I want the NPC to colonize towns so that I have a worthy enemy.

F-P9	As a player I want the NPC to use spells so that I have a worthy enemy.
F-P10	As a player I want the NPC to solve island quests so that I have a worthy enemy.
F-P11	As a player I want the NPC to interact with farming villages so that I have a worthy enemy.
F-CM1	As a community manager I want to create new NPCs via the admin tool.
F-CM2	As a community manager I want to be able to manage existing NPCs via the admin tool.
F-SI	As a system integrator I want to have a simulation of a human player so that I am able to do server load tests.

Nonfunctional Requirements

NFA-1	As a player I want the NPC not to be available all the time so that I have a better game play experience. (Availability)
NFI-1	As a developer I want to be able to work on the system without spending much time in working into the used technology so that I can quickly understand the system. (Implementation)
NFIT-1	As a product owner I don't want that the system can be misused to cheat in the game. (Integrity)
NFP-1	As a player I don't want that the performance of the game is worsened with this new feature so that I can play the game as before. (Performance)
NFR-1	As a product owner I want the system to be tested during the implementation and integration. (Reliability)
NFS-1	As a system integrator I want to be able to run the system on the current server setup so that minimal extra configuration is needed. (Supportability)
NFU-1	As a player I want to be able to distinguish NPCs and other players. (Usability)

4.5 Milestones

After the requirements were gathered, a milestone analysis was done to define when a specific phase of the project needed to be finished. The following milestones were defined:

	Milestones	Description	Planned Date	Achieved Date
1	Start of the project		01.03	01.03
2	Concept ready	Requirement analysis is complete, model for AI is developed and concept to implement AI is complete.	31.03	04.04
3	Feature Freeze	No new features should be implemented afterwards.	15.07	20.07
4	Implementation finished	All functionality and tests are implemented.	31.07	02.08
5	Testing finished	All functionality is tested.	15.08	15.08
6	Submission of thesis	Documentation and Thesis are complete.	30.08	29.08

Table 4.1: Milestones

Milestone 1 states the start date of the project. Milestone 2 is the point, when the concept to be implemented should be completely finished. The outcome of milestone 2 was a requirements analysis document and a described concept for the implementation. The next milestone is the feature freeze with milestone 3. After the feature freeze no new features should be implemented. The feature freeze was introduced to secure that all implemented features are thoughtfully implemented and that there is enough time in the implementation phase to test all implemented features and to document the product. Milestone 4 is reached, when the implementation is completely done. This includes cleanup of the source code, writing of unit and integration tests and in-source documentation. After that, when milestone 5 is reached, the project should be tested concerning acceptance and functionality. Finally, milestone 6 states the end of the project when the documentation and the thesis should be done.

Chapter 5

Conception

After the analysis was complete, the concept for the AI could be created. This section gives an insight into the decisions made during the conception phase. The AI techniques found during the research phase are evaluated concerning their fit to the requirements of the game. Finally, the created model for the AI and for the decision making is described.

5.1 Programming Technology

The technology used to implement the AI was not restricted by the product owners. This is why during the conception phase the question arose which programming technology to use to implement the project. Two possibilities existed for the implementation: The project could be implemented on the one hand by integrating it into the backend code of Grepolis, on the other hand it could be possible to create an independent solution that uses the provided APIs to communicate with the backend similar to the mobile and browser frontend of the game.

The advantages of implementing the project in the backend code were firstly that this enables to use all the functionality of the backend directly. This means being able to use the models of the game without the need of extra APIs or interfaces and the ability to access the database directly. Secondly the integration into the backend code means that artificial intelligence and backend code are implemented in the same programming language, which makes the maintenance for the developers easier. On the downside using the same technology as the

backend code for the AI can increase the load for the servers that run the game. Also one can argue that the artificial intelligence is logically not part of the backend code and should also be separated physically from the backend. Additionally, it can be questioned if PHP as a scripting language, whose main purpose is the creation of dynamic web applications[45], is the best solution to implement an artificial intelligence.

The implementation of the AI as an independent application that communicates with the backend via APIs enables to use a language that might be more suited for AI programming like LISP or PROLOG[46]. Using LISP or PROLOG could make an implementation of, for instance, decision trees or state machines¹ easier. On the other hand, using an independent solution would mean that interfaces are needed to communicate with the APIs from the backend. Another major disadvantage for using another technology is that a runtime environment for the used technology needs to be installed on the servers that run the game. With more than 50 running game worlds each using three servers this would have a major impact for system administration.

After consideration of all the facts, the decision was made to implement the AI mainly by integrating it into the backend code of the game. The need of installing additional runtime environments on the servers was not acceptable. Also the fact that using a technology like LISP or PROLOG, which are not used in the company InnoGames yet, makes the maintenance of the code for developers harder, since they have to get accustomed to the language, had a big influence on the decision. Additionally, PHP supports since version 5 object-oriented programming. This means that an implementation of, for example, decision trees or state machines can be easily achieved (see section 6.3).

5.2 Abilities of the AI

During the conception phase, an analysis was done to find out what the abilities are that the AI needs to have. Since the AI needs to be able to perform the same tasks as a human player, the functionality of the game was analyzed. All possible actions of a player were found and ranked according to their importance for progressing in the game. The actions were later categorized into essential and additional abilities. While the essential abilities are crucial to play the game, the additional abilities can be seen as not so important. Furthermore, the essential

¹see section 3

abilities could be categorized into the two main game play mechanics of the game which are construction of a town and warfare tasks.

This division helped during the development process since it made the decision of what to implement at which time easier. It was clear that it should be started with implementing all the essential abilities, so that the AI is able to play the game on a basic level. After the essential abilities were working, remaining project resources could be used to implement some of the additional abilities.

Essential abilities

- Construction of a town
 - Construction and upgrading of buildings
 - Researching technologies in the academy
- Warfare tasks
 - Recruiting of units
 - Attacking of other players
 - Defending of the town

Additional abilities

- Interaction with farming villages
- Assignment and use of heroes
- Trading resources in the marketplace
- Colonizing towns
- Use of spells
- Solving island quests to get rewards
- Interaction with alliances
- Chatting with players

5.3 AI Model

Basic Model To develop an appropriate model for the AI the model proposed by Ian Millington in his book Artificial Intelligence for Games[47] was taken. However, the model did not fit completely to the requirements of the AI needed for Grepolis. Some parts like movement, animation and physics of Millingtons model were not relevant for the AI of Grepolis. So they could be removed from the model. Besides that, the model was further developed to support a modular implementation of the game features.

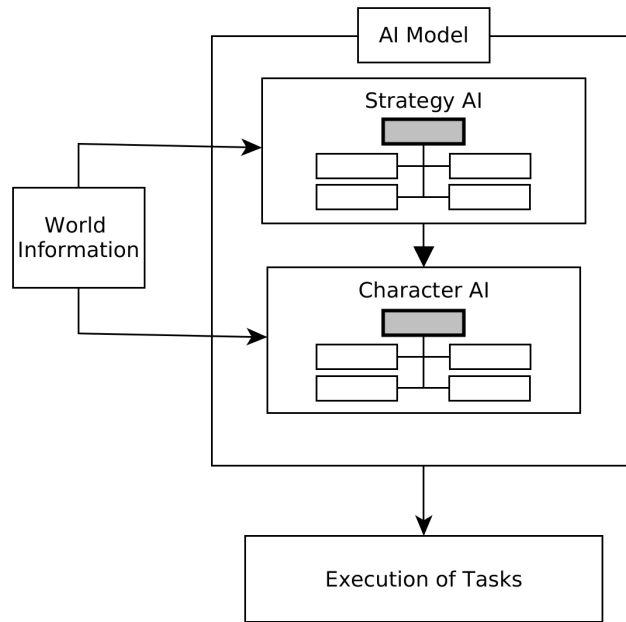


Figure 5.1: Developed model of the AI

Strategy and Character AI The developed model as shown in Figure 5.1 can be divided into two parts, a Strategy AI and a Character AI. This division was needed since in Grepolis a player can have several towns under his control. Thus also an NPC controlled by the AI should be able to control several towns. The two parts of the AI handle different areas of the game. While the Character AI is responsible for managing tasks that concern a specific town, the Strategy AI manages decisions that are relevant for the overall strategy of an NPC or concern several towns. Since one Character AI is always responsible for

one town, an NPC with several towns under his control will also have several Character AIs each controlling a specific town and one Strategy AI managing the overall strategies of the NPC (see Figure 5.2).

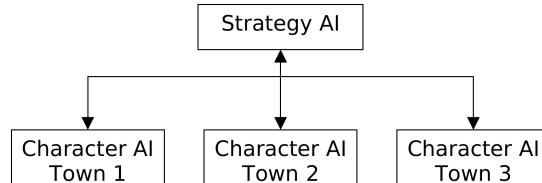


Figure 5.2: Simplified AI Model for an AI entity with several towns

Modular Structure As written above, the model for the AI was developed to support a modular implementation of the features of the game. That means that different features can be added or removed to the AI with minimal effort and without influencing other modules. To support that, the Strategy AI and the Character AIs each contain a main decision making class. These main classes call all the active modules that are responsible for making decisions concerning a specific part of the game. The *Main Character Decision Maker*, for example, could call the modules to decide what buildings to upgrade, what units to train and what technologies to research in the academy. Each module returns its decisions to the Main Character Decision Maker. Within the Main Character Decision Maker all the decisions from all the modules are afterwards processed and executed.

Figure 5.3 shows an example of some modules processed by the Character AI.

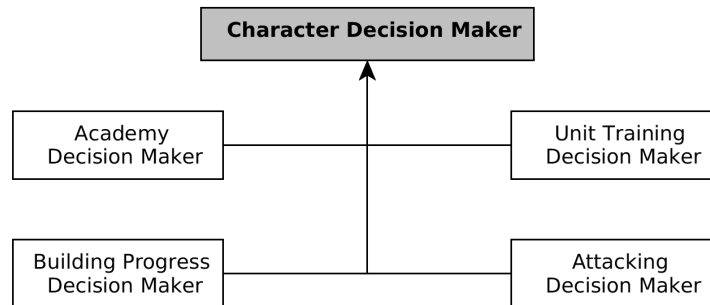


Figure 5.3: Example of modules for the Character AI

5.4 Conception of Decision Making

5.4.1 Evaluation of Decision Making Techniques

All the individual decision making techniques described in section 3.4 did not fit entirely to the requirements of this project. This section will explain why it was necessary to use several different decision making techniques and combine them to a new way of decision making.

Decision Trees Decision Trees are a good way of structuring the process of decision making. They can be used to implement reactive behavior straightforward because they provide “*an efficient way of matching a series of conditions*”[48]. However, only using decision trees for making decisions of an AI might not be the best idea. An NPC that has an AI that only uses decision trees for decision making always relies on input coming from the environment before it can come to a decision. The decisions will always be reactive and such an NPC would never set its own goals to achieve.

State Machines State Machines are very useful in separating an AI into different states that define different behaviors of the AI. This makes structuring an AI much easier and reduces the complexity of a model for decision making. Having defined states can also help in debugging an AI. If it is known what behavior one can expect in which state, this can be easily tested. The downside in using state machines lays in the transitions between two states. If there are several conditions for a transition between two states, this can get unnecessarily complex if using only state machines.

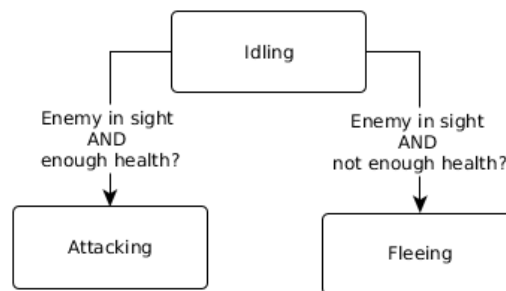


Figure 5.4: A state machine with two very similar transitions

The very simple state machine diagram in Figure 5.4 shows an example of the issues with transitions of state machines. The AI of this example has two transitions from the state Idling to the states Attacking and Fleeing. However, the transitions have very similar conditions that process almost the same information individually. So it would be better to have a solution that can evaluate several conditions and can chose the next state based on that input.

Additionally, the concept of state machines did not fit completely to the requirements of this project. Typically, a state machine only allows a character to be in one state at a time. The character of the example in Figure 5.4 cannot be simultaneously in the states Idling and Attacking because it is simply not possible that a character does both things at a time. The states of a state machine typically define the behavior of a character. They determine what actions he can do and how he acts. In this project, though, the state of a character did not need to be defined that strictly. A character that is attacking a town of another player might be in a state called “Attacking”. The gameplay of Grepolis still allows the player to continue doing other actions not related to the attack, like upgrading the town or researching technologies. That means the character might be in the state Attacking but he could also be doing other not fight related actions at the same time.

