Rule Based Planner Framework

Game AI Technology Applied to Climate Control

Master of Science Thesis

Brian Johnsen
brian@deepthought.dk

Morten Lykkegaard Kristiansen
morten@deepthought.dk

Maersk Mc-Kinney Moller Institute for Production Technology
University of Southern Denmark, Odense

Supervisor : Bo Nørregaard Jørgensen

May 2007

Preface

This report is the outcome of the master thesis project of Brian Johnsen and Morten Lykkegaard Kristiansen. It has been developed from May 2006 to April 2007. This report also marks the end of our educations as Master of Science in respectively Applied Mathematics and Software Production, and Computer Science and Mathematics.

Where to Find the Goodies

All produced source code, tests, documentation, and this writing, are available on the enclosed CD-ROM. It is furthermore available on the web on the project homepage:

`http://www.deepthought.dk/sidious`

Conventions

The project concerns itself primarily with software development. As such, it contains terminology specific to that area, requiring the reader to have some insight in software development and technologies.
The typographic conventions are:

italic font	Indicates terms that are being defined, are borrowed from the Java™language, or are well-known software terms.
`courier font`	Computer stuff (Java class names and keywords, filenames, commands, and so on).

Note that all code excerpts throughout the thesis have had comments, logging statements, and the like omitted to enhance readability.

What's in a Name

To give the project an identity, a kind of brand, the project was named *Sidious*. Henceforth, the project will occasionally be referred to by its given name.

Acknowledgments

The authors would like to thank Bo Nørregaard Jørgensen for allowing us to pursue this subject, and for the patience and constructive guidance along the way. We would also like to thank Rising Tide for believing enough in the project, and giving us the chance to develop it for their game platform; it has been a great ride!

Morten Lykkegaard Kristiansen & Brian Johnsen
May 1, 2007

Abstract

How can the environment in a greenhouse be controlled automatically and autonomously? Does this control problem have anything in common with how an enemy agent in a computer game attempts to defeat the player? This project tries to solve the climate control problem by applying game inspired artificial intelligence.

Multiple AI technologies were evaluated, all benchmarked against the climate control problem. This evaluation culminated in a *rule based planner* as the candidate technology. This candidate was designed, implemented, and tested, and is the outcome of this thesis.

While neither of the authors are experts in climate control, it was necessary during development, to maintain a high degree of flexibility of the system. This flexibility led to the realization that the solution constituted a framework that, besides the climate control problem, prospectively could solve planning and control problems in general.

Since this project was equally a software development task, flexibility was also maintained during implementation through agile principles. This included fully automating every possible aspect of the project development.

The developed solution proves that the *rule based planner* was sensible, and combined with prudent expert knowledge is fully capable of field duty.

Contents

Chapter 1

Introduction

Two, seemingly unrelated, worlds, the world of computer games and the world of climate control are attempted merged; taking Artificial Intelligence or AI from games and applying it to climate control. This will involve the development of a framework to handle the greenhouse climate control problem.

It must be noted that most climate control components explained later in this thesis are on a *proof of concept* basis only. The authors of this thesis are not domain experts in the field of plant growth. This must however not detract from the quality of the results found in this thesis, as the problems are not the exact science of plant growth, but rather the problems and conflicts that arise when growing plants.

1.1 The Problem Domain

When growing plants on a large scale many considerations come into play. Primarily, the grower has to have some idea about what the current state of the environment is. He gets this information from sensors placed in strategic places around the greenhouse. Examples of such sensors are thermometers, hygrometers, and irradiance sensors. All the sensors contribute to the foundation for the grower to make his decisions. The grower then uses his expert knowledge to make a decision about whether or not to change the state of the environment.

1.1.1 Setpoints

The change in the environment is done through adjusting some setpoints. Setpoints are values of specific system settings like temperature (day and night), soil temperature etc. The setpoint adjustments usually have a direct result like rising the temperature in the greenhouse, but in most cases, they also have one or more indirect results; adjusting the temperature results in adjustment of the relative humidity. The problem with these indirect results is that they are not always predictable for a human; for instance, it is hard for a human, on the fly, to calculate the resulting photosynthesis rate when adjusting some setpoint. Besides these short term direct and indirect results the adjustments also have an impact on the production as a whole. This is one aspect where the grower must rely upon experience to make the right decisions. The production as a whole does not involve many adjustments of the setpoints because this would require the grower to make constant adjustments. This could also, without decision support from a seasoned expert or an expert system, lead to unforeseen and fatal consequences for the production line. This rigid type of climate control also results in a high energy consumption.

1.1.2 IntelliGrow

A research project where the goal was to reduce the energy and pesticide usage in plant production was begun October 1. 1999. This team developed an application called *IntelliGrow*[1] which would be part of the climate control system in greenhouses. One of the major differences from how it was before was the introduction of biological setpoints. The biological setpoints differed from ordinary setpoints in that they ensured that plant specific or indirect values where kept constant like the rate of photosynthesis and not temperature and other environmental directly measurable values. Their grand vision was being able to steer plant growth down to flowering time, height, etc.

The way these biological setpoints were reached was by dynamically adjusting the "old" setpoint values. For instance the photosynthesis rate is dependent on light, CO_2 level and more. All setpoint values where determined by the IntelliGrow application. These modules that control different aspects of plant growth are called components. Some of the components also dealt with costs and pesticide use. What the team found was that some times the wishes of these components conflicted. This would mean that they had

to run the setpoints and environment data though the system multiple times. This poses a problem because it is impossible to determine the amount of runs it would take to find equilibrium.

When looking at what the IntelliGrow application can do, it is justified to call it, albeit simple, but *Artificial Intelligence*. IntelliGrow is actually a simple single-pass expert system. Although the application was simple it achieved some great results in energy use reduction which lead to the idea that it was good course to pursue. It functioned almost like a proof of concept.

Knowing the basics of the IntelliGrow system, the *mission objectives* of this thesis can be stated.

1.1.3 Mission Objectives

Given a greenhouse with environment sensors that read temperature, humidity levels, etc. and setpoints to adjust temperature level CO_2 level etc. The application developed in this thesis must make use of the following things from IntelliGrow:

- Use dynamical setpoint adjustment to maintain biological setpoints.
- Use models developed to calculate the wanted environmental state.
- Making the system as flexible as possible to maintain several plant species.

Besides using the things discovered when developing IntelliGrow it was also imperative to get rid of as many shortcomings of IntelliGrow as possible. Some of these are:

- Calculating the new setpoint values by taking all the components into consideration.
- Make conflicting (read expense generating) setpoint adjustments only happen as a last resort.
- Make the system adhere to a master plan or strategy.

In other words, the mission objective of this thesis was to use the discoveries made of the IntelliGrow team, and develop a decision support system that eliminated the major flaws of that system.

Besides attempting to solve the greenhouse control problem, this thesis is also about developing software, and learning the "craft". Therefore, different strategies was tried out, and adapted into the development. In any case, the mantra of the development was *how to do "the right thing, the first time"*.

As the sub-title of the thesis indicates, the angle of approach for solving the greenhouse control problem was to investigate if artificial intelligence of computer games could be applied. The next couple of lines give the rationale for thinking it could.

1.2 Growing Plants vs. Computer Games

How could growing plants and computer games have anything in common? Well, to answer that question let us first look at a specific genre of games, the Role Playing game.

1.2.1 Role Playing Plant

In the role-playing game, the whole purpose is to develop your character. This character development is done by a series of choices and this ultimately defines whether the players character wins or loses. In some games, the progress you made through the game is even rated such that you can compare individual characters on how well they fared and fought in the game world. This kind is well represented by a game like *Diablo*. [2]

To apply this pattern of thinking to growing plants, consider a person growing a houseplant. The choices made by the person (or player) have a very big impact on the life of the plant. Even down to whether it wins, the plant flourishes, or loses, the plant withers. The actions of the person can even be rated by subjectively scoring the plant on looks etc.