Goal-Oriented Behavior An AI that has Goal-Oriented Behavior (GOB) is able to define goals to achieve on its own. This gives the player the impression that the AI handles autonomously which increases the believability of the AI. The AI can also adapt to a player’s behavior by defining goals based in the previous actions of a player. With GOB an NPC obtains a long term goal existing over several iterations of decision making. But developing an AI only with GOB is hardly imaginable. GOB only drives the AI into a specific direction. But how should the AI come to the decision that it will pursue a specific goal? For this another kind of decision making is necessary.

5.4.2 The Decision Making Model

Combining different Decision Making Techniques

Each of the previous described decision making techniques has their advantages and disadvantages. Individually they would all not fit to the requirements needed for this project. This is why the implemented solution for decision making in this project is a combination of all three techniques.

As mentioned before, transitions in state machines can get problematic. To solve the issue of the example in Figure 5.4 one can combine state machines with decision trees by replacing the transitions of the state machines with conditions from decision trees. This is an approach also proposed by Ian Millington in his book *Artificial Intelligence for Games*[48]. Figure 5.5 shows the same states and transitions as Figure 5.4. However, now the conditions are only processed once for both state transitions at the same time. Instead of having to check four conditions the decision maker in Figure 5.5 only has to evaluate two conditions.

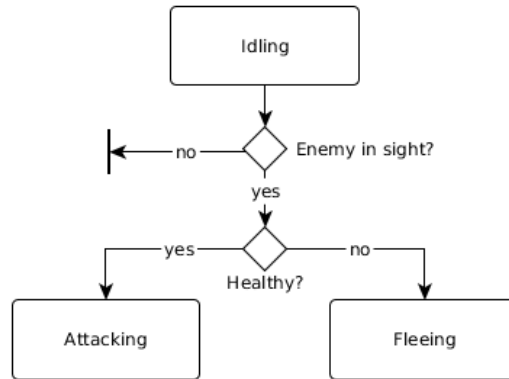


Figure 5.5: Combination of a state machine with a decision tree

Millington's approach only takes the conditions from the decision tree model and keeps the rest from the state machine model. However, as mentioned in section 5.4.1, states in this project are not as strictly divided from each other as they are in a usual state machine. That is why a slightly different approach of combining state machines with decision trees had to be taken. Instead of using state machines as the base model and adding features of decision trees, in this project the opposite was done. The principle of decision trees was taken as the base model and features of state machines were integrated into this model so

that decisions can now also check the state of an NPC and actions can lead into state changes.

Beside the known approach of combining decision trees with state machines, this project also integrated GOB into the decision making model. By integrating goals into the concept of decision trees the AI gets the ability to make up individual long-term objectives to achieve. Similar to the integration of state machines, the decisions in the decision tree can also check what goals a character has and make decisions based on that. Additionally, actions can result in new goals to pursue.

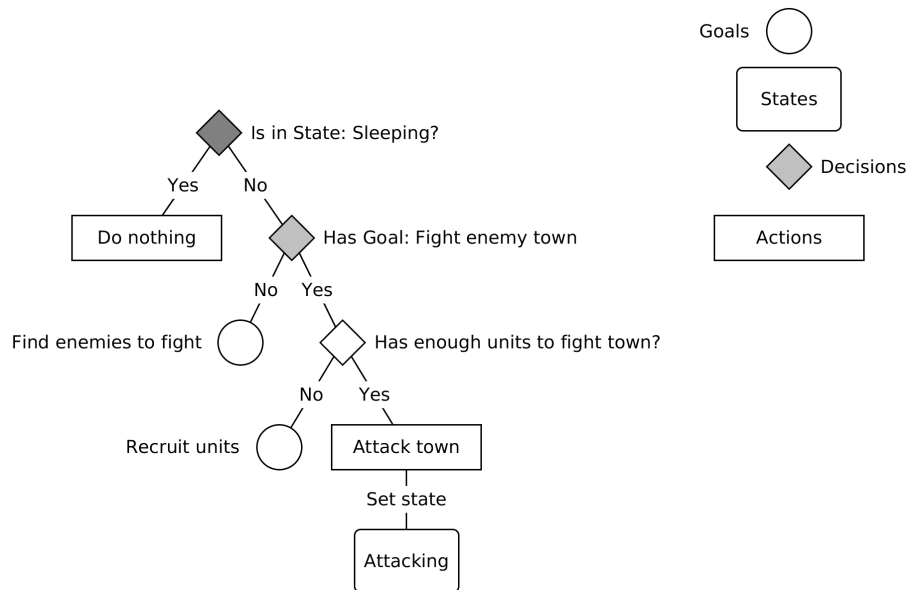


Figure 5.6: Combination of decision trees, state machines and goal-oriented behavior

Figure 5.6 shows an example of a decision making process how it is used in this project. It is visible that both, decisions and actions, integrate goals and states in the process of making the decision if a character should attack another town. The diamond symbol stands for decisions that need to be evaluated. To visualize the integration of states and goals, decisions that include states are marked dark gray, while decisions using goals are light gray and all other usual decisions are white. Actions that lead to new goals to pursue are shown as circles, while normal actions are shown as rectangles.

Chapter 6

Implementation

This chapter addresses the implementation phase of the project. Besides the description of the implementation of the AI, used design patterns are explained.

6.1 Daemon

The need for a Daemon Grepolis needs a different kind of artificial intelligence, as it is usually implemented in a game. Typically, the AI of a game is only needed, while the player is playing the game or interacting with an NPC controlled by the AI. For Grepolis this does not apply. The game play of Grepolis required an AI that is available all the time even when no player is playing the game at that time. PHP, which was the chosen technology¹, is a programming language made for request-response applications. This means that the web server will create a PHP instance when a client requests an action. After completion of the action and return of the result, the PHP instance will be destroyed. Without a request beforehand no code on the server will be executed. This would be a problem for the AI, as this would mean that the code for the AI can only run after a player has done a request. To solve this issue a daemon was created that runs the code for the AI.

Forking As written in section 4.2.2, a PHP daemon enables to run instances of PHP programs continuously on the web server without the need of a user request. The daemon is realized using *forking*. One process creates, before it

¹see section 4.2 and 5.1

dies, a copy of itself as a child process that only differs from the parent process by the process identifier (PID). The program flow after the fork will now be determined by the PID. The parent process will die while the child process will start a new instance of the PHP daemon. Afterwards the child process will create a new child process and the loop starts over again. This way an infinite loop of PHP processes can be achieved. Listing 6.1 shows how forking can be used to implement a daemon loop in PHP.

Listing 6.1: Implementation of a daemon loop in PHP

```

1 $pid = pcntl_fork();
2 if ($pid == -1) { // Problem with forking
3     return false;
4 } elseif ($pid) { // Parent process
5     return $pid;
6 } else { // Forked child
7     $this->daemon->runLoop();
8     exit(0);
9 }

```

Task providers Inside the daemon loop several task providers are called. The task providers are responsible for gathering the tasks that the AI needs to handle in each loop. After that, task handlers execute the previously gathered tasks. Figure 6.1 visualizes this flow.

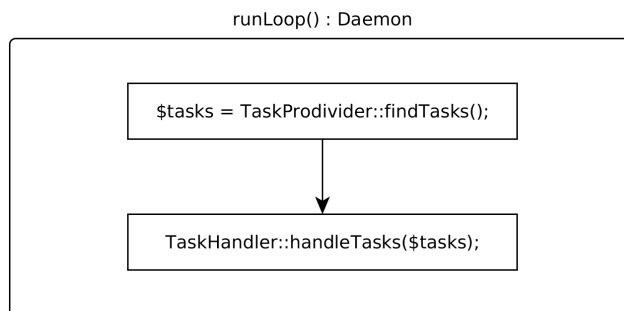


Figure 6.1: Task providers and handlers are called in the daemon loop

The most important task provider is the *GameTimedZoeCallTaskProvider* which provides all the so called *Timed Call Tasks* that are relevant for the AI. Timed calls are calls of methods that need to be executed at a specific point in time.

This way actions of the NPCs can be executed at specified times. A table in the database is used to store the information about which method needs to be called at which time. Figure 6.2 shows the structure of the table.

Column Name	Data type	Description
id (primary key)	integer	Unique identifier for each row in the table.
classname	text	Name of the class that contains the method.
methodname	text	Name of the method that should be called.
arguments	text	Parameters for the method call stored as JSON.
execute_at	integer	Unix timestamp of the point of execution.
priority	integer	Priority of the call.

Figure 6.2: Database table `game_timed_zoe_call`

The `GameTimedZoeCallTaskProvider` gathers all tasks that are due to be called and passes them to the `GameTimedZoeCallTaskHandler`. For each due task an object of the type `GameTimedZoeCall` is created and passed to the method `call()` of the task handler.

Listing 6.2: Execution of the timed calls

```

1 function call(GameTimedZoeCall $call)
2 {
3     $class_name = $call->classname;
4     $args = json_decode($call->arguments, true);
5     $class = new $class_name();
6     return call_user_func_array(
7         [$class, $call->methodname],
8         $args);
9 }

```

This method, shown in Listing 6.2, firstly uses the PHP function `json_decode(<JSON>, <Associative>)` to transform the arguments from JSON, how they are stored in the database, to an PHP array. After that, an object of the given class is created and lastly the PHP function `call_user_func_array(<Class Name>, <Method Name>, <Parameters>)` is used to call the given method.

6.2 Design Patterns

In this section some design patterns and used programming paradigms are described related to how the project benefited from their use.

6.2.1 Object Oriented Programming

This project, as the rest of the backend code of Grepolis, was written following the object oriented programming (OOP) paradigm. OOP is a programming paradigm to structure complex software applications. The fundamental data structure are *objects* that communicate with each other. Objects can contain data fields, which are often called *attributes*, and algorithms known as *methods*. Typically, in OOP objects are instances of *classes* that define the attributes and methods an object has. Different objects instantiated from the same class do not have to be identical, but they should have enough similarities that they can be defined as one group. Therefore, classes are *abstractions* of objects. OOP is based on the main features *Encapsulation*, *Inheritance* and *Polymorphism* [49].

Encapsulation In OOP the attributes and methods of objects are encapsulated in the classes they are instantiated from. This leads to the advantage that the rights to who can access the attributes and methods of an object can be restricted. In PHP, for example, the keyword *private* indicates that an attribute or method should only be accessed by the class itself. *Protected* attributes or methods can be accessed by the class itself and any inherited child classes, while *public* attributes and methods can be accessed by any class. Another advantage of encapsulation is that classes can take care of consistency of their data by themselves. By restricting the access to attributes, it is possible to, for example, only allow two fields to be changed at the same time by providing a *Setter* - Method.

Listing 6.3 shows an example of encapsulation in OOP. The attributes `first_name` and `family_name` are declared as `private`. This means that they cannot be accessed directly from the outside of the class. The method `setName` is `public` and thus can be called from any other class. The method `setName` takes two parameters and sets the attributes `first_name` and `family_name` to the corresponding values. This way it can be secured that both variables are always set.

Listing 6.3: Example of encapsulation of attributes and methods

```

1  class Human
2  {
3      private $first_name;
4      private $family_name;
5
6      public function setName($first, $family)
7      {
8          $this->first_name = $first;
9          $this->family_name = $family;
10     }
11 }

```

Inheritance Inheritance can be used to create generalized parent classes that inherit attributes and methods to several different other child classes. Child classes are usually a specialization of their parent class with additional attributes and methods. This way a relation between different classes with same attributes can be achieved.

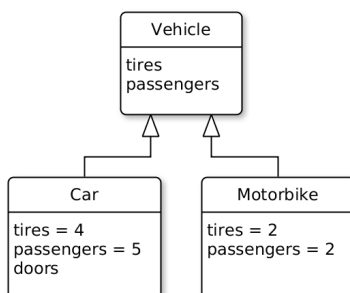


Figure 6.3: Example of inheritance of classes

Figure 6.3 shows an exemplary implementation of inheritance. A class *Car*, for example, is a specialized child of the class *Vehicle*. That means that it inherits the attributes of the *Vehicle* class, which are the same for every type of vehicle. Every vehicle has, for example, a specified amount of tires and can carry a specified amount of passengers. The class *vehicle* does not specify how many tires a vehicle has or how many people a vehicle can carry, the *car*, however, is a specialized type of vehicle with typically four tires and the possibility to accommodate five passengers. Additionally, a car has doors, which is an attribute that is not existing in all types of vehicles. Another specialization

of the class vehicle could be a motorbike with two tires and two passengers.

Polymorphism Inheritance in OOP also enables polymorphism. Polymorphism means that an object may take several different forms. Child classes, for example, can implement one method of a parent class in different ways.