1.2.2 Real-time Strategy for Plants

When growing plants on a larger scale in a greenhouse it can be considered as an army of characters just like in a real-time strategy game (RTS). A real time strategy game is a game where armies are pitted against each other and the player controls the actions of each character. A well known example of this kind of game is *Command & Conquer*.[3]

In the greengouse the grower (or player) governs the plants (or characters). The grower has to devise a strategy for how he wishes to grow these plants e.g. the lowest possible energy consumption, the best looking plants, fastest growth etc. It could even be a hybrid of these. In the RTS a player, as well as computer controlled opponent, must also devise a master plan. This plan could be that it must focus on building defenses, tank rushing, upgrading etc. The master plan or strategy must then be divided into several smaller bits. For the grower this is decisions like whether or not to adjust the temperature or change the amount of water each plant gets based on the input the grower gets from sensor readings. In the game the AI has to consider whether or not to research a given upgrade based on the knowledge it has about its opponents. Both the grower and the game AI are rated against how well they did in general and with regard to their strategy. Winning for the grower means that the majority of the plants survived and losing that the majority, or all, of the plants died.

Therefore, to answer the question posed earlier; it would seem like games and growing plants have a lot in common, but there are, not surprisingly, differences too.

1.2.3 Transparency

In game AI the only interesting part is the outcome of the AI, meaning that the calculations in between is of no interest to the user. This is, however, not adequate in a decision support system for a greenhouse. It is necessary to make the decisions of the software transparent to the grower, and equally important that the grower understands the way the system is influencing the operating results.

From the point of view of the grower, the following hesitations and questions have often been recorded:[4]

- Have claimed benefits been proven in real situations?
- Are decisions made by the system transparent and understandable for me, the grower?
- What are the risks of combinations of settings produced by these new controllers that are outside the experience domain of me and my colleagues?

- Which combinations will result in stress, and how do I know the critical boundary conditions of the crop?
- Does the system deal with quality aspects of the cultivation? If not, can I still intervene to correct undesirable conditions?
- Can the models really be trusted?
- Is the system flexible enough to accommodate change of crops and new investments?
- Is there a strong user support?

These statements emphasize the importance of the proposed solution being able to somehow reason about the choices it makes. This will be an important benchmark when evaluating which technologies to apply to the problem.

1.3 Roadmap

This thesis is structured in such a way that Chapter 2 first introduces the reader to an array of different AI technologies. Chapter 3 evaluates each of these technologies with respect to the greenhouse problem, and a specific candidate is chosen. Chapter 4 elaborates on the architecture of the developed solution, and gives the play-by-play of calls through the system. The framework development is the subject of Chapter 5, which is dedicated to the inner workings of the general components of the framework. Chapter 6 gives a quick tutorial to the framework, and elaborates on how to extend it. The greenhouse specific implementations are also described in this chapter. The development process used in the making of the framework is discussed in Chapter 7, which also holds a description of how the entire development cycle was automated. Chapter 8 explains the tests written for the system, and gives the rationale for writing them the way they were written. Chapter 9 is the discussion and conclusion chapter, which summarizes the achievements of the project; it is also the last chapter of this thesis.

It is time to get this show on the road; we will start by having a look at the various technologies that were evaluated.

Chapter 2

Technologies

> "I am sorry to say that there is too much point to the wisecrack that life is extinct on other planets because their scientists were more advanced than ours."
>
> — John F. Kennedy (1917-1963)

This chapter elaborates on the technology candidates for the greenhouse control problem. Each technology is described in summary form to give an idea about how they work. Furthermore, it is described how they have been applied in computer games.

2.1 Artificial Intelligence

Intelligence and the illusion of intelligence.

What is artificial intelligence? That is in the eye of the beholder. There is an important distinction between AI studied in academic research and that used in computer games. Traditionally academic AI is divided into two camps: *strong* AI and *weak* AI. The field of strong AI concentrates on trying to imitate the human thought process, and the field of weak AI focuses on solving real-world problems by applying AI technologies. The main focus of the two, however, is to solve the problems *optimally* with little or no consideration on hardware or time limitations. Academic AI research will often be more than happy to have a simulation running for hours or

even days on a Beowulf Horseshoe 800+ processor cluster, as long as the simulation spits out the *correct* answer.

In game AI on the other hand, time is usually of the essence and processor cycles are a sparse resource. Because of these constraints game AI has to cut some corners to provide sufficient realism and performance – the job of the game AI is to give an *illusion of intelligence*. The AI designers for the game Halo[5], for instance, discovered that their playtesters thought the AI agents they were playing against more intelligent, simply by increasing only the hitpoints necessary to kill them.[6]

What we were looking for was something in between. Having the system run for days to provide a solution needed within the next couple of minutes, is of course not an option. Nor can we get away with only growing plants that can take a beating.

The following sections introduce the various AI technologies that were evaluated during this project. Each section concerns itself with a specific part of AI technology. The sections give only rough introductions to their target AI. These introductions are mainly to equip the reader with a general idea of the domain, and to lay out alternatives to the final choices made.

When considering AI as alternative to old-fashioned *know-how*, an obvious place to start looking would be in the domain of *expert systems*.

2.1.1 Rule Based Expert Systems

This section elaborates on the subset of AI technologies known as *expert systems*.

Any kind of AI will try to imitate the human thought process in some way. The only problem is, that the thought process of humans is way too complex to be represented as an algorithm. However, most experts are capable of expressing their knowledge as rules for problem solving.

Representing Knowledge as Rules

Often the rules provided by the experts can be formulated as `IF-THEN` statements. The syntax of a rule is:

IF $< condition >$
THEN $< consequent >$

Rules can have multiple conditions, either in conjunction (`AND`):

IF $< condition_1 >$
AND $< condition_2 >$
$\vdots$
AND $< condition_n >$
THEN $< consequent >$

Or in disjunction (`OR`):

IF $< condition_1 >$
OR $< condition_2 >$
$\vdots$
OR $< condition_n >$
THEN $< consequent >$

And the consequent can have multiple clauses:

IF $< condition >$
THEN $< consequent_1 >$
$< consequent_2 >$
$\vdots$
$< consequent_m >$

The knowledge of the domain experts are extracted by the *knowledge engineer*[1] and put into the *knowledge base* of the system.

Structure

The structure of an expert system can be split into three main components:

The Knowledge Base contains the extracted expert knowledge stored as a set of rules.

The Database consists of the facts of the system, used to match against the conditional part of the rules in the knowledge base.

The Inference Engine handles the reasoning of the system. It links the rules from the knowledge base with the facts from the database, and by exercising those against each other, reaching the solutions.

[1]The knowledge engineer is the person responsible for transferring the knowledge and reasoning of the expert into the expert system.

Inference Techniques

The inference engine compares the facts from the database with the rules from the knowledge base. When an IF condition from the rule matches a fact, the rule is *fired* and its THEN consequent is executed. This execution may add a new fact to the database, and thereby cause another rule to fire. This is called inference chaining.

When an inference engine decides which rules are to be fired, it uses one of two principal ways; either *forward chaining* or *backward chaining*.

Forward Chaining is also called *data driven*, since it starts with only the known data, and proceeds forward, firing only rules which condition is fulfilled.

Backward Chaining is also called *goal-driven* inference. The inference engine is given a hypothetical solution and it then tries to find evidence to prove it. When presented with a goal it searches the knowledge base to find any rules that has that goal as their consequent. It then stacks that rule and searches for rules that match its condition part. It then stacks them and keeps this chaining until it reaches a condition that is met by facts in the database.

Conflict Resolution

An expert system needs to be able to solve conflicts between the rules that comprise its knowledge base. Among the solutions to this problem are:

- **Priority:** all rules in the knowledge base are given priorities, and conflicts are handled with respect to the priority of the involved rules.
- **Freshness of Data:** if conflicts arise fire the rule that uses *data most recently entered* in the database.

Dealing with Uncertainty

Often the information available to the expert system is incomplete, inconsistent, or uncertain. Some approaches to enable expert systems to cope with uncertainties have been developed; the main paradigms are *Bayesian Reasoning* and *Certainty Factors*.

Bayesian Reasoning works by applying a conditional probability, p, to the consequent (THEN part) of the rules. When the rules are fired their p-value is "accumulated", and the expert system produces an answer with a probability of being right. Bayesian reasoning relies on well proven probabilistic theory, and is therefore primarily applicable if sufficient statistical data is available.

Certainty Factors is a more heuristic approach to overcoming uncertainty. Like in Bayesian reasoning the consequent of the rules are assigned a value, *cf*, however, the certainty factor of the rule is as measure of the expert's belief, or disbelief, in the rule. The use of certainty factors is a practical alternative to Bayesian reasoning, as it coincides nicely with the thought process of a human expert.

Computer Game Relation

The computer game utilization of expert systems was primarily in the seventies and eighties for sport simulation games, e.g. baseball manager games.[8] Where the expert system was queried for the next "move" with respect to a given situation.

One of the most well known expert systems ever is probably IBM's *Deep Blue*, which defeated chess Grand Champion Garry Kasparov back in 1997. IBM Research Group them self describes it as: *"...a turbocharged expert system."*[9]

2.1.2 Fuzzy Reasoning

Fuzzy logic is based on the realization that the human language is ambiguous. The language deals in terms like: "slightly", "often", "quite" and "very". Classical logic deals in *crisp* values, where crisp are exact values like "true" and "false", or temperature=22.5 °C. Fuzzy logic, however, uses the same terminology as the spoken language. Instead of stating IF (temperature > 25 °C) THEN (open windows), fuzzy logic uses statements as IF (temperature is high) THEN (open windows). This is managed by applying *membership functions* to assign crisp values to *fuzzy sets*. The membership functions and the fuzzy sets are defined by experts who have domain knowledge. The fuzzy reasoning works in three major steps:

1. **Fuzzification**: using membership functions to assign crisp values to appropriate fuzzy sets.

2. **Rule evaluation**: application of fuzzy rules.

3. **Defuzzification**: obtaining the crisp results from aggregation of rule outputs.

The easiest way to explain how it works is through an example. Lets say we have three fuzzy sets; FS1="temperature is low", FS2="temperature is medium", and FS3="temperature is high"; and the current temperature, which of course is a crisp value, is 22 °C. The membership function then assigns the current temperature to the three sets, with a given *degree of truth*, usually between 0 and 1, giving them a partial membership of the given sets, see Table 2.1.

Fuzzy Set	FS1	FS2	FS3
Partial membership	0.1	0.6	0.3

Table 2.1: Partial membership with respect to the fuzzy sets

The logic then concludes that the current temperature is primarily medium but slightly high, which is consistent with what a human would conclude, and the rules can then act accordingly.

The application possibilities of fuzzy logic are very wide. In expert systems or hybrid intelligent systems, hybrids are explained in Section 3.2, where human experts provide knowledge into the system, fuzzy logic is very applicable because of its high cohesion with the spoken language.

Computer Game Relation

Consider writing the rules for a henchman's actions towards the player in a computer game. Whether the henchman should attack, defend, or retreat. Assume the only parameters he needs to consider are his health and firepower. Using traditional logic for this problem would result in a massive collection of rules, like:

IF $health > 90\%$
AND $firepower > 50\%$
THEN *attack*

IF $health < 10\%$
AND $firepower > 50\%$
THEN *defend*
$\vdots$ (accounting for every possible scenario)

By using fuzzy logic, the rules would look like:

IF *health_is_low*
AND *firepower_is_medium*
THEN *defend*

Actually mapping the rules into a simple associative matrix. See Figure 2.2. This greatly reduces the number of rules necessary and thereby reducing the complexity of the system.

	Low Health	Medium Health	High Health
Low Firepower	Retreat	Defend	Attack
Medium Firepower	Defend	Defend	Attack
High Firepower	Defend	Attack	Attack

Table 2.2: Fuzzy Associative Rule Matrix.

2.1.3 Artificial Neural Network

An *Artificial Neural Network* is the AI technology that resembles the way a biological brain works the most. Even though no precise definition of a neural network exists, the way they are built is agreed upon by most. Neural networks consist of a network of nodes. These nodes have many names, but the most commonly used names are "Neurons" or "Processing Elements". The simplest neural network consists of one of these neurons in a "layer" between input and output.

Artificial Neurons

The building blocks of neural networks are interesting in such way that we by looking at which problems they are suited for, get an idea about what makes neural networks tick.

The first "artificial neuron" was described by Warren McCulloch and Walter Pitts in 1943.[11] The neuron they described is also called the "Threshold

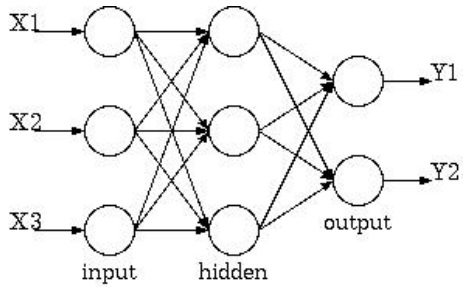

Figure 2.1: Structure of a perceptron.

Logic Unit". What it did was to take one or more inputs, sums it and produce an output of either zero or one. In other words it splits the input into two categories.

Later more advanced artificial neurons where created. They use more advanced mathematical functions to classify their input into more nuanced categories.

Perceptrons

Neural networks that consist of one or more layers between input and output are commonly referred to as layered perceptrons or just perceptrons.[12] Other types of networks do exist but the perceptrons are usually the technique chosen for most machine learning tasks. The structure of a perceptron can be seen in Figure 2.1.

Perceptrons are good at dividing input up into various groups and, when put into a network, can make even better divisions into groups. This capability of finding complex relations between input and output can be used for many things; one is to find patterns in data. These abilities can and have been applied in many areas.

The other variations of neural networks that exist can be applied to a multitude of problems. Many of them are very similar to perceptrons in their construction. The description of them will be omitted here, but good descriptions can be found in the literature in the bibliography of this thesis.

Machine Learning

What really makes neural networks interesting is their capability to be trained.

It has been proven that the training of a neural network is an NP-complete problem, i.e. when the problem grows linearly the training complexity increases exponentially.[13] This has however not deterred people from training neural networks.

When training a neural network the answer it provides must somehow be evaluated. This means that when deciding whether or not a solution delivered by the neural network is good enough, a cost function is usually applied. This cost function will almost always depend on the task.

In Machine learning three main paradigms exist:

Supervised Learning: What characterizes supervised learning is that the output for a given set of input is already known. This usually simplifies the cost function into being the distance between the output delivered by the network and the known result. This can e.g. be used in regression and pattern recognition.

Unsupervised Learning: When using this method there is no training set of data. The cost function will be the advanced part giving some fitness level of the relation between input and output. A neural network trained with unsupervised learning can e.g. be used for estimation and compression.

Reinforcement Learning: With this technique nothing is known of the relation between input and output beforehand rather the learner perceives the environment and then chooses an action from its set of possible actions. The environment then responds to this action and a "reward" is then given to or perceived by the learner. The goal for the learner is then to maximize the reward. This technique could also be called "learning by doing". Neural networks using this learning technique are usable in, for instance games and control problems.

General Problems

Neural networks are subject to noise, which can result in over- and underfitting. If we look at the input it is split up into two parts; signal and noise. The signal is the foundation for the solution and the noise is just that, noise. When the network is underfitting it is not capturing the entire signal and hence not producing a good solution. When overfitting the noise becomes

part of the base of the solution. This can be avoided by using very large training sets or using specific learning strategies.

Another problem is the "blackbox" nature of neural networks. They tend to be good at solving problems but give no explanation on how or why specific solutions where chosen.

Neurons meet computer games

Neural networks have been used in many computer games. They are used to calculate the best path and steer around a racetrack[14] or controlling the desires of non-player characters like in *Black & White*[16] where the desires and actions of the creature is controlled by several neural networks.[17] In general however, the game developers have stayed away from letting the neural network learn online, i.e. while playing, just in case it was to learn something unexpected and for this to result in broken gameplay.

2.1.4 Evolutionary Computation

> "Natural selection is a mechanism for generating an exceedingly high degree of improbability."
>
> — Sir Ronald Aylmer Fisher (1890-1962)

Evolutionary computation was developed with inspiration from the theory of evolution. The whole idea is that solutions are "bred" through a natural selection process, not unlike "survival of the fittest". The way this is calculated is through the use of Evolutionary Algorithms.

Evolutionary Algorithms

Most evolutionary algorithms are built with four phases. The first phase is the *Initialization* phase where the initial population is created. A member of the population, or individual, consists of a number of "chromosomes" which are parts of the input data.

The next phase is the *Evaluation* phase where all the individuals are evaluated and classified according to how well they solved the problem. This evaluation measurement is also called the fitness level of an individual.

Then the phase of *Selection* begins. In this phase an amount of individuals are selected, based on their level of fitness, for "breeding" new individuals. The fourth and last phase, before starting over from the Evaluation phase again, is the recombination or *Reproduction* phase. In this phase the chromosomes of the individuals chosen in the selection phase recombined. Again terms from the biological world is borrowed namely the selected individuals are called the parents and the recombined individuals the children.