Listing 6.4: Example of polymorphism

```
1 abstract class Animal
2 {
3     abstract public function makeNoise ();
4 }
5
6 class Dog extends Animal
7 {
8     public function makeNoise()
9     {
10         print_r("Wuff!");
11     }
12 }
13
14 class Cat extends Animal
15 {
16     public function makeNoise()
17     {
18         print_r("Miau!");
19     }
20 }
```

Listing 6.4 shows a basic example for polymorphism. The abstract class² `Animal` has an abstract method `makeNoise`. The two classes `Cat` and `Dog` implement the method on different ways.

There are several further ways how polymorphism can be achieved in OOP. Some languages, for example, support method overloading, which means that the signature of a method defines what implementation of the method should be called. Another way of polymorphism is overriding of a parent method in the child class (method overriding).

²Abstract classes cannot be instantiated directly. Only classes inheriting from abstract classes can be instantiated.

6.2.2 Factory Pattern

The factory pattern is a pattern of the group of the creational patterns. When using the factory pattern the developer calls a factory method instead of calling the constructor of a class to create an instance of an object. This way the problem of having to specify which exact class will be created is solved.

Listing 6.5: Example of the Factory Pattern used in this project

```

1  class BuildingStrategyFactory
2  {
3      public static function createBuildingStrategy ($strgy)
4      {
5          switch ($strgy) {
6              case BuildingStrategyEnum::STRATEGY_LAND_DEFF:
7                  return new DataLandDeffBuildingStrategy ();
8              case BuildingStrategyEnum::STRATEGY_LAND_OFF:
9                  return new DataLandOffBuildingStrategy ();
10             ...
11         }
12     }
13 }
```

Listing 6.5 shows an example of how the factor pattern was used in this project. During the decision making process, as it is described in section 5.4, some decisions rely on the defined strategy of a town. The decisions which building a town needs to upgrade to which level, for example, are based on parameters defined in the Data[...]BuildingStrategy classes. The decision maker gets the strategy of a town (e.g. “land_off”) as a string from the database. Afterwards the decision maker calls the factory method shown in Listing 6.5 with the following statement:

```

$building_strategy = BuildingsStrategyFactory::
    createBuildingStrategy($str);
```

Due to the factory pattern the decision maker does not need to know which class exactly it needs to instantiate. This code is outsourced into the factory method which will return the instantiated object of the corresponding class.

6.2.3 Singleton Pattern

The singleton pattern is, like the factory pattern, a creational pattern. It restricts an application to only allow one instantiated object of a class. Singletons are useful if explicitly one object of a class should be instantiated. An advantage of using the singleton pattern is that it allows a detailed access control of the instance. Since the instantiation process is encapsulated inside of the class itself, the class has full control over how and when an object can be created. However, the singleton pattern is also seen critically in the world of software developers. The fact that the single instance of the class is globally reachable and the instantiation process is invisible from the outside of the class, can make it hard to detect the dependencies of a singleton object [50, 51].

In this project the singleton was used for the class `ZoeEventHandler`. It is an intermediary class between the daemon and the task providers and should only be instantiated once.

Listing 6.6: Example of the Singleton Pattern used in this project

```
1 class ZoeEventHandler
2 {
3     private static $instance;
4
5     public static function getEventHandler()
6     {
7         if (self::$instance === null) {
8             self::$instance = new ZoeEventHandler;
9         }
10        return self::$instance;
11    }
12
13    protected function __construct()
14    {
15        $this->config = new ZoeEventHandlerConfig();
16    }
17
18    protected function __clone()
19    {
20    }
21 }
```

Listing 6.6 shows how the singleton pattern was implemented for the class `ZoeEventHandler`. A static class variable called *\$instance* contains the single instance of the class. The static method *getEventHandler* returns the reference to the instance and instantiates it if it does not exist yet (lazy initialization). The constructor and the copy constructor are declared as protected. This way no object of this class can be instantiated from outside of the class. With the following statement the single instance of the `ZoeEventHandler` can be accessed in every part of the code:

```
$event_handler = ZoeEventHandler::getEventHandler();
```

6.2.4 Transactions

Transactions in the area of database management systems define a group of read and write operations against a database. They are used to make the operations against a database reliable and flawless. Transactions on databases should always follow the ACID pattern. ACID defines the four major properties Atomicity, Consistency, Isolation and Durability that database transactions should have [52].

Atomicity A transaction is atomic if it is executed only if no action fails. If one action of the transaction fails, the entire transaction should fail. This means that a transaction is executed completely but only committed at the very end, when everything worked. Only after the commitment of the transaction, the changes are valid. If one part of the transaction fails, a rollback is done to undo all changes of the transaction.

Consistency Consistency refers to the fact that a transaction needs to leave a valid state of the database behind, after the transaction is completed. Defined rules concerning the structure and content of tables (i.e. data types and values, primary and foreign keys) need to be checked during operations on the database.

Isolation Isolation of transactions is important to prevent that transactions that run concurrently result in a consistent state at the end. This is realized by restricting read and write access to tables or rows of tables that are currently read or written by another transaction.

Durability Durability means that the changes done by a committed transaction should be stored permanently in the database. Even in the case of

system or hardware failure due to i.e. power loss the data should still be available.

Transactions using Zend and PostgreSQL The Zend Framework used in this project provides methods to implement transactions. It enables the developer to use commitment and rolling back to ensure the atomicity of a transaction. Furthermore, PostgreSQL allows to lock specific rows of the database table to prevent concurrent tasks on the same entries of a table.

Listing 6.7: Transactions using the Zend Framework

```
1 $db->beginTransaction();
2 try {
3     $stmt = $db->query("SELECT *
4                         FROM game_player WHERE ID = ?
5                         FOR UPDATE", [1]);
6     // "FOR UPDATE" to lock table row.
7     $player = $stmt->fetchObject();
8     $player->name = "Poseidon";
9     $player->save();
10    $db->commit(); // Commit changes if everything worked.
11 } catch (Exception $e) {
12     $db->rollBack(); // On failure roll back changes.
13 }
```

Listing 6.7 shows an example implementation of a transaction. At the beginning in Line 1 the transaction is started. Following in the try-block a row of the table `game_player` is selected with the parameter “FOR UPDATE”. This sets a lock on the selected row to signalize that this row will be changed by the following code. This way, no other process can update this row until the lock is released. Afterwards, the column “name” of the row is changed and the row is updated with the “save()” method. If everything worked, the transaction is finally saved. If an exception is thrown by the code, the transaction will be rolled back and no changes will be applied to the table.

6.3 The AI

6.3.1 Implementation in the Backend

Integration of the AI in the Backend To emphasize the independence between the AI and the remaining code of the backend, the AI was developed in an own namespace³ *Ig/Grepo/Zoe*. For this project no line of the main Grepolis backend code was changed. This way it could be secured that the development of the AI has no influences on the rest of the game. The implemented solution can be seen as a layer that lays above the APIs of the game. All actions that are executed by the AI are using the APIs that are also used by calls coming from the Frontend⁴.

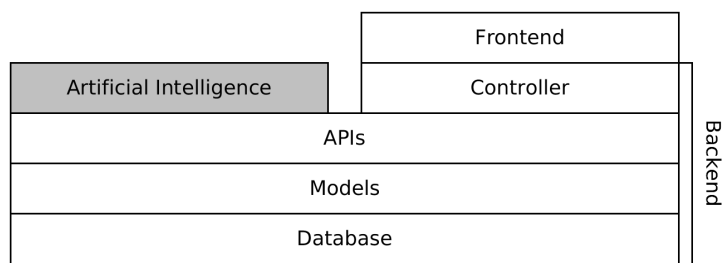


Figure 6.4: Layers of the components of the backend of the game

Data Persistence To be able to use as much of the standard Grepolis code as possible, the NPCs are basically handled as standard player accounts. In the database the same entries are created for the accounts and towns of NPCs as they are for human players. To distinguish the NPCs from the human players a new table was created called *GamePlayerZoeRelation*. This table is a simple 1:1 relation to be able to differentiate NPCs from usual players. This table is used, for example, in the task providers to be able to know which town is an NPC.

³Namespaces can be used in PHP to encapsulate modules of a projects source code.

⁴see section 4.2

6.3.2 Implementation of Decision Making

The implementation of the created concept described in section 5.4 takes advantage of the features of object oriented programming as they are described in section 6.2.1. An abstract class is the base class for all action and decision nodes.

Listing 6.8: Abstract class is parent of decisions and actions

```

1  abstract class AbstractNode
2  {
3      public abstract function decide();
4  }
```

Two classes inherit from the *AbstractNode* class, the classes *Action* and *Decision*. The method *decide* from the *AbstractNode* is declared as abstract and will be defined individually for the child classes. This is a good example of polymorphic methods in OOP. For action nodes the decide method will return the action that needs to be performed.

Listing 6.9: Structure of an action node

```

1  class Action extends AbstractNode
2  {
3      public function decide()
4      {
5          // Return action to perform
6      }
7  }
```

For decision nodes the decide method runs the check which node in the decision tree will be handled next.

Listing 6.10: Structure of a decision node

```

1  class Decision extends AbstractNode
2  {
3      protected $yesNode, $noNode;
4
5      public function decide()
6      {
7          return $this->makeDecision()->decide();
8      }
```

```

9
10 public function makeDecision()
11 {
12     if (...) { // Carry out test
13         return $yesNode;
14     } else {
15         return $noNode;
16     }
17 }
18 }

```

Each decision node implements additionally the *makeDecision* method. In this method the check is done, which node of the decision tree should be checked next. The variables *yesNode* and *noNode* contain the decision or action that follows in the decision tree. A decision tree can then be referenced by its first decision node. By calling the *decide* method of the first decision of a decision tree, the whole tree will be recursively traversed and, at the end, will return the action to perform.

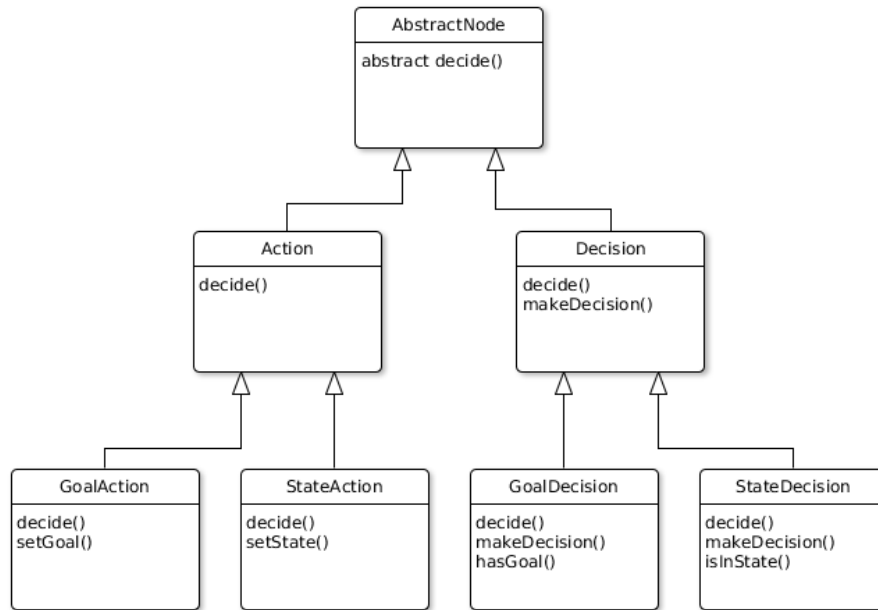


Figure 6.5: Class Diagram of Decisions and Actions

Figure 6.5 shows the complete model of inheritance for decisions and actions. It can be seen that decisions that incorporate goals or states into the decision making process additionally implement either the method *hasGoal* or the method *isInState* from the respective sub class. Similarly, actions that result in goals or state changes have the additional methods *setState* or *setGoal*.

Column Name	Datatype
id	integer
player_id	integer
town_id	integer
state	text

Table 6.1: Table game_zoe_states

Column Name	Datatype
id	integer
player_id	integer
town_id	integer
goal	text

Table 6.2: Table game_zoe_desires

To persist the goals and states, they are stored in individual tables on the database. Table 6.1 and Table 6.2 show the structure of those tables. The columns *player_id* and *town_id* are foreign keys referencing to the tables *game_player* and *game_town*.

6.3.3 Learning and Adaption

As described in section 3.5, an AI can really benefit from learning and adaption in several ways. The following subsections describe how learning and adaption were realized in this project.