The three phases *Evaluation*, *Selection*, and *Reproduction* are repeated any number of times until some criteria for termination is met. This criteria can be based on time, fitness level etc. After termination the child with the best fitness level is chosen as the best guess for a solution to the given problem.

A big problem, in the field of evolutionary computing is that a chance of ending up in an evolutionary "dead end" with only suboptimal solutions left is always a possibility. Some methods have been introduced to remedy this problem.

Mutation

One of these methods is known as *mutation*, where some randomness is introduced into the population via altering a few chromosomes at random in or after the reproduction phase. This keeps the population diverse and significantly reduces the risk of "dead ends".

Variants of Evolutionary Algorithms

There are many variants of evolutionary algorithms. Two of the variants are

Genetic Algorithms Here the individuals are represented by lists of numbers. One or more crossovers points (places where the individual is split into chromosomes) are present in this list.

Genetic Programming In this method the individuals are small programs which are then recombined to fix a bigger computational problem. This branch of evolutionary computation is very interesting, as this has actually produced two entirely new patentable solutions.[18]

Evolving Game AI

Evolutionary computing is used in many games as a mean to generate the good opponents. Many first person shooter games use genetic algorithms to train the computer-controlled opponents. This way of training is highly rewarding. This is because, when testing and developing good game opponents the game is tested at the same time. Taking a mediocre collection of opponents and then using evolutionary computing to develop them can result in unexpected behavior, but always results in an improvement of fitness after some generations.[19]

2.1.5 Planner

When performing any task the same pattern often arises. A goal defines when the task is done, and reaching this goal is possible in many different ways. It is usually possible to divide a task into several smaller subtasks which have some prerequisite, e.g. other completed sub tasks. Completing the task in an effective manner requires the formulation of a plan. If this formulation is done at random, the task may not be completed in the shortest possible time. It is also possible that the same work can be done another way or as a consequence of other subtasks completed first. This will then result in the task being completed with less energy used or less work done.

Problem Representation

When a computer has to simulate this behavior it does so by translating the problem into data it can understand. This *state* is represented internally as e.g. some sensory input like the known placing of game characters or what the current temperature is. These states can then be changed by some *actions*, like turning on a heater or picking up a weapon. This then alters the state of the system, but any action will usually involve a *cost*. This cost could be the energy used by a heater or the loss of ammo when firing a gun. It is also important that the goal of any plan or task can be translated into a system state e.g. the known position of the player character is on the floor in front of you.

The best way to represent these states and state changes is as graphs. A graph, used in a planner, is a set of vertices representing the various states and a set of edges representing a connection between those states. For more

complex problem representations, the edges can also be weighted. This adds the possibility to attach the cost of actions resulting in a change from one state to another.

Formulating a Plan

When the entire problem has been translated into something, the computer can understand only one task remains: formulating a plan. With the above representation, there are two possible ways of generating a plan: searching forward form the current state to the goal or backwards from goal to source. Searching backwards is advantageous when every state has some prerequisites which must be met.[20] Searching forward is advantageous when states do not have any specific prerequisites. Either way this is a *pathfinding* problem, which can be solved more or less efficient depending on the choice of pathfinding algorithm.

When using a good pathfinder it is possible to calculate optimal plans for a given problem and save energy by doing that. When using an efficient pathfinder it is even possible to save computation power which is paramount when this type of AI is used in games.

Planners and Free Agents

Applying planners to games yield many benefits. When maintaining the rules of the game e.g. adding the ability to reload a weapon but only when the weapon is drawn. This is an easy addition because the only thing that needs to be updated is the graph. If no planner was employed all the different plans involving that weapon would have to be updated.

Agents using the planner are free to formulate their own plans at runtime. This allows for greater diversity and complexity in the plans; all this is given by letting the decisions be made by these "free" agents.[20]

Chapter 3

Technology Reflection

> "The unexamined life is not worth living."
>
> — Socrates (470–399 BC)

In this chapter the AI technologies described in the previous chapter are evaluated with respect to their value in climate control. Also combinations of these technologies are evaluated before finally chosing a candidate for solving the greenhouse control problem.

3.1 AI vs AI

We have considered several AI technologies, as described in Chapter 2. The following reviews the pros and cons of the various techniques in relation to the greenhouse problem domain.

3.1.1 Expert Systems

The analysis of an expert system as candidate for the greenhouse control problem reveals several weaknesses. First and foremost expert systems need the entire problem domain quantified to work properly – if no rule exist for a given situation, the expert system has no basis for reasoning and therefore will not work. In a greenhouse environment unforeseen situations most certainly will occur, situations were the expert (grower) knows when to "bend the rules". An expert system is incapable of learning and will not know

when the rules need bending. Another problem is that the interrelationship between the individual rules is non-transparent; which can result in nondeterministic behavior.

3.1.2 Artificial Neural Network

When considering a neural network as a candidate, the technique seems like a prime candidate. They can be trained with existing data and results. The plans they then yield will also be good candidate plans for solving the problem at hand. Even if the gardener chooses to correct some of the decisions made by the neural network it would just learn and adapt.

However, some problems emerged. Retraining of the neural network could be needed if there is a drastic change in the rule set used to govern the plant growth. This change in the rules could be introduced by many things i.e new plant diseases, fungus etc. This then amounts in another set of training data that has to be generated. To generate this data a few generations of plants would have to be grown in diverse ways for the neural network to learn how to act. This could generate massive loss in the production for a while.

There is also the fact that there has to be a trained neural network for each plant type. The complexity of this training is also an important thing to consider here. If the training is not good enough the entire crop could be destroyed. It is said that neural networks should only be used where it is not of paramount importance that the neural network is right.[15] When controlling the greenhouse it is ok to make some minor errors in judgment. But a few, even one, severe problem and the entire crop could be lost. To avoid this, substantial amounts of data would have to be mined to produce both training and test sets. This data is at present not available.

Even if all of the above is achieved and rectified, one problem persists; it is next to impossible to give any explanation why specific steps in the plan were chosen. This poses a problem because it is necessary for the grower to know why a specific course of action is chosen and act accordingly, as the grower may know something the neural network does not.

The amount of problems this technique posed deemed it impractical to be used for the greenhouse control problem.

3.1.3 Evolutionary Computation

Using evolutionary computation as a means for solving the greenhouse control problem is a definite possibility. Each plan that has to be performed by the climate computer is a list of actions. These separate actions are easily implemented as chromosomes for use in an evolutionary algorithm. It is then possible for a good plan to be bred.

This approach may not be as efficient as other approaches but it can get the job done. However, the problem with an evolutionary approach is similar to the problem with neural networks. It is very hard to explain/reason why specific steps in the plan where chosen. It is even hard to explain why one child was chosen over another in the evolutionary algorithm, because there is an element of randomness to the choice.

This technique has also been deemed unusable due to the problem with not being able to explain decisions.

3.1.4 Planner

Applying the planner technique to the greenhouse control problem yields good results. This is because all problems can be divided into state changes based on actions. This makes it possible for the planner to seek out an optimal plan. The challenge in applying this technique is that predicting or calculating the consequences of adjusting setpoints can be hard. One of the challenges is that setpoint adjustments are not orthogonal i.e. two different setpoints can have an impact on the same variable. This is because setpoints usually have a direct effect and one or more side effects. An example of this is adjusting the heater has a direct impact on temperature but also has an indirect impact on the relative humidity.

Feedback regarding conflicts, limits, and costs have to be modeled. These have to be added to the planner as e.g. edge weights or somehow removed as possible states in the graph. This poses another challenge, because these edge weights or state removals have to be generated or controlled somehow, but if this is overcome the planner would be able to handle both conflicts and limits very well.

One great advantage when using the planner technique it is easy to retrieve the reasoning for each step in the plan which adds to the feeling of transparency.

3.2 Hybrid Intelligent Systems

The technologies reviewed so far have their advantages and disadvantages. However, by combining the different technologies it is possible to eliminate some of the obstacles associated with each specific technology, and thereby achieve a better overall result.

Combination of AI technologies is the very basis of *soft computing*, an initiative of Lotfi Zadeh, the "father" of fuzzy logic.[21] The idea of soft computing is to combine fuzzy logic, neural networks, evolutionary computing, and probabilistic reasoning. Such a system would be capable of reasoning and learning, and, at the same time, tolerant of uncertainty, imprecision and partial truth. Lotfi Zadeh is reputed to have said that *"...a good hybrid would be 'British police, German mechanics, French cuisine, Swiss banking and Italian love'. But, 'German police, French mechanics, British cuisine, Italian banking and Swiss love' would be a bad one."*[7]

It is, however, not necessary to apply the whole array of technologies to benefit from combining them.

3.2.1 Italian Love and German Mechanics

Neural Expert Systems As discussed earlier, expert systems are not capable of learning or adapting to a changing environment. This is on the other hand the strong side of neural networks. The drawback of neural networks is the fact that they represent a "black-box" for the users. They are unable to explain their reasoning. This is, however, the strong side of expert systems. Thus, when combining expert systems and neural networks, the spawn is a hybrid intelligent system, capable of learning and adapting, and being able to explain its reasoning. The neural expert system uses a trained instead of the knowledge base of an ordinary expert system. The neural net is capable of generalization and the hybrid is therefore equipped to deal with noisy or insufficient data. This ability is called *approximate reasoning*.

Neuro-fuzzy Systems The neural expert system hybrid is still dependent on traditional Boolean logic. This limitation is addressed by neuro-fuzzy systems. The neural network part deals with the raw data where it performs well. The fuzzy part deals with the higher-level reasoning. The hybrid combines the excellent parallel computation and learning

abilities of the neural net, together with the human-like knowledge representation and explanation abilities from the fuzzy systems. The neural net becomes the inference engine of the fuzzy system.

Problems of Using Neural Networks Neural networks, in any constellation, still require a significant dataset, to use for training and testing the networks. Unfortunately, we do not have large data sets for the greenhouse problem.

Evolutionary Neural Networks One of the problems with plain old neural networks is that the learning algorithms cannot guarantee an optimal solution. The teaching of the network might converge in a suboptimal solution, from which it cannot escape. This problem manifests itself both in the weights, as well as in the topology, of the network. For most real-world applications, this is unacceptable. Evolutionary computation, on the other hand, deals with optimizing based on "natural selection", as described in section 2.1.4. A way of circumventing the drawback of the neural network is to combine it with evolutionary computation. By encoding the weights or the topology of the network into chromosomes and performing a *genetic search*, an optimal solution would eventually emerge.

Same Old, Same Old The usual problem with neural networks still persist though; the total lack of explanatory capabilities.

Fuzzy Evolutionary Systems Genetic algorithms can be applied to generate fuzzy rules and adjusting membership functions. To apply the genetic algorithms an initial population of feasible solutions is needed. This approach, however, is primarily applicable to classification problems. For instance, it could be used as knowledge acquisition method for data mining tasks in complex databases.[7]

Especially fuzzy logic is used in combination with many other technologies. As mentioned in Section 2.1.2, the main advantage of fuzzy logic is how it correlates with the spoken language, and thereby the way humans think. This advantage is highly applicable combined with other technologies, as it simplifies development and reduces potential combinatorial explosion when trying to account for the entire sample space of a domain.

3.3 The Chosen One

When piecing together the different advantages of the technologies it became clear that the best solution would be a hybrid of some of the systems. The best course of action for making a choice was to look at the IntelliGrow application. It makes use of something called components. These components were actually small expert system modules telling the system which values to choose. As small expert systems, they functioned well, but when needed to work together problems arose. This problem was solved by making them have desire levels telling when their opinion was important and when it was not. These desires made the whole AI seem a lot like something to control non player characters.

With this solution, one challenge remained; the potential conflict in the chosen setpoint values. The solution was to make use of a planner which could simultaneously handle the potential conflicts and still maintain the ease of explaining each step in the plan. The conflict handling was done by using the desire levels of the rules as values for the edges. The rules that describe the conflicts would then add massive weights to the edges that contained conflicting settings. This would cause conflicting setpoint values only to be chosen as a last resort. Furthermore, each step could be explained because each rule would give an explanation why it returned some specific desire value.

The combination then ended up as a *Rule Based Planner*. This is a very flexible solution applicable to many more areas than greenhouse climate control.

The following chapters deal with how this technology applies to the mission objectives and the greenhouse control problem, and how it evolved into a framework.

Chapter 4

Framework Architecture

"Perfection does not exists - only the evolution towards it."

— Motto, Ferrari Formula One (1975-6)

This chapter elaborates on the conceptual design and architecture of the framework. The following sections outline the conceptual overview of the framework. The main components of the planner are described, as well as the bits and pieces that hold the framework together.

4.1 Conceptual Overview

The framework operates with three major concepts: *goals*, *graphs*, and *rules*. This section is dedicated to defining this threesome.

4.1.1 Goals

A goal represents a system state. In the greenhouse, any goal consists of a list of the sensors, which have some relevance to the deliverer of the goal. E.g., a goal devised by some temperature rule, see Section 4.1.3, may only contain information for the temperature sensor or thermometer, because other sensors has no relevance to the rule. The system then extrapolates the remaining data.

4.1.2 Graphs

The heart of the framework is comprised of graphs. A graph consists of a set of vertices and edges; which in this framework is represented in such a way that each vertex represents a state and each edge a transition between two states. Within an edge information is stored about what has been altered to result in the state change. One important thing to note here is that all state changes are not necessarily reversible and therefore the graph must in most cases be a directed graph. Furthermore, to keep the results from the path finding sound, the best course of action is to keep the graph acyclic, this is, however, not a necessity.

4.1.3 Rules

The building blocks of this framework are the rules. They represent constraints, restrictions, conflicts, and directions. The primary functions of the rules are to give the goals for the planner and to restrict the calculations.

The rules may be *goal-oriented*. E.g., the goal of a mean temperature rule may be a temperature state of 18 °C, whereas a maximum temperature rule may only "know" that the temperature must not exceed 28 °C. When the system calculates the goal to pursue for the next plan, all rules are queried for their preferred goal given the current state of the system. If a rule only represents e.g. some boundary constraint, it may not have any opinion about the result of the plan, and may then return nothing. This means that rules have direct *influence on the result* of the planner.

Furthermore, the rules are queried for an "opinion" of each state-change calculation. When queried, they must respond with a *desire-for-change*; a measure of how much they want to change the system state from one to the other. This *desire* is used to calculate the edge-weights in the graph. This modus operandi means that the rules have direct *influence on the direction* of the calculations.

4.2 Overview of Framework

The planner framework, which is implemented to be used as a service, consists of three major components; the *blackboard*, the *goal handler*, and the *planner*. Each of which has a specific set of responsibilities with respect to calculating the best plan of action for the system. The primary responsibility of the

blackboard is to mediate communication between the different parts of the system. The goal handler is responsible for choosing the next goal to base calculations on. The planner, the major player in the framework, handles the calculation and generation of plans. The following sections will elaborate further on these components. Figure 4.1 shows the general outline of these components.

4.3 Primary Framework Components

This section explains the specific modules necessary to understand the framework. These descriptions are kept at a provisional level, only to equip the reader with a general understanding of the framework structure. The gory details are postponed to the development and extension chapters.

4.3.1 Blackboard

The blackboard handles the interactions between the components of the system. This architecture decouples the interactions between the different participants of generating a plan. Besides handling internal communication, the blackboard is also responsible for external communication. To ensure all available information only be put and accessed in one place, the blackboard is an implementation of the Singleton pattern.[22]

The blackboard acts as a façade to the rest of the system; it holds the method for requesting a plan. The first step in generating a plan is to figure out which is the top priority goal to aim for; this is the domain of the goal handler.