6.3.3.1 Learning the Fight System

Each fighting unit in the game has weaknesses and strengths compared to the other units and is either more effective when attacking or defending. It was necessary that the AI gains this knowledge to be able to decide which units can be used in which situation. To be able for the AI to understand the mechanics of the fight system, a learning algorithm was implemented. In this project learning was implemented to allow parameter modification. This means that the outcome of the learning algorithm changes parameters that have an influence on decisions made by the AI. The implemented AI uses *offline learning* to be able to understand the mechanics of the fight system of Grepolis. As described in section 3.5 the offline learning was done during the development of the AI and not during the operation time.

The implemented solution uses statistics about occurred fights and analyzes the result of these fights. The advantage of this method is that the algorithm could

be implemented without having a deep understanding of the implementation of the fight system. An analysis could even be done without any access to the game code itself only by gathering statistics about past fights and analyzing the outcome of the fights.

To have enough statistics about fights a script was written that generates several fights with different units fighting against each other. During the implementation of the script to generate fights, it was important to consider all the dependencies that could have an effect on the outcome of a fight. The building “wall”, for example, increases the defensive values of a town. Since the learning should only consider the strengths and weaknesses of the pure units, all those dependencies needed to be removed so that the results of the fights are not falsified.

Beside the strengths of the units, the costs required to recruit units also needed to be considered during the analysis. The unit “Chariot” for example has a very strong attack value. However, it also costs a lot of resources to recruit. Therefore, it was necessary to check the cost-benefit ratio of the units to find out if the units with, for example, the highest attack value really are the best attacking unit compared to a cheaper unit.

Unit	Resource Cost per Unit	Population Cost	Attack Value
Chariot	(200, 400, 320)	4	56
Slinger	(55, 100, 40)	1	23

Available Resources (1000, 1000, 1000)
Available Population 50

Chariot	Slinger
Possible Amount	Possible Amount
2 (1000 / 400)	10 (1000 / 100)
Attack Value of 2	Attack Value of 10
112 (56 * 2)	230 (23 * 10)

Figure 6.6: Calculation and comparison of the cost-benefit ratio of the unit types Chariot and Slinger

Figure 6.6 shows an example calculation of the cost-benefit ratio of the unit types Chariot and Slinger. The Chariot has a comparatively high attack value of 56 per unit while the Slinger has a lower value of 23. The Chariot, however, costs much more resources to recruit than a Slinger. With a limited amount of

e.g. one thousand resources of each kind one can only recruit two units of the type Chariot. On the other hand, one could recruit up to ten Slingers with the same amount of resources. Multiplying the attack value of one unit with the amount of units one can recruit shows that with Slingers one can obtain more than twice as much attack power.

To consider the above, the script that generates fights for the analysis used a fixed amount of resources and population for the unit creation and did not set a fixed amount of units for each fight. The amount of units for each fight was calculated using the available resources and population. This made it possible to include the cost-benefit ratio into the calculation of strengths and weaknesses of the units.

The statistics of the fights could later be analyzed. This way it was possible to find out which unit is strong in defense, which is strong in offense and which unit has an advantage towards another unit. Table 6.3 shows the best offensive and defensive units for each attack type. A detailed result of the analysis can be found in Appendix B.

	Distance	Blunt	Sharp
Best in Offense	Slinger	Rider	Hoplite
Best in Defense	Swordsmen	Hoplite	Archer

Table 6.3: Extract of the result of learning the fight system

6.3.3.2 Adapting to a Player's Behavior

The AI is able to adapt its behavior to the surrounding players. Using the knowledge gained with the fight system learning described in section 6.3.3.1, the AI is able to adjust the units that are being recruited in the barracks. If a defensive oriented town⁵ of an NPC is, for example, often attacked with units of the attacking type⁶ blunt, it makes sense that the AI focuses on recruiting units that are strong in defending against this type. In this case the AI should recruit more units of the type Hoplite.

To be able to perform this adaption, the AI gathers statistics about the different types of units that players used to attack the town of the AI. These statistics are evaluated and the outcome is used in the decision of what units an NPC should recruit. The algorithm checks what types of units were used and how often were

⁵The adaption described in this section only relates to towns having a defensive strategy (see section 6.3.4).

⁶see section 2.2

they used to attack the town of the NPC. If a specific type of attack was used significantly more often than the other types, the AI will create a new goal⁷ for the NPC to recruit units that are strong in defending against the respective type. This adaption is a good example for parameter modification using *online learning*⁸, as it is done during the operation of the game.

As described in section 3.5, a difficulty with learning in game development is the problem of over-fitting. The adaption used in this project could also suffer from over-fitting. If the AI would make up its decisions about what units to recruit solely based on the adaption described above, this would result in over-fitting. Imagine a player attacking a town only with units of the type blunt. A basic adaption for the AI could be to only recruit units of the type Hoplite, because they are strong in defending against units of the type blunt. However, the possibility that the same player will attack with a different type of attack still exists. Furthermore, other players could also attack the NPC with different types of units. If an NPC only recruits units of a specific type, he will have a strong weakness against the other types of attacks. Therefore, for the decision making process of which units to recruit the AI incorporates additional base unit recruitment goals into the goals created by the adaption algorithm. This way it can be secured that a minimal base amount of each type is available. The AI checks in each iteration if the base amounts of each defensive type is reached and, if not, creates a new goal to recruit units of the corresponding type.

6.3.4 Strategies

Due to the limited resources and the fact that the amount of population per town in Grepolis is limited, it makes sense to specialize towns into e.g. defensive and offensive towns. A town that is oriented offensive, for example, should only recruit offensive units and should not research technologies that are irrelevant for an offensive strategy. Defensive towns, on the other hand, should focus on recruiting defensive units and building a town that is very strong in the defense. Hence the characters controlled by the AI also needed to be able to pursue different strategies with their towns.

Recruitment of units Towns can either be offensive or defensive specialized in terms of warfare. The strategy has effects on the units a town wants to recruit

⁷see section 5.4

⁸see section 3.5

and on the way a town interacts with towns of opponents. An offensive town will try to attack as many opponents as possible (considering the strength of a town) while defensive towns try to build a strong defensive army, which is able to support towns of allies. The parameters found using the learning algorithms described in section 6.3.3.1 were used to define which units are important for a specific kind of strategy. A class was created for each strategy containing the units a town wants to recruit.

Listing 6.11: Example of a data class for a Unit Training Strategy

```

1  class DataLandDeffUnitTrainingStrategy extends
2  DataUnitTrainingStrategy
3  {
4      public function __construct()
5      {
6          $this->ground_units = [
7              GroundUnitsEnum::SWORD => 0.45,
8              GroundUnitsEnum::ARCHER => 0.2,
9              GroundUnitsEnum::HOPLITE => 0.35
10         ];
11     }
12     ...
13 }
```

These classes were used to make the decision which kind of units a specific town wants to recruit or not. As it can be seen in Listing 6.11, the classes contain, among other things, a variable *ground_units* that stores the units a town wants to recruit and the relative amount of the units. During the decision of recruiting units, it can now be checked if the amount of units a town has relatively corresponds to the values stated in the strategy.

Using only the predefined values from the strategies would make the NPCs very predictable. Towns following the same strategy would eventually have the same amount of units. Therefore, the adaption described in section 6.3.3.2 was implemented. This way the decision of recruitment of units was done considering the base values for a strategy coming from the data class and the values from the statistics about previous fights. For offensive towns additionally the attacking goals were considered for the recruitment of units.⁹ During the decision making for recruitment the attacking goals are checked to find out which units are needed

⁹For each town a player wants to attack a new goal is created in the database containing the type of attack (sharp, blunt or distance attack).

to be recruited for the goals. The next paragraph will explain the attacking in greater detail.

Attacking As mentioned before, only offensive towns have the desire to attack other towns. Therefore, the decision tree responsible to make the decision to create new goals to attack other players in the first step checks the strategy of a town and only creates new goals if the strategy of the town is offensive directed (see Figure 6.7).

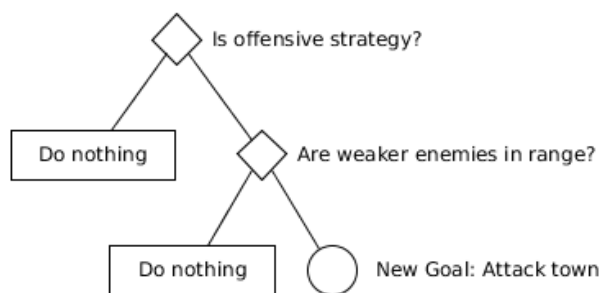


Figure 6.7: Decision tree to create new goals to attack other players

Each attack is either of the type pierce, blunt or distance. During the creation of the new goals to attack other players a type of attack is chosen and associated with this goal. After that, during each iteration of the AI for each attacking goal of an NPC, it could be checked if enough units for an attack are available (see Figure 5.6).

Supporting One of the main tasks of defensive towns is supporting other towns with units. This means that, if another allied town is being attacked, a town with a defensive strategy can send units to this town to support it. This is especially important for offensive towns that are not recruiting any defensive units by themselves. Without support coming from defensive towns offensive towns would be very easy to conquer. Therefore, each NPC checks during each iteration of the AI if one its towns is being attacked. If a town is being attacked and the NPC has defensive town in range¹⁰, the defensive town will send support troops to the town that is being attacked.

¹⁰Being in range means in this context that the supporting troops can reach the city that is being attacked before the attacking troops are arriving.

Construction of towns Similarly to the recruitment of units, also for the construction of the towns, data classes were created. These classes contain information about the maximum level a building should have and about the technologies a town should research. During the process of making the decision which building to upgrade, the actual levels of the buildings could be compared with the values from the strategy. If a building did not reach the maximum level stated in the strategy, an NPC would have the wish to upgrade that building.

Listing 6.12: Example of a data class for a Construction Strategy

```

1  class DataLandDeffBuildingStrategy extends
2  DataBuildingStrategy
3  {
4      public function __construct()
5      {
6          $this->setBuildingLevel(BuildingsEnum::BARRACKS, 30);
7          $this->setBuildingLevel(BuildingsEnum::FARM, 40);
8          $this->setBuildingLevel(BuildingsEnum::LUMBER, 30);
9          $this->setBuildingLevel(BuildingsEnum::IRONER, 30);
10         $this->setBuildingLevel(BuildingsEnum::STORAGE, 30);
11         ...
12         $this->setResearch(ResearchesEnum::PHALANX);
13         $this->setResearch(ResearchesEnum::INSTRUCTOR);
14         ...
15         $this->setImportantBuilding(BuildingsEnum::WALL);
16         ...
17     }
18 }
```

Listing 6.12 shows an example of such a data class. Besides the maximum level of the buildings and the researches of a town the data class can also be used to define important buildings. Buildings marked as important will be upgraded with a higher priority. A defensive town, for example, needs to quickly have a strong wall, to be able defend itself. That is why for the example in Listing 6.12 the wall is set as an important building.

The construction of buildings did not rely completely on the predefined values from the data classes. To be able to adapt to the situation of each town individually, some decisions needed to be made dynamically for each town. A town that has a very low value of free population needs to upgrade the farm with a higher priority to gain more population. Another example are the resource

production buildings. During each decision making iteration the three resource production buildings are checked. It is checked if the production of all three resources is in the same range. If one building produces very few resources compared to the others, this building is upgraded with a higher priority to keep the resource production of all three resources on the same level. Last but not least the building dependencies¹¹ also needed to be considered. During each iteration it is checked if other buildings depend on another building, so that buildings that are the dependency of many other buildings also are upgraded with a higher priority.

6.3.5 Daily Rhythm

Sleeping Rhythm Grepolis is a game that can be played twenty-four hours a day. Players can act or attack other players during the full day. That means people need to be available the whole day and night to be able to react on, for example, incoming attacks. However, a human player will not play the game the whole day. At least during the night, the majority of players are not playing the game and therefore cannot react. That is why it would be an unfair advantage if the AI would act all day and night long. Also players might question the believability of the AI if NPCs would do actions all day long without any breaks. To solve this problem, a daily rhythm for the AI was introduced. The daily rhythm controls when the daemon is allowed to handle tasks and when it should not handle tasks. For a specified amount of time each day the AI will not act in the game. That way a sleeping rhythm can be simulated for the AI characters. Each day for seven hours at night the daemon will not handle any tasks.

A state called “sleeping” was introduced for the AI. The daemon checks in the task provider classes, if the current NPC is in the state of sleeping and, if so, the daemon will not provide any tasks to be executed (see Listing 6.13).

Listing 6.13: The task providers check the state of the NPC

```
1 if ($this->state == ZoeStatesEnum::SLEEPING) {  
2     return [];  
3 } else {  
4     return $tasks;  
5 }
```

¹¹Some buildings depend on other buildings to be on a specific level, before they can be constructed or upgraded.