4.3.2 Goal Handler

First and foremost the goal handler retrieves the goals from the rules and prioritizes them. All interaction with the rules is done through the *rule engine*, which is explained further in Section 5.8.4. The goal handler represents each request for the goals in a specialized internal queue, that queue is explained in Section 5.8.3. The primary reason for the goal handler was to abstract the prioritizing, or possible merging, of goals into a separate component. As of this writing the goals are merely prioritized by comparing their desire, but

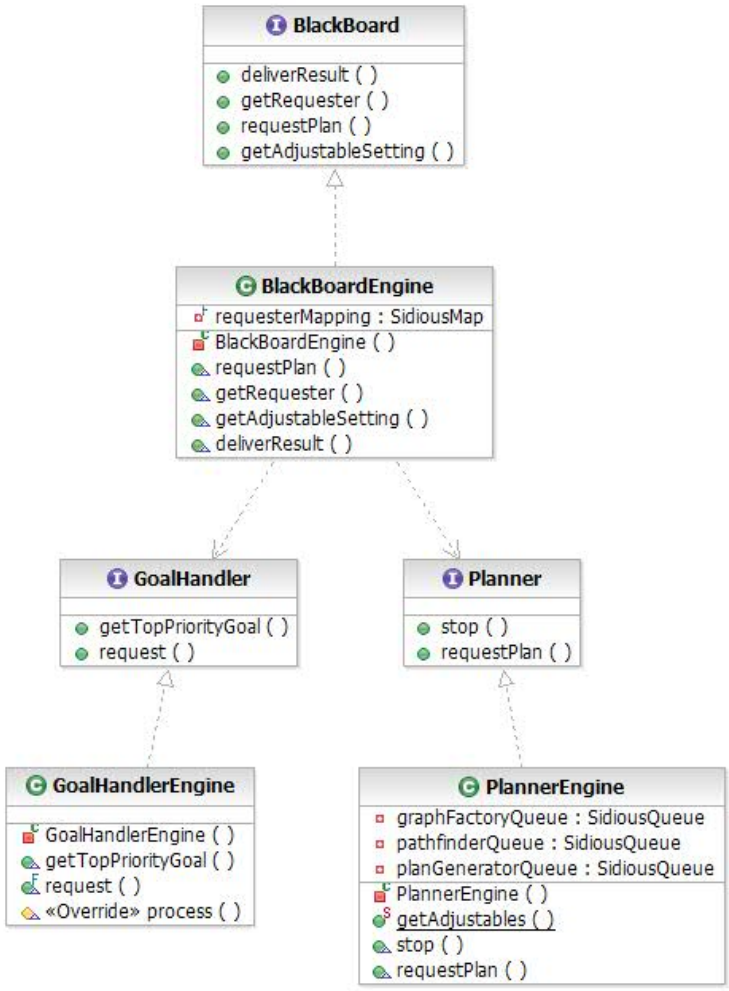

Figure 4.1: Overview class diagram

any other prioritizing of the goals can be done within the domain of the goal handler.

4.3.3 Planner

The planner is where the magic happens. The real-world problem is translated into a graph problem, the graph is searched, and a plan is generated. The planner is divided into three smaller parts; the *graph generator*, the *pathfinder*, and the *plan generator*. They are briefly described here. Section 5.6 in the following chapter is dedicated to unraveling how the magic is really done.

Graph Generator The graph generator constructs the graph for the pathfinder to search. The graph is constructed from a goal and the current system state.

Pathfinder The constructed graph is searched by the pathfinder. This search marks the shortest path[1] through the graph, by a pointer from each vertex to the vertex immediately preceding it.

Plan Generator The path from the pathfinder search is translated back into a real-world solution by the plan generator. The result of the plan generator is a *plan*, consisting of a sequence of steps to carry out by the requester.

4.4 Auxiliary Framework Components

This section explains the various utility classes used in the framework. The same level of description, as used in the previous section, is applied in this section. Again, all the more intricate details are postponed to the framework development and extension chapters.

4.4.1 Goal

The aim of any plan is to reach a goal. Because the goal concept in this system only holds information of where to go, it is represented as a single class. The `Goal` class describes two things:

[1] the shortest path is defined as the path that has the least accumulated desire.

- The end state it represents; which is the goal state.
- A *desire* measure of how much the rule that delivered the goal wants to have it fulfilled.

The goals are provided by the rules. This will be elaborated in the following Section 4.4.3.

4.4.2 Graph

The graph is interchangeable in the framework. This enables the use of both predefined and dynamically generated graphs, but they must adhere to the requirements of the search algorithm implemented in the pathfinder. The graph and its components are therefore defined through a set of interfaces with implementing classes. The outline of the `dk.deepthought.sidious.graph` package is depicted in Figure 4.2.

4.4.3 Rules

The concept of rules was divided into an abstract class, holding a variety of convenience methods, and a set of concrete rule implementations, holding the domain specific knowledge. Figure 4.3 shows the relationship, exemplified by the `dk.deepthought.sidious.rules` package.

The important aspects of rules are the *desire* functions and the *goal extraction*.

Desire Functions The planner is based on being able to query the rules for a measure of how "happy" the given rule is with a given state-change, this measure is called its *desire*. The edge weights is derived from that desire, and it is therefore of great importance for the correctness of the system. Section 6.4.2 is devoted entirely to this subject.

Goal Extraction A planner needs a goal to aim for. Again, the rules are called upon to provide that goal. This goal forms the basis for the generation of the graph. The provided goals must have a desire value attached. Section 6.1.1 explains this task further.

The abstract base class is described further in Section 5.4, and the concrete implementations are described in Section 6.3.2.

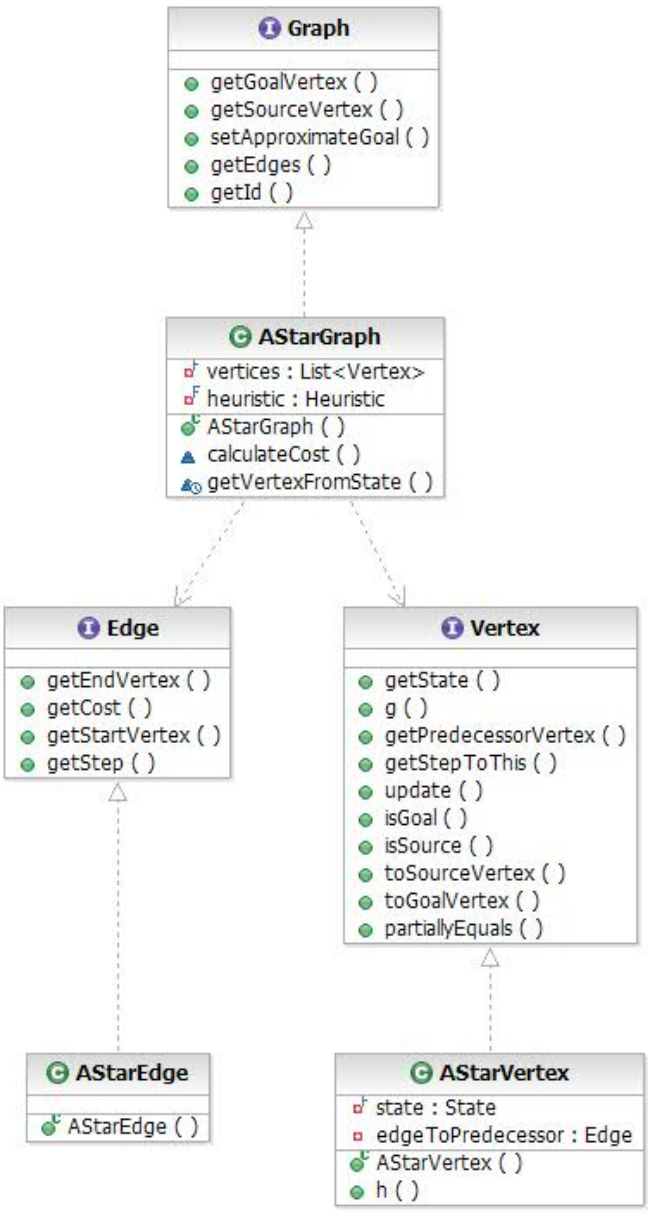

Figure 4.2: Class diagram of the `dk.deepthought.sidious.graph` package

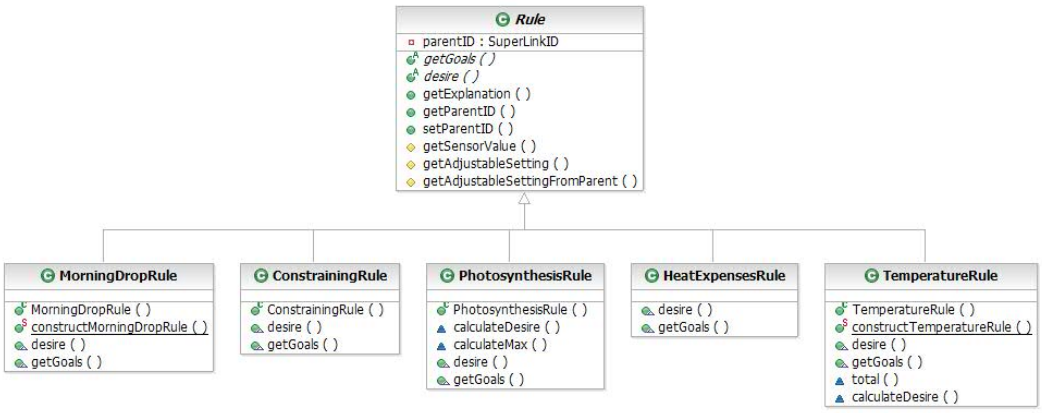

Figure 4.3: Class diagram of the `dk.deepthought.sidious.rules` package

4.4.4 SuperLink ID

The `SuperLinkID` class is an encapsulation of the identifier in the SuperLink application.[23] It acts as a unique identifier of all individual external components, e.g. climatic sensors, set points, the greenhouse etc. Instances of this class are immutable.

4.4.5 State

The system operates on state. When planning starts, the current environmental state of the client is used as starting point, and the goal of the planning is the state where the client wants to be; or at least where the rules of the client think it wants to be.

`State` is represented as an interface in the system. This promotes low coupling, and lets the clients have full flexibility of how to represent state in their system.

However, classes implementing `State` is presumed to be a container of *state descriptors*. The `ClimaticState` class, that implements the `State` interface, is exactly that. These state descriptors are implemented as `SensorInput`, which is the subject of the next section. The `ClimaticState` class is described in Section 6.3.3. The collaboration is shown in Figure 4.4.

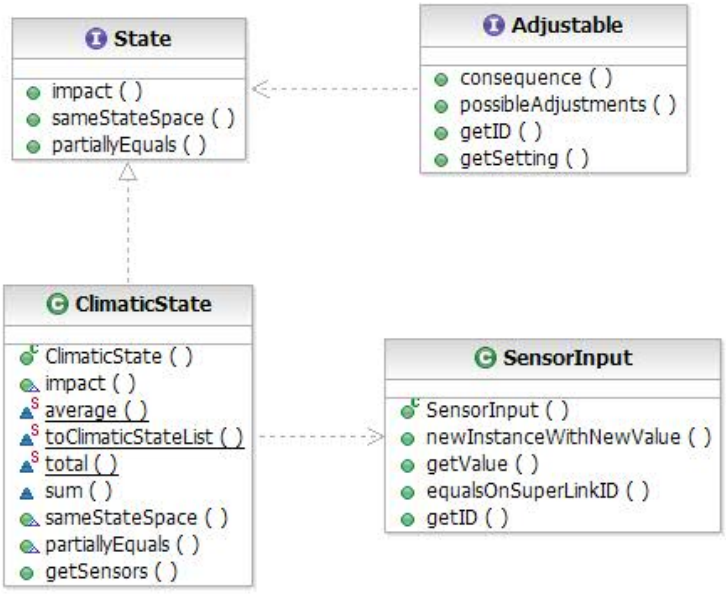

Figure 4.4: Class diagram of `State` and its collaboration classes

4.4.6 SensorInput

A `SensorInput` represents a microstate. It captures specific measurable environmental variables; disguised as sensors. The `SensorInput` class encapsulates the id of a specific sensor along with its current value. The class is immutable, and its overridden `equals` method ensures that sensors are equal on their ids.

4.4.7 Adjustables

The planner needs to be able to change the state that comprises the environment. The actuators for doing so are called *adjustables*, as they represent certain places for the system to adjust the state. To maintain flexibility this concept is implemented as an interface. The implementation is discussed further in Section 5.3.

4.5 Explanatory Capabilities

As it is stated in the introduction to the thesis, and all AI technologies have been evaluated upon, it is of vital importance that the system is capable of reasoning about its choices. It should of course be able to calculate a viable plan, but it should also be able to explain how and why it ended up the way it did.

The VIP players in the planning are the rules. They decide what goal to pursue and which direction to follow. As such, they are prime candidates for being part of the explanatory capabilities of the system.

The reasoning approach of the system is to capture the "state" of each rule for each calculated step in the plan. This is accomplished by letting the evaluation of the rules store the rule-desire association in specialized objects, and publish them through each step in the plan. The collaboration is shown in Figure 4.5. The concrete implementation is discussed further in Section 5.7.

4.6 Play-by-Play

This section outlines the flow of the framework. First is a short overview of the flow through the blackboard, goal handler, and planner. This is followed by a more thorough description of the flow in the planner component.

4.6.1 Flow Outline

The framework is built to be used as a service. Figure 4.6 illustrates the general flow through the framework. The service is activated by a request for a plan (1.1). When this request is made, the blackboard requests a goal from the goal handler (1.1.1). When the goal handler delivers the requested goal (1.1.1.1), the blackboard requests the planner to calculate a plan (1.1.2). The planner calculates the plan and delivers it back to the blackboard (1.1.2.1). The blackboard then places the calculated plan on the originating requester (1.1.2.2).