Unpredictability It was important to make the sleeping rhythm unpredictable for the users. Otherwise players could exploit the rhythm. If the AI would always “go to sleep” at the same time each day, players could detect this very quickly. After knowing at which time the AI cannot react, players could always attack the NPCs at these times. To avoid this risk, the AI does not go to sleep at the same time each day. The implemented solution uses a random algorithm that lets the NPC go to sleep between 22:00 and 24:00 o’clock and lets the NPC wake up between 07:00 and 09:00 o’clock the next morning. Secondly each NPC controlled by the AI goes to sleep at a different time. That makes it even harder to find a pattern in the sleeping rhythm of the AI controlled characters.

6.4 Admin Tool

Grepolis provides a so called Admin Tool that can be used from members of the community management (CM) team to manage the game worlds and the players. Since one of the requirements was that the AI needed to be manageable by the community management, a new section was added to the Admin Tool to manage the AI.

Figure 6.8 shows how the tool can be used to create new NPCs. Before an NPC can be created, a normal player account has to be created since an NPC uses very much of the same logic as the players¹². After that, the AI Tool can be used to assign the created player account to be handled as an NPC. In Figure 6.8 the player account with the id “1” and the name “John Doe” is assigned to be an NPC. When clicking on the “Configure AI” button, one can edit the strategy¹³ of each individual town that this player has. The “Pause” option can be used to pause the decision maker for this NPC. No actions will be executed by the AI for this player for a character that is paused. Lastly the “Unassign” button can be used to remove the relation between the player account and the AI.

¹²see section 6.3.1

¹³see section 6.3.4

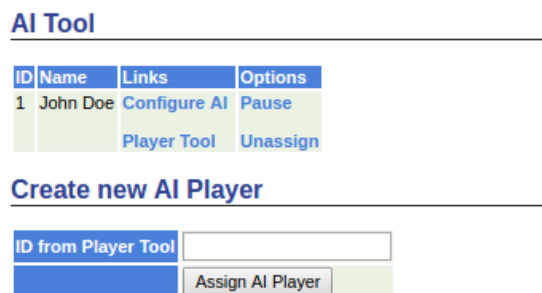


Figure 6.8: Screenshot of the Admin Tool

6.5 Quality Assurance

To assure the correct functionality of the implementation, several unit and integration tests were written. While unit tests check the correct functionality of a single part of the implementation, integration tests are used to test the inter-operation of different parts of the project. Unit tests usually mock¹⁴ dependencies of other classes and integration tests are used to test the real behavior of dependencies. As it is done for the rest of the backend code¹⁵, the tests were written using the PHP library PHPUnit. To perform tests that involve changes to database entries, Grepolis provides a test database. This database is set up before every run of the tests and creates a mirror of the table structure of the production database. Additionally, default entries for testing, like several players with towns, are created during the set up.

Listing 6.14: Example of the use of the `assertEquals` method in a unit test

```

1 $task_provider = new GameTimedZoeCallTaskProvider();
2 $tasks = $task_provider->findDue();
3 $this->assertEquals($expected_amount, count($tasks));

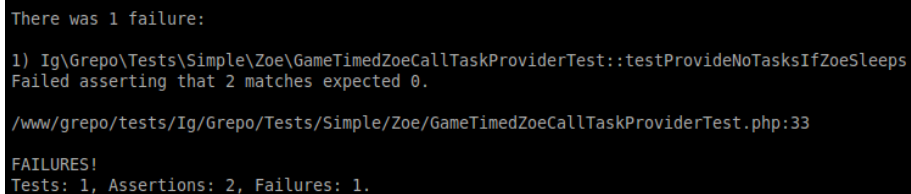
```

PHPUnit provides different methods to check, for example, that a variable contains an expected value, or that a condition is met. Listing 6.14 shows an extract of a unit test that makes use of the method `assertEquals` to test the behavior of the `GameTimedZoeCallTaskProvider`. As mentioned in section 6.3.5, this task provider should only gather tasks for NPCs that are awake. This test can be

¹⁴A mock object is an object that simulates the behavior of another object. In unit testing it is used to reduce the complexity and to remove dependencies to other classes.

¹⁵see section 4.2.2

used to check if the behavior works correctly. The `assertEquals` method in line three compares the number of found tasks with an expected amount. So if, for example, there are four tasks due, but two of them belong to an NPC that is currently sleeping, the expected amount should be two.

A screenshot of a terminal window showing the output of a failed unit test. The text is as follows:

```
There was 1 failure:  
1) Ig\Grepo\Tests\Simple\Zoe\GameTimedZoeCallTaskProviderTest::testProvideNoTasksIfZoeSleeps  
Failed asserting that 2 matches expected 0.  
  
/www/grepo/tests/Ig/Grepo/Tests/Simple/Zoe/GameTimedZoeCallTaskProviderTest.php:33  
  
FAILURES!  
Tests: 1, Assertions: 2, Failures: 1.
```

Figure 6.9: Output of a failed unit test

If a test fails, PHPUnit will produce an output similar to the one in Figure 6.9. The line “*Failed asserting that 2 matches expected 0.*” signifies that an assertion failed in the given test case. Several test scenarios were created and implemented. The whole list of all tests can be found in Appendix C.

Chapter 7

Conclusion

This chapter concludes the project. It describes the process of evaluating the AI, as well as summarizing the results of the project, and it finally gives an outlook for the implemented solution.

7.1 Evaluating the AI

After completion of the implementation, an evaluation was done to analyze the implemented solution. The AI should be evaluated in different ways to find weaknesses of the implementation, to evaluate the robustness of the created model for AI and to check if the solution works as desired. This process is described in the following subsections.

7.1.1 Horizontal and vertical Robustness

To test the robustness of the AI a load test was realized. The test checks if the AI is still working under extreme conditions. Two tests were executed to test the vertical and horizontal robustness of the AI concerning the amount of NPCs and cities it has to handle. To test the vertical robustness, a scenario was created, where the AI has to handle one NPC which has a large number of cities under his control. The average amount of cities players have under their control is around 10-20 depending on the game stage. Several evaluation cases were done with incremental increasing amount of towns the NPC has. The horizontal robustness was evaluated by creating a scenario, where the AI processes very many NPCs that each have one city to control. Similar to the

Evaluation case	No. of NPCs	No. of Towns per NPC	Avg. execution time
Horizontal 1	1	5	0.022 s
Horizontal 2	1	10	0.036 s
Horizontal 3	1	50	0.055 s
Horizontal 4	1	100	0.131 s
Vertical 1	500	1	0.5 s
Vertical 2	1000	1	0.7 s
Vertical 3	4000	1	1.3 s
Vertical 4	6000	1	1.42 s
Mix 1	1	1	0.001 s
Mix 2	50	20	0.81 s
Mix 3	100	20	1.35 s
Mix 4	500	20	1.56 s

Table 7.1: Evaluation cases for the horizontal and vertical robustness

vertical evaluation, several scenarios were evaluated with different numbers of NPCs. Since the average amount of active players per game world lies between 1000 and 4000, the values for the test cases are located in that range. Finally, a third scenario was created with a mixed amount of numbers of NPCs and towns per NPC. Each scenario ran for thirty minutes. During this time the execution time of each cycle of the daemon loop was recorded so that an average could be calculated afterwards.

The result of the test was that both, horizontally and vertically, no issues exist. That means that the AI can be used to create a large amount of NPCs without any problems. However, it has to be mentioned that the execution time of one cycle of the daemon increases, if the AI has to run the decision makers of many NPCs at the same time. If the AI has to handle the decision makers of one thousand NPCs at the same time, the execution time of one cycle can get up to more than twenty seconds. It is, however, very unlikely that all NPCs need to make decisions at the same time. As Table 7.1 shows, the average execution time does not increase very much with a higher amount of NPCs. Even the last evaluation test “Mix 4” where 500 NPCs each have 20 towns, which makes a total of 10.000 towns, has an acceptable average execution time of 1.56 seconds per cycle¹.

¹The execution time may differ depending on the server that runs the AI.

7.1.2 Long-Term Evaluation

To evaluate if the AI can be run over a long period of time, a long-term evaluation was done. One NPC was set up with two cities under his control and the daemon responsible for handling the AI was run for one week in a development environment without a break. The two towns of the NPC had different strategies assigned to them. One was supposed to be an offensive oriented town and the other one was a defensive town. At the beginning of the test, the two cities were in a basic state with only the minimum of buildings existing to start the game.

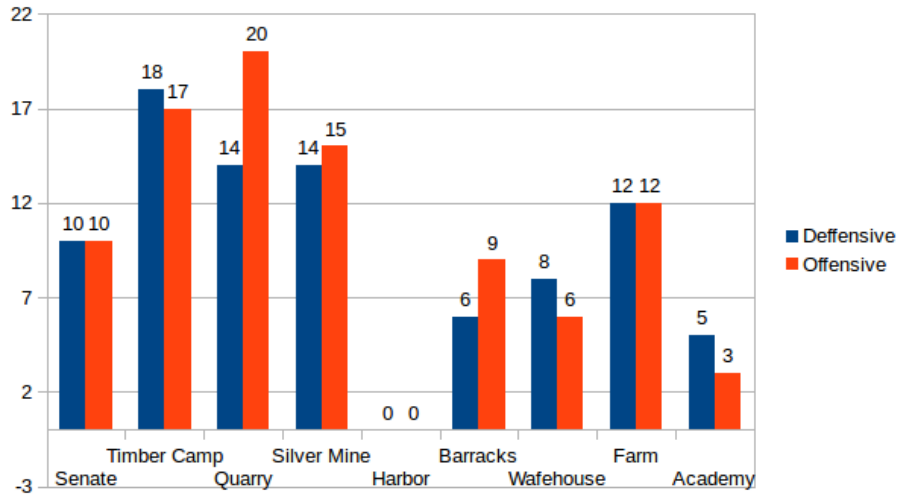


Figure 7.1: Comparison of the cities of a defensive and offensive town of an NPC after one week

After one week the daemon was stopped and the results of the work of the AI could be analyzed. Fortunately, the AI ran the whole week without any issue coming up. This shows that a long-term use of the AI is possible. As Figure 7.1 shows, after one week both cities were constructed according to the needs. In the start phase of the game the difference between offensive and defensive towns concerning the constructed buildings does not differ very much. But still some differences are visible. The offensive town, for example, has the Quarry² four levels higher than the defensive town. This is due to the fact that the

²The Quarry is the building that produces the resource stone.

	Stone	Wood	Silver
Swordsman	0	95	95
Archer	0	120	75
Slinger	100	55	40

Table 7.2: Comparison of the resources needed for the units Swordsman, Archer and Slinger

offensive town concentrated on recruiting the unit Slingers while the defensive town mainly recruited Swordsmen and Archers. Since one only has to spend the resources wood and iron to recruit Swordsmen and Archers, but for Slingers one also has to spend stones, it makes sense that the offensive town has a further constructed Quarry, because the demand for stone is higher in that town. Table 7.2 shows the resources needed to recruit the three types of units.

One finding of the long-term run was that the NPC often had very long periods of waiting, where nothing was possible to do. If the NPC has a lack of resources, it might be possible that he cannot do anything and has to wait, until he has gathered enough resources to continue upgrading the buildings or recruiting units. Of course, also human players are in the same situation and have to manage the use of their resources carefully. However, the players can use game features that are not implemented in the current AI to obtain additional resources. Features like Farming Villages³ or Island Quests⁴ can be used by the players to shorten the waiting time.

7.1.3 Simulation of Live Worlds

Since one of the requirements was to be able to use the AI to simulate Live Worlds in the development environment, one part of the evaluation was to find out if the AI meets this expectation. Therefore, an evaluation was done that should compare the behavior of a server populated with several NPCs and the behavior of a live world. The actions taken by players on a live world (EN92) were logged over a time frame of one day. Simultaneously in a development environment a scenario was set up that was similar to the conditions of the live world concerning game configurations (e.g. the speed of the world). The chosen live world had an active user base of around 4100 players per day. Therefore, the development world was also set up with 4100 NPCs. The actions taken by

³Farming Villages are villages on the islands that can be raided to gather resources.

⁴Island Quests are tasks a player has to solve to get a reward like, for example, resources.

the NPCs were also logged. Afterwards the results of both statistics could be compared.

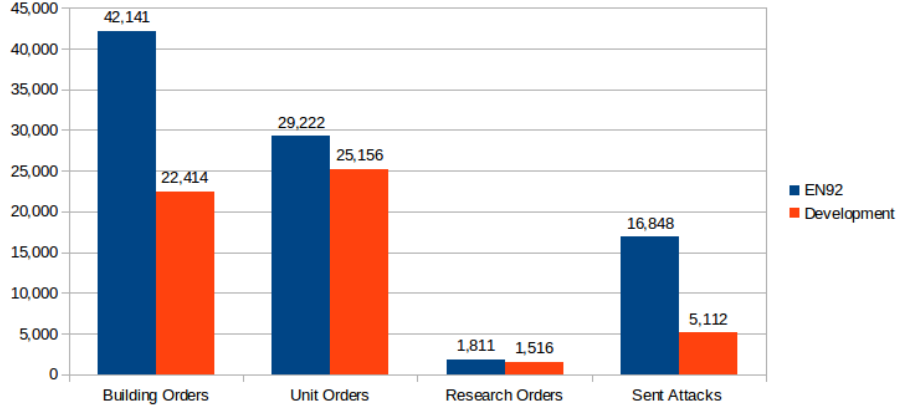


Figure 7.2: Comparison of actions on a live world versus a simulated live world

Figure 7.2 shows that the NPCs are able to produce a significant amount of actions. However, it cannot completely achieve the amount of actions of a live world. This could be caused due to the fact explained in section 7.1.2. The NPCs cannot use all the features like human players to obtain resources. Therefore, they have a longer waiting period between their actions. Additionally, with the current solution it is not possible to simulate the dynamics happening on a live world between the players. If, for example, two rivaling alliances clash with each other, this can result in a massive increase of attacks and unit orders for a specific time.

Another issue found during the evaluation was that no HTTP requests will be sent to the servers due to the way the AI is implemented. Usually each action taken by a user results in an HTTP request sent to one of the web servers. While the web servers of the chosen live world had to handle more than 2.5 million requests in the twenty-four hours, the simulated live world had to handle no requests at all. Since the AI is integrated directly into the backend, no requests are needed for the actions taken by the NPCs. Thus, the simulation cannot be used to simulate the behavior of many requests coming to the web servers.

7.1.4 Individuality of Decisions

An evaluation was done to check the individuality of decisions made by the NPCs. Two NPCs in exactly the same situation should optimally behave dif-

ferently, if there are different possibilities of what to do. This is important to increase the believability of the AI from the player's perspective. To check how individual the decision making process is, a scenario was created, where two NPCs are in exactly the same situation. They each have one city with buildings upgraded to a specific level, they have the same amount of units in the barracks and they have the same opponents on their island. The scenario was run for one hour and, after that, the resulting actions of the two NPCs were compared.

	NPC 1	NPC 2
1	Research Technology Hoplites	Train Units 2 Slinger
2	Upgrade Building Silver mine	Research Technology Hoplites
3	Train Units 5 Hoplites	Upgrade Building Barracks
4	Upgrade Building Marketplace	Train Units 5 Hoplites
5	Train Units 6 Slingers	Attack Town 22
6	Attack Town 13	Upgrade Building Silver mine
7	Upgrade Building Cave	

Table 7.3: Sequence of actions performed by two NPCs in the same situation

As Table 7.3 shows, the two NPCs performed different actions in the scenario. Although some actions, like researching the technology Hoplites, training units or upgrading the Silver mine, were the same, each NPC also performed actions, that the other NPC did not perform. The fact that two NPCs in exactly the same initial situation behave differently makes it hard for the players to identify a pattern in the decisions made by the AI. This can increase the believability of the AI, since it gives each NPC more individuality.

7.2 Results

Implemented Features The implemented solution meets the requirements specified at the beginning of the project. The AI can be used to create NPCs in the game. The NPCs controlled by the AI are able to perform all essential game features. NPCs can construct cities, including researching of technologies and

upgrading of buildings, based on predefined static parameters combined with dynamically changing parameters depending on the game situation. Furthermore, the NPCs can recruit units based on a predefined strategy that incorporates the current situation of the character and NPCs are able to attack players in the proximity. Lastly, the NPCs are able to cooperate between their owned cities by sending supporting troops.

Usage of the AI The AI can be used as an in-game feature by placing an NPC on an island inhabited by players. The NPC will independently construct its town and attack the human players or defend its city based on the chosen strategy. Players can ally to defeat the NPC together or they could even ally with the NPC to protect him from attacks from other players.

Secondly, the AI can be used to simulate some of the real conditions of the live servers. Although the usage of the AI will not produce any HTTP requests to the web server, like a normal player would do⁵, the actions performed by the AI will produce load to the database servers and also to the daemons of the game. Such a simulation could help the system integration team to discover bottle necks of the server architecture and could also help the game development team to resolve bugs that only occur on live servers.

Finally, the AI could be used to perform so called A/B testing. One could set up different worlds with different game configuration settings, like the speed of the world, or the base amount of resources being produced and let the AI run several NPCs on each world. By analyzing the progress of the NPCs on the different worlds one could find out what consequences the different settings have.

AI Model The way the AI was implemented allows an easy further development. Since the AI uses a modular model⁶, new game features can be added without the need to touch the existing logic. A new module only needs to be included either in the `CharacterDecisionMaker` or the `StrategyDecisionMaker`. Furthermore, single modules can be deactivated to change the behavior of the AI. If, for example, the decision is made to allow cheating for the AI⁷ so that the AI does not need to construct its town, the module responsible for the construction of a town can simply be deactivated.

⁵see section 7.1.3

⁶see section 5.3

⁷see section 3.3

Decision Making The combination of the three decision making techniques decision trees, state machines and goal-oriented behavior provides an easy to understand solution that autonomously produces decisions depending on the current situation of a character. Learning and adaption is implemented to increase the individuality of each NPC. The evaluation described in section 7.1 shows that the decisions of different NPCs in the same situation differ. This makes it harder for players to identify a pattern in the actions of the AI and will help to increase its believability.

7.3 Limitations

Competitiveness The game features that the NPCs are able to use are restricted to the essential game features. Features like Farming Villages or usage of gods are not implemented. Many game features that are not implemented help the players to gather extra resources. As section 7.1.2 shows, the lack of resources is the main problem of the AI. While human players can obtain extra resources by trading, or interacting with Farming Villages, the NPCs can only wait until the resources are produced in the resource production buildings. Therefore, the human players have a great advantage and can progress faster in the game compared to the NPCs. Secondly, as long as the NPCs are not able to conquer the cities of a player, the NPCs are not able to defeat human players. The question, if it is desired that the NPCs are able to defeat a player needs to be answered individually.

Strategies of Towns The strategies that an NPC uses to specialize his town⁸ need to be set manually for each town in the Admin Tool. It would be better if the NPCs are able to select the desired strategy of a town by himself depending on the current situation of the game. This can make it easier for the NPC to adapt to a changing situation in the game.

Collaboration of Decision Makers As described in section 5.3 the different modules of the decision making process return their results to either the CharacterDecisionMaker or the StrategyDecisionMaker. In these main decision makers all the actions that an NPC wants to perform are gathered. However, usually not all the desired actions can be performed due to lack of resources or units.

⁸see section 6.3.4

Therefore, a decision needs to be made what actions should be performed in the current situation. In the implementation of this project the decision what actions to perform is simply done randomly. All actions are shuffled and executed in a random order until all resources or available units are used. The randomness ensures that different NPCs in exactly the same situation will not perform the same actions and also makes sure that all different types of actions are executed with the same probability. Since each iteration is independent from the previous iteration, the probability for each action to be performed is always defined by the formula for the discrete uniform distribution[53] $P_{(Action)} = \frac{1}{N}$, where N is equal to the number of actions to be performed. However, using randomness causes that a decision that might be more important in a specific situation is not performed due to lack of, for instance, resources. An improvement would be to incorporate the current situation of the NPC into the evaluation of the actions to be performed. This way important actions could be executed with a higher prioritization, before other actions, that might be less important in the current situation, are performed.

7.4 Outlook

The result of the project can be used to create NPCs in the game Grepolis. However, there is still room to improve the solution and to further develop it.

Implementing more Game Features Though the project can be seen as a success, it still can be further developed to improve the solution. As described in section 7.2 the NPCs are able to perform all essential game features. Nevertheless, there are still some game features that the NPCs cannot perform. One major feature that the NPCs would need to be able to defeat human players is the ability to conquer a player's town. Additionally, features like interacting with Farming Villages or the usage of gods could be implemented to make the NPCs more competitive. The ability to chat with human players could also be implemented as an additional feature.

Dynamic Selection of Strategies In the current implementation the strategies that define the behavior of a town of an NPC need to be set manually in the Admin Tool. To make the AI more autonomous, it would be a good idea to let the NPC choose the strategy for his towns by himself. This way the NPC could select the right strategy for each town depending on the current situation. Also

changing the strategy of a town dynamically could be implemented to allow the NPC to adapt to different game situations.

Improve Admin Tool The implemented admin tool described in section 6.4 could also be improved. By adding an overview of the goals a character currently has, or by visualizing the state of a character the community managers could get a better overview about the current situation of an NPC.

Improve Learning and Adaption The AI could be improved by adding more learning and increasing the level of adaption. The AI could adapt more to its current situation by evaluation who attacked an NPC and how often. This way the NPCs could identify aggressive players that might be a threat to the NPC and try to concentrate on attacking them. Additionally, the decision trees that were created manually could be replaced with decision trees created with a learning algorithm like ID3[27]. To be able to use ID3 an analysis would be needed beforehand to gather observations and actions of specific game situations. Learned decision trees might help to let the decisions of the NPCs be closer to the decisions of a human player in a similar situation.

Integrate AI into other Game Features The AI, as a single module of Grepolis, could be integrated in other modules of the game. It could be used, for example, in game events to make them more complex. An event, where players have to ally against an AI controlled NPC, could be implemented to increase the activity of the players. Another possibility would be to use the AI as a “New-Player-Protection”. NPCs could be placed on islands, where new players start the game and help by supporting them with troops or resources to make the start phase of the game easier.

Separate AI from Backend To be able to have a real simulation of a human player, it would be needed to separate the AI completely from the backend code. If the AI is located on a different server than the backend of Grepolis, the NPCs need to perform standard HTTP requests, like human players do, to perform actions in the game. This way all modules of the Grepolis backend would be integrated in a simulation of a live world with NPCs.

Reuse of the AI Model The results show that the developed model was successful. The model for the AI and for the decision making is a functional

solution that is not restricted to Grepolis. Therefore, it could also be reused and applied to other game development projects with a similar game play as Grepolis. It is a model that can be applied to different projects where decision making is needed.

Bibliography

- [1] InnoGames. Innogames: Fast facts, 03 2016. <https://corporate.innogames.com/en/press/fast-facts.html> (Accessed on 18.03.2016).
- [2] InnoGames. Innogames: History, 2016. <https://corporate.innogames.com/en/press/history.html> (Accessed on 18.03.2016).
- [3] InnoGames. Buildings portal, 13 2013. https://wiki.en.grepolis.com/wiki/Buildings_Portal (Accessed on 16.06.2016).
- [4] K. Schwaber and J. Sutherland. *The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game*. Scrum.Org and Scrum Inc, 2013.
- [5] H. Kniberg and M. Skarin. *Kanban and Scrum - Making the Most of Both*. InfoQ.com, 2009.
- [6] Trello. Trello, 2016. <https://trello.com/> (Accessed on 22.03.2016).
- [7] D. Leffingwell. Levels of stakeholder involvement. In *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, page 121. Addison-Wesley Professional, 2010.
- [8] Corinne N Johnson. The benefits of pdca. *Quality Progress*, 35(5):120, 2002.
- [9] Git. git –distributed-even-if-your-workflow-isnt, 2016. <https://git-scm.com/> (Accessed on 19.03.2016).
- [10] S. Chacon and B. Straub. Chapter 3 - git branching. In *Pro git - second edition*, page 43. Apress, 2014.
- [11] M. Buckland. Introduction. In *Programming Game AI by Example*, page xix. Wordware Pub Co, 2005.