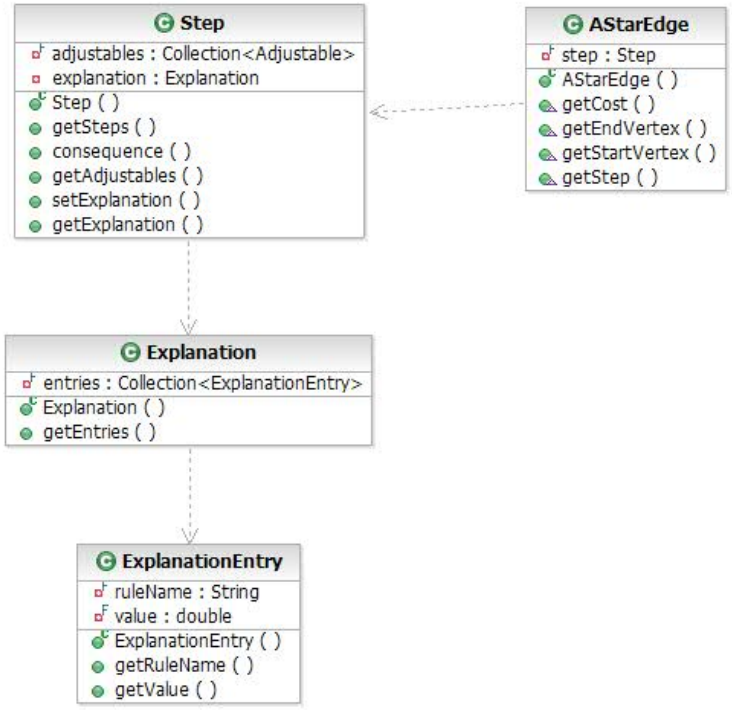

Figure 4.5: Class diagram of the explanatory capabilities

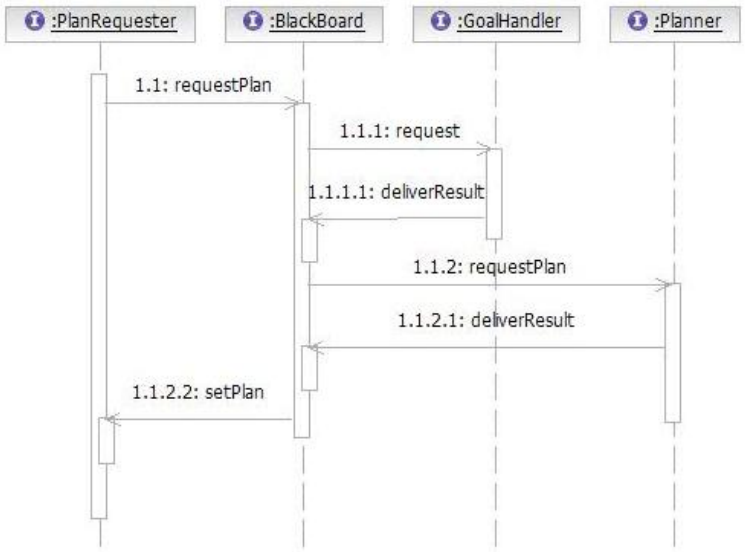

Figure 4.6: General framework flow

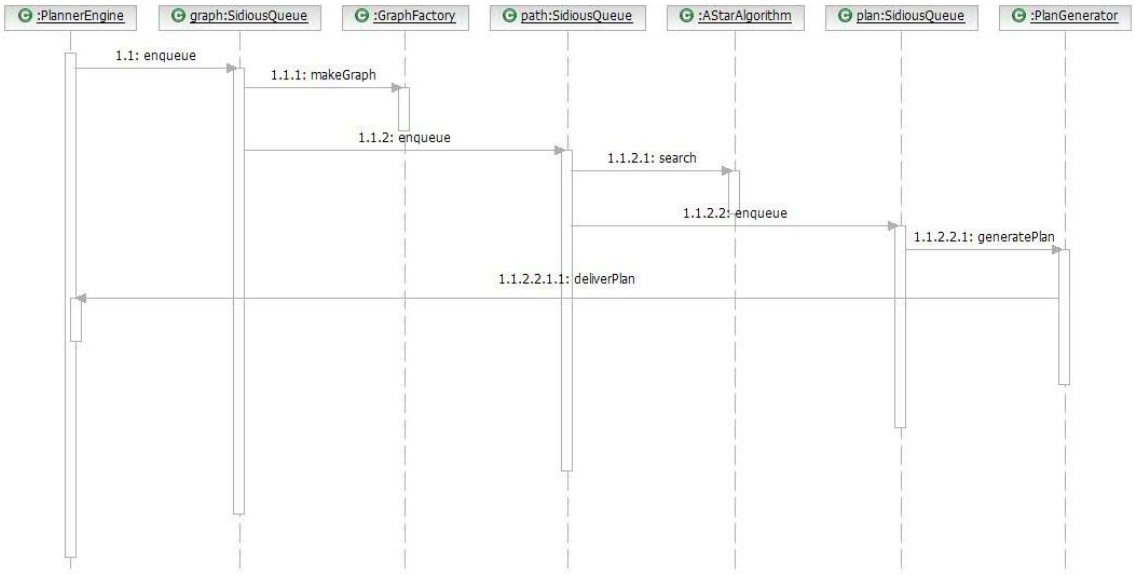

Figure 4.7: Sequence diagram of the planning.

4.6.2 Planner

The planner basically consists of three queues. Each of these provides separate thread-safe processing for the components comprising the planner. Figure 4.7 illustrates the call sequence. (See Appendix C.1 on page 128, for a larger view of figure 4.7.)

When the planner receives a request for a plan it enqueues the goal for processing by the graph generator (1.1). The graph generator dequeues the goal and constructs a graph (1.1.1). The graph is then enqueued for processing by the pathfinder (1.1.2). The pathfinder then searches the graph (1.1.2.1); during this search all information generated is stored within the graph. This graph is then delivered back to the planner, to be enqueued for processing by the plan generator (1.1.2.2). The plan generator processes the graph and constructs a plan (1.1.2.2.1). The constructed plan is then delivers back to the planner (1.1.2.2.1.1). The circle is now complete and the plan is delivered to the blackboard. The collaborating classes are shown in Figure 4.8. As promised, these classes will be fully disclosed in the following chapter.

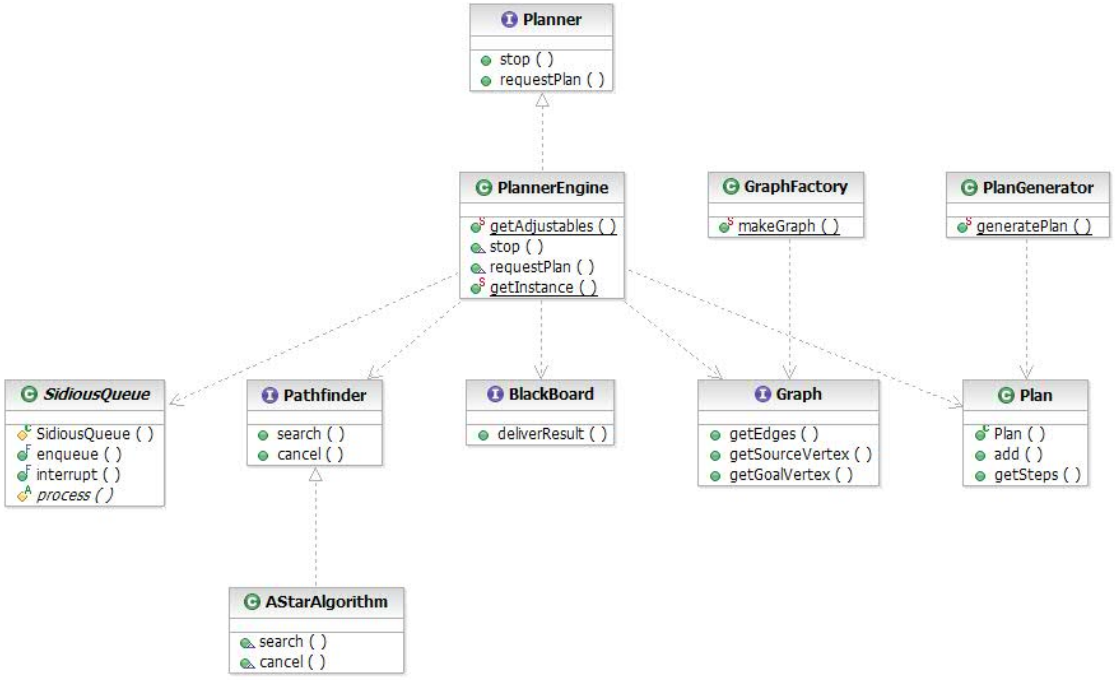

Figure 4.8: Collaborating classes of the planning.

4.7 GUI Package

The project also holds a GUI package, the `dk.deepthought.sidious.gui` package. This package functions only as a debug hook into the planner algorithm. It is implemented as a standard *model-view-controller*, where the controller registers itself for "events" from the model, and passes them on to the view.

The package provides nothing more than an insight of how the adjustables in the planner change value during calculation. It is implemented as a debugging tool only, and will not be elaborated further in this thesis.

Chapter 5

Framework Development

> One day Alice came to a fork in the road and saw a Cheshire cat in a tree. "Which road do I take?" she asked. "Where do you want to go?" was his response. "I don't know," Alice answered. "Then," said the cat, "it doesn't matter."
>
> — Lewis Carroll (1832-1898)

This chapter investigates the development and implementation of the framework. It touches upon every aspect of development. First, the different topics used in the development are covered, then the implementation of the individual parts and the history and milestones of the development is described. Lastly, the requirements for running and developing are stated, along with considerations regarding the quality of the code.

5.1 Principles and Algorithms

The following describes principles and algorithms which have been integrated or implemented in the framework.

5.1.1 Multi Threaded Environments

Once upon a time, concurrency was an "advanced" topic; but with ordinary desktop computers becoming multi cored, those days are over. Even if a program is designed to take advantage of only one thread, running it in a

```
@Immutable
public class Explanation {
    private final Collection<ExplanationEntry> entries;

    public Explanation(final Collection<ExplanationEntry> entries) {
        this.entries = new ArrayList<ExplanationEntry>(entries);
    }

    public Collection<ExplanationEntry> getEntries() {
        return new ArrayList<ExplanationEntry>(entries);
    }
}
```

Snippet 5.1: The Explanation class illustrates immutability.

multi core environment can render it **broken** if no consideration has been made regarding *thread-safety*.[1]

The framework is developed with thread-safety in mind. Several measures have been taken to ensure thread safety, and thereby the correctness of the system.

Immutability have been used where possible. An object is immutable if its state cannot be changed after is has been instantiated. Immutable objects are inherently thread-safe; they can be freely shared between threads because they do not possess any mutable state, and as such cannot be seen in an inconsistent state. An example is the `Explanation` class. Shown in Snippet 5.1. It contains a collection that is set when the object is instantiated, and thereafter never altered. Notice that the collection is *defensively copied* when it is passed to the constructor, and again when the `getEntries()` method is invoked. This ensures that the internal collection of the class cannot be altered from neither the creator of the class, which perhaps still holds a reference to the collection, and nor from a client as it only sees a copy of the collection.

To ensure proper sharing of mutable state *synchronization* is necessary. The Java keyword `synchronized` guaranties that only one thread is allowed within the block of code it encloses. Equally important, it guarantees that any state altered by a thread within a synchronized block will be visible to other threads after the block is exited. Synchronization comes at a cost, so to ensure the *liveness* of the program, the scope of any synchronized block

[1] An excellent "definition" of thread-safety from Brian Goetz: "...a thread-safe class is one that is no more broken in a concurrent environment than in a single-threaded environment."[24]

was minimized throughout the system. Snippet 5.2 shows an example from the `BlackBoardEngine`.

The `getRequester` method limits the scope of the synchronized block to a single statement, and