- [12] K. Dill. What is game ai? In S. Rabin, editor, *Game AI Pro*, page 8. CRC Press, 2013.
- [13] Alexander Nareyek. Ai in computer games. *Queue*, 1(10):58–65, February 2004.
- [14] I. Millington. Game ai. In *Artificial Intelligence for Games*, pages 21–23. Morgan Kaufmann Publishers, 2006.
- [15] Principles behind the agile manifesto. <http://agilemanifesto.org/principles.html> (Accessed on 27.04.2016).
- [16] B. Scott. The illusion of intelligence. In *AI Game Programming Wisdom*, pages 16–20. Charles River Media, 2002.
- [17] Games Workshop Limited. Warhammer 40,000: Dawn of war iii. <https://www.dawnofwar.com/> (Accessed on 19.06.2016).
- [18] Kotaku. The three (or more, or less) laws of gaming ai, 05 2009. <http://kotaku.com/5271733/the-three-or-more-or-less-laws-of-gaming-ai> (Accessed on 20.05.2016).
- [19] Take-Two Interactive Software, Inc. Sid meiers civilization. <http://franchise.civilization.com/en/home/> (Accessed on 19.05.2016).
- [20] Take-Two Interactive Software, Inc. 2k games. <https://www.2k.com/> (Accessed on 19.06.2016).
- [21] TVTropes. The computer is a cheating bastard. <http://tvtropes.org/pmwiki/pmwiki.php/Main/TheComputerIsACheatingBastard> (Accessed on 20.05.2016).
- [22] I. Millington. Overview of decision making. In *Artificial Intelligence for Games*, pages 301–303. Morgan Kaufmann Publishers, 2006.
- [23] R. Davis and J. J. King. The origin of rule-based systems in ai. In B. G. Buchanan and Edward H. Shortliffe, editors, *Rule-Based Expert Systems*, pages 20–25. Addison-Wesley Publishing Co., Inc., 1984.
- [24] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. Trees. In *Data Structures and Algorithms*, pages 91–96. 2001. [http:](http://)

- [//dlia.ir/Scientific/e_book/Science/Mathematics/QA_75.5_76.95_Electronic_Computers_Computer_Science_/008394.pdf](http://dlia.ir/Scientific/e_book/Science/Mathematics/QA_75.5_76.95_Electronic_Computers_Computer_Science_/008394.pdf) (Accessed on 01.07.2016).
- [25] M. Sipser. Finite automata. In *introduction to the theory of computation*, pages 31–40. Thomson Course Technology, Boston, Massachusetts, USA, 2006.
- [26] J. Orkin. Applying goal-oriented action planning to games. *Mono-lith Productions*. http://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf (Accessed on 11.07.2016).
- [27] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
- [28] I. Millington. Learning. In *Artificial Intelligence for Games*, page 564. Morgan Kaufmann Publishers, 2006.
- [29] I. Millington. Over-learning. In *Artificial Intelligence for Games*, page 566. Morgan Kaufmann Publishers, 2006.
- [30] N. Palmer. Machine learning in games development. <http://ai-depot.com/GameAI/Learning.html> (Accessed on 21.07.2016).
- [31] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O’Connor, and S. Pfeiffer. Html 5 - a vocabulary and associated apis for html and xhtml, 10 2014. <https://www.w3.org/TR/html5/> (Accessed on 27.06.2016).
- [32] Khronos WebGL Working Group. WebGL public wiki, 04 2015. https://www.khronos.org/webgl/wiki/Main_Page (Accessed on 27.06.2016).
- [33] Unity Technologies. Create and connect with unity, 2016. <http://unity3d.com/> (Accessed on 27.06.2016).
- [34] Adobe Systems Incorporated. Adobe air, 2016. <http://www.adobe.com/products/air.html> (Accessed on 27.06.2016).
- [35] The jQuery Foundation. jquery, 2016. <https://jquery.com/> (Accessed on 15.08.2016).
- [36] J. Ashkenas. Javascript’s utility _belt, 07 2016. <https://github.com/jashkenas/underscore> (Accessed on 15.08.2016).

- [37] Zend. Zend framework, 2016. <https://framework.zend.com/> (Accessed on 30.06.2016).
- [38] N. Adermann and Jordi Boggiano. Composer - dependency manager for php, 2016. <https://getcomposer.org/> (Accessed on 30.06.2016).
- [39] S. Bergmann. Welcome to phpunit, 2016. <https://phpunit.de/> (Accessed on 30.06.2016).
- [40] PostgreSQL Global Development Group. Postgresql, 2016. <https://www.postgresql.org/> (Accessed on 27.06.2016).
- [41] NGINX Inc. nginx, 2016. <http://nginx.org/> (Accessed on 30.06.2016).
- [42] B. Wil. Github repository, 03 2016. <https://github.com/BrandonWill/GrepolisBot> (Accessed on 30.03.2016).
- [43] M. Gessler. Interessengruppen/interessierte parteien. In *Kompetenzbasiertes Projektmanagement*, page 71. Deutsche Gesellschaft fuer Projektmanagement, 2011.
- [44] D. Leffingwell. Goodbye iron triangle. In *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, pages 16–17. Addison-Wesley Professional, 2010.
- [45] The PHP Group. Php: What is php? - manual. <http://php.net/manual/en/intro-what-is.php> (Accessed on 22.04.2016).
- [46] What language do you use to create your ai programs and why? *KI - Kuenstliche Intelligenz*, 26(1):99–106, 2011.
- [47] I. Millington. My model of game ai. In *Artificial Intelligence for Games*, pages 9–11. Morgan Kaufmann Publishers, 2006.
- [48] I. Millington. State machines. In *Artificial Intelligence for Games*, page 341. Morgan Kaufmann Publishers, 2006.
- [49] B. Lahres and G. Rayman. Die basis der objektorientierung. In *Objektorientierte Programmierung*. Rheinwerk Computing, 2009.
- [50] M. Hevery. Root cause of singletons. 08 2008. <http://misko.hevery.com/2008/08/25/root-cause-of-singletons/> (Accessed on 04.08.2016).

- [51] M. Hevery. Singletons are pathological liars. 08 2008. <http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/> (Accessed on 04.08.2016).
- [52] T. Kyte and D. Kuhn. Transactions. In *Oracle Database Transactions and Looking Revealed*, pages 79–109. Apress Media, 2014.
- [53] H.-O. Georgii. Diskrete gleichverteilung. In *Stochastik: Einführung in die Wahrscheinlichkeitstheorie und Statistik*, pages 29–30. Walter de Gruyter GmbH & Co KG, 2015.

Nomenclature

AI	Artificial Intelligence
API	Application Programming Interface
CIP	Continuous Improvement Process
CM	Community Management
CSS	Cascading Style Sheets
DAU	Daily Active Users
DHTML	Dynamic HTML
F2P	Free-to-Play
GOB	Goal-Oriented Behavior
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
NPC	Non-Player Character
OOP	Object Oriented Programming
PID	Process Identifier

List of Figures

1.1	Percentage of login activity in weeks after registration	2
2.1	Screenshot of the game showing an island with several towns . . .	5
2.2	Screenshot of the game showing an advanced town	6
2.3	Screenshot of the game showing the unit swordsman in the barracks	7
2.4	The building queue as a standard (1) and premium (2) user . . .	8
2.5	Example of a Kanban board	9
2.6	Plan-Do-Check-Act (PDCA) Cycle	11
3.1	Schema of a general decision making process (adapted from [22])	17
3.2	Example of a rule-based system	18
3.3	Example of a decision tree	19
3.4	Example of a state machine for a game AI	20
4.1	Overview of the call flow for an API	26
4.2	Scheme of the Server Architecture of a Grepolis market	28
4.3	Screenshot of the tool GrepolisBot	29
4.4	Stakeholder Matrix	30
4.5	Flipping of the iron triangle with agile development (adapted from [44])	31
5.1	Developed model of the AI	38
5.2	Simplified AI Model for an AI entity with several towns	39
5.3	Example of modules for the Character AI	39
5.4	A state machine with two very similar transitions	40
5.5	Combination of a state machine with a decision tree	42
5.6	Combination of decision trees, state machines and goal-oriented behavior	43

6.1	Task providers and handlers are called in the daemon loop	45
6.2	Database table game_timed_zoe_call	46
6.3	Example of inheritance of classes	48
6.4	Layers of the components of the backend of the game	54
6.5	Class Diagram of Decisions and Actions	56
6.6	Calculation and comparison of the cost-benefit ratio of the unit types Chariot and Slinger	58
6.7	Decision tree to create new goals to attack other players	62
6.8	Screenshot of the Admin Tool	66
6.9	Output of a failed unit test	67
7.1	Comparison of the cities of a defensive and offensive town of an NPC after one week	70
7.2	Comparison of actions on a live world versus a simulated live world	72

Listings

4.1	Example of the parameters of an API call from frontend to backend	27
6.1	Implementation of a daemon loop in PHP	45
6.2	Execution of the timed calls	46
6.3	Example of encapsulation of attributes and methods	48
6.4	Example of polymorphism	49
6.5	Example of the Factory Pattern used in this project	50
6.6	Example of the Singleton Pattern used in this project	51
6.7	Transactions using the Zend Framework	53
6.8	Abstract class is parent of decisions and actions	55
6.9	Structure of an action node	55
6.10	Structure of a decision node	55
6.11	Example of a data class for a Unit Training Strategy	61
6.12	Example of a data class for a Construction Strategy	63
6.13	The task providers check the state of the NPC	64
6.14	Example of the use of the assertEquals method in a unit test . .	66

List of Tables

4.1	Milestones	34
6.1	Table game_zoe_states	57
6.2	Table game_zoe_desires	57
6.3	Extract of the result of learning the fight system	59
7.1	Evaluation cases for the horizontal and vertical robustness	69
7.2	Comparison of the resources needed for the units Swordsman, Archer and Slinger	71
7.3	Sequence of actions performed by two NPCs in the same situation	73

Appendix A

Requirements Analysis Document

Description Appendix A contains the requirement analysis document, which was created after the requirements analysis, to document the found requirements.

Requirements Analysis Document

Version 1.0

30.03.2016

Created by Lars Engel

Table of contents

1	Introduction.....	
1.1	Purpose of the System.....	
1.2	Core System Functionalities.....	
1.3	Objectives and Success Criteria of the Project.....	
1.4	Definitions, Acronyms and Abbreviations.....	
2	Current System.....	
3	Requirements.....	
3.1	Main Story.....	
3.2	Functional Requirements.....	
3.3	Nonfunctional Requirements.....	
	Availability.....	
	Implementation.....	
	Integrity.....	
	Performance.....	
	Reliability.....	
	Supportability.....	
	Usability.....	
3.4	Essential Requirements.....	
3.4.1	Essential Requirements.....	
3.4.2	Additional Requirements.....	
4	Change History.....	
5	References.....	

1 Introduction

1.1 Purpose of the System

To improve the long term play value of Grepolis and to be able to create new and more complex events an intelligent non-player character (NPC) has to be implemented.

1.2 Core System Functionalities

The implementation must be able to:

- Build one or several towns independently.
- Do researches in the academy independently.
- Attack other players independently.
- Defend itself independently.
- Allow a management of the non-player characters via the admin tool.

The solution should furthermore be able to be used to simulate human players in a development environment to be able to

- Perform load tests
- Simulate live worlds
- Find bottle necks in the server architecture.

1.3 Objectives and Success Criteria of the Project

The project can be seen as successful if the core system functionalities stated in 1.2 are implemented. Since the project is developed in an agile environment the requirements stated in this document may change at any point during the project.

1.4 Definitions, Acronyms and Abbreviations

Admin Tool	The tool used by CMs to manage game functionalities.
CM (Community Manager)	A person managing the community of players
Independently	In this context the word “independently” means that an NPC is making his own decisions without the need to tell him what to do.
NPC (non-player character)	A non-human entity behaving like a person playing the game
Player	A person playing Grepolis
Product Owner	A person who is responsible for the game.
System Integrator	A person who is responsible for managing the game servers and the other components that run the game.
The game	The online multiplayer game Grepolis

2 Current System

There is no current system in place right now that acts as an NPC in the game Grepolis.

3 Requirements

3.1 Main Story

As a player I want to be able to interact with a non-player character so that I have a better game play experience. (See page 6 for splitting up main story into player relevant sub stories)

3.2 Functional Requirements

F-P1: As a player I want the NPC to do researches in the academy so that I have a worthy enemy.

F-P2: As a player I want the NPC to upgrade his towns buildings so that I have a worthy enemy.

F-P3: As a player I want the NPC to defend itself on an upcoming attack so that I have a worthy enemy.

F-P4: As a player I want the NPC to attack other players so that I have a better game player experience.

F-P5: As a player I want the NPC to recruit units so that I have a worthy enemy.

F-P6: As a player I want the NPC to interact with farming villages so that I have a worthy enemy.

F-P7: As a player I want the NPC to use heroes so that I have a worthy enemy.

F-P8: As a player I want the NPC to trade resources so that I have a worthy enemy.

F-P9: As a player I want the NPC to colonize towns so that I have a worthy enemy.

F-P10: As a player I want the NPC to use spells so that I have a worthy enemy.

F-P11: As a player I want the NPC to solve island quests so that I have a worthy enemy.

F-CM1: As a community manager I want to create new NPCs via the admin tool.

F-CM2: As a community manager I want to be able to manage existing NPCs via the admin tool.

F-SI: As a system integrator I want to have a simulation of a human player so that I am able to do server load tests.

3.3 Nonfunctional Requirements

Availability

NFA-1: As a player I want the NPC not to be available all the time so that I have a better game play experience.

Implementation

NFI-1: As a developer I want to be able to work on the system without spending much time in working into the used technology so that I can quickly understand the system.

Integrity

NFIT-1: As a product owner I don't want that the system can be misused to cheat in the game.

Performance

NFP-1: As a player I don't want that the performance of the game is worsened with this new feature so that I can play the game as before.

Reliability

NFR-1: As a product owner I want the system to be tested during the implementation and integration.

Supportability

NFS-1: As a system integrator I want to be able to run the system on the current server setup so that minimal extra configuration is needed.

Usability

NFU-1: As a player I want to be able to distinguish NPCs and other players.

3.4 Essential Requirements

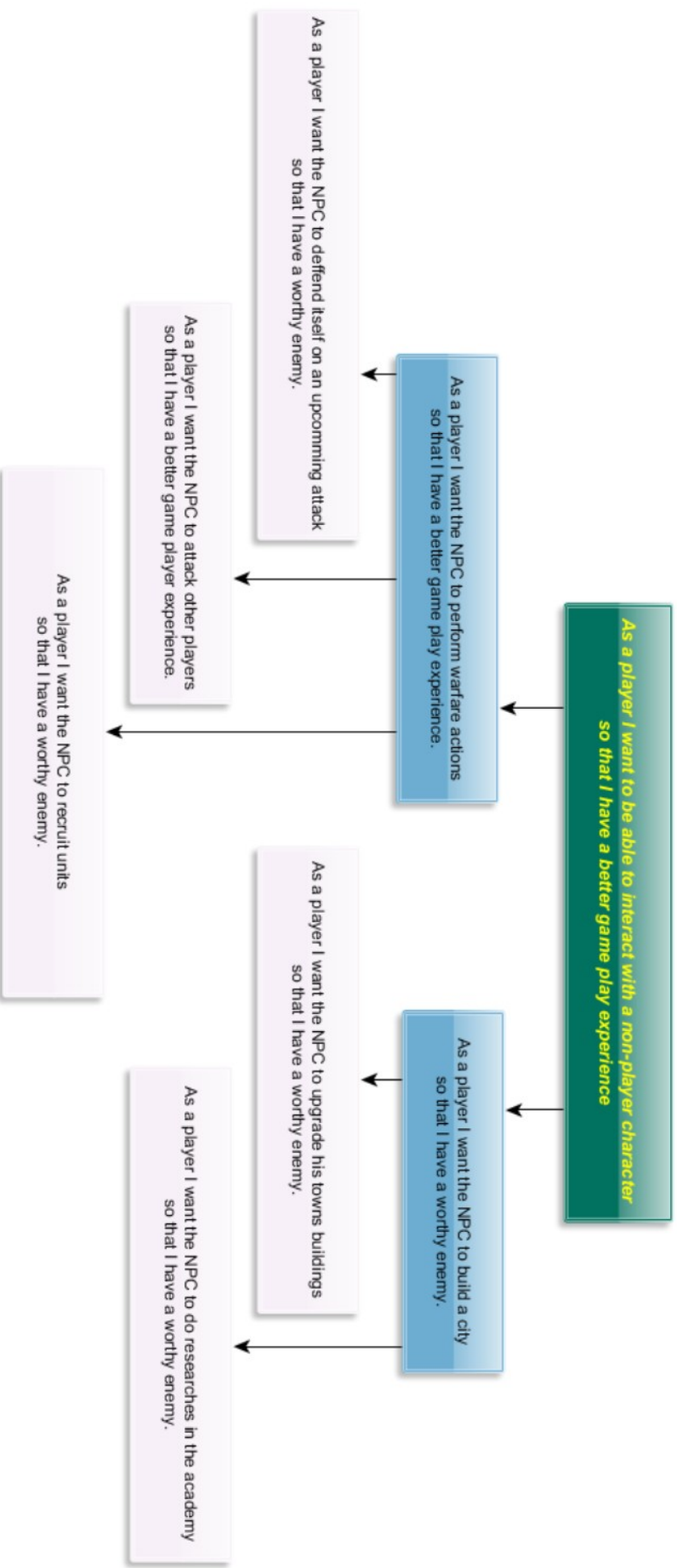
Since not all the requirements mentioned above can be implemented during the project due to limited time and resources, the functional requirements were divided into essential and additional abilities to have a better understanding what is important to be implemented first.

3.4.1 Essential Requirements

- F-P1
- F-P2
- F-P3
- F-P4
- F-P5
- F-CM1
- F-CM2
- F-SI

3.4.2 Additional Requirements

- F-P6:
- F-P7
- F-P8
- F-P9:
- F-P10
- F-P11



4 Change History

<u>Author</u>	<u>Date</u>	<u>Reason</u>
Lars Engel	30.03.2016	Creation of Document

5 References

- RAD adapted from
<http://www.cs.fsu.edu/~lacher/courses/COP3331/rad.html>

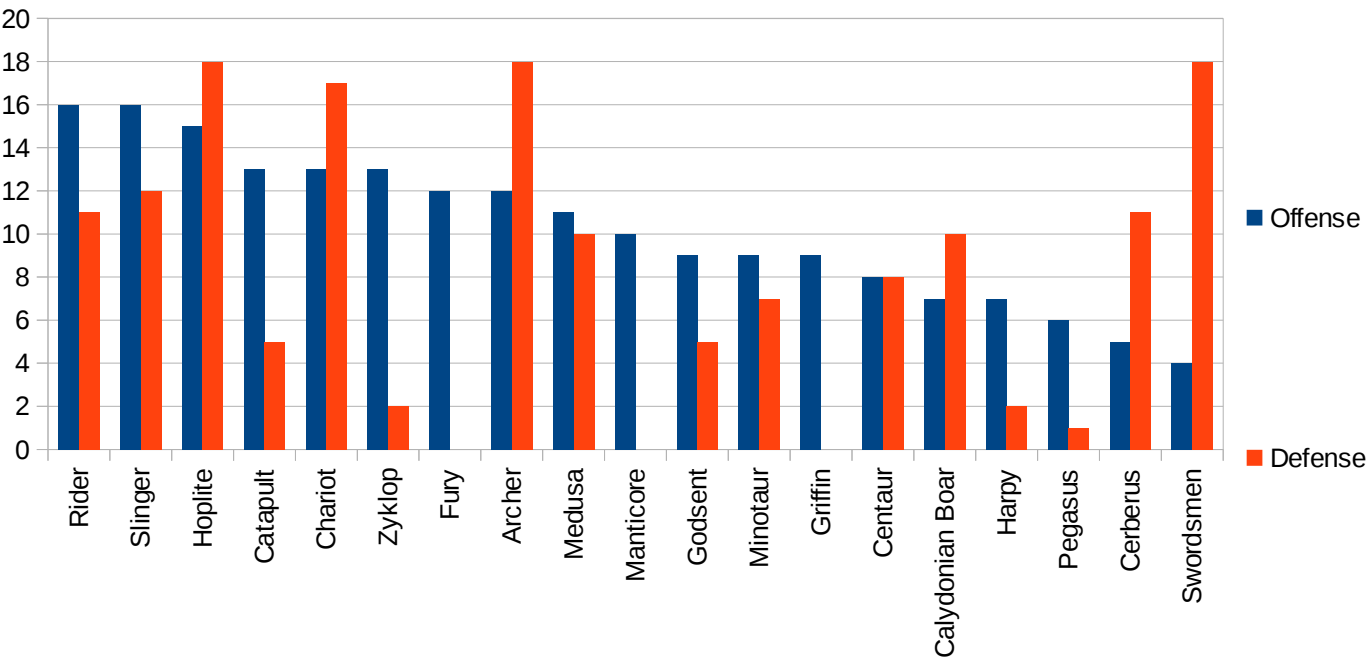
Appendix B

Learning the Fight System

Description Appendix B contains some detailed statistics of the outcome of the learning of the fight system. This statistics were used to find out which unit is strong in defense, which unit is strong in offense, and which unit is strong against which type of attack.

1 Statistics about Fights overall

	Wins Offense	Wins Defense
Rider	16	11
Slinger	16	12
Hoplite	15	18
Catapult	13	5
Chariot	13	17
Zyklop	13	2
Fury	12	0
Archer	12	18
Medusa	11	10
Manticore	10	0
Godsent	9	5
Minotaur	9	7
Griffin	9	0
Centaur	8	8
Calydonian Bo	7	10
Harpy	7	2
Pegasus	6	1
Cerberus	5	11
Swordsmen	4	18



2 Statistics about Types of Attack

	Type of Attack	Worst Fight against
Rider	Blunt	Hoplite
Slinger	Distance	Swordsmen
Hoplite	Sharp	Archer
Catapult	Distance	Swordsmen
Chariot	Sharp	Archer
Zyklop	Distance	Swordsmen
Fury	Distance	Swordsmen
Archer	Distance	Swordsmen
Medusa	Sharp	Archer
Manticore	Sharp	Archer
Godsent	Blunt	Hoplite
Minotaur	Blunt	Hoplite
Griffin	Blunt	Hoplite
Centaur	Distance	Swordsmen
Calydonian Boar	Sharp	Archer
Harpy	Blunt	Hoplite
Pegasus	Sharp	Archer
Cerberus	Blunt	Hoplite
Swordsmen	Blunt	Hoplite

3 Cost-Benefit Ratios

Defense per farm space: Swordsman: 17.3333333 Archer: 15 Hoplite: 12.3333333 Chariot: 12.3333333 Slingers: 5.66666667 Rider: 4.77777778 Catapult: 2	Attack per farm space: Slingers: 23 Rider: 20 Hoplite: 16 Chariot: 14 Archer: 8 Catapult: 6.66666667 Swordsman: 5
Defense per resource: Swordsman: 0.288888889 Archer: 0.230769231 Hoplite: 0.164444444 Chariot: 0.154166667 Slingers: 0.0871794872 Rider: 0.0597222222 Catapult: 0.0428571429	Attack per resource: Slingers: 0.117948718 Rider: 0.0833333333 Hoplite: 0.0711111111 Chariot: 0.0583333333 Catapult: 0.0476190476 Archer: 0.041025641 Swordsman: 0.0277777778
Attack / Defense ratio: Rider: 4.18604651 Slingers: 4.05882353 Catapult: 3.33333333 Hoplite: 1.2972973 Chariot: 1.13513514 Archer: 0.533333333 Swordsman: 0.288461538	

Appendix C

Test cases

Description Unit and Integration Tests were created to test the correct behavior of the functionality. Appendix C contains the list of created test cases.

1 Unit Tests

Class to Test	Testcase	Expected Result
DecisionResearchTechnologies	ResearchQueueFull	The NPC shall not try to research any technologies if the research queue is full.
DecisionResearchTechnologies	NoResources	The NPC shall not try to research any technologies if no resources are available.
DecisionResearchTechnologies	DoResearch	The NPC shall try to research technologies that are needed.
DecisionTrainUnits	TrainingQueueFull	The NPC shall not try to train units if the training queue is full.
DecisionTrainUnits	NoResources	The NPC shall not try to train units if no resources are available.
DecisionTrainUnits	DoRecruitment	The NPC shall try to train units if needed.
DecisionUpgradeBuildings	BuildingQueueFull	The NPC shall not try to upgrade buildings if the building queue is full.
DecisionUpgradeBuildings	NoResources	The NPC shall not try to upgrade buildings if no resources are available.
DecisionUpgradeBuildings	DoUpgrade	The NPC shall try to upgrade buildings that need to be upgraded.
GameTimedZoeCallTaskProvider	ProvideNoTasksIfZoeSleeps	The dataprovider shall not provide any tasks if the NPC is in state "sleeping"
GameTimedZoeCallTaskProvider	ProvideTasks	The dataprovider shall provide the correct amount of due tasks for each NPC
GameZoeReportTaskHandler	CalculateStatistics	The AI shall calculate the correct statistics out of the fight reports.

2 Integration Tests

Case to Test	Testcase	Expected Result
Attacking	SendAttack	An NPC shall attack towns of other players in case he has enough units to do so.
Attacking Goals	MakeNewGoals	The NPC shall create new goals to attack players if desired.
Character Decisions	MakeDecisions	The character decision maker shall gather the decisions from all sub modules and execute as many decisions as possible.
Daily Rythm	GoToSleep	An NPC shall go to sleep and wake up the next morning.
Defending	DefendTown	An NPC shall defend its own town in case he is being attacked.
Recruiting	RecruitUnits	An NPC shall recruit units based on goals and the predefined strategy.
Report Handling	HandleFightReports	The AI shall handle fight reports for NPCs and calculate statistics about used units.
Researching	DoResearch	An NPC shall research needed technologies in the Academy.
Strategy Decisions	MakeDecisions	The stratefy decision maker shall gather the decisions from all sub modules and execute as many decisions as possible.
Supporting	SendSupport	An NPC shall support its own towns with units in case a town is attacked and enough defensinve units are available.
Upgrading Buildings	UpgradeBuilding	An NPC shall upgrade its buildings based on the chosen strategy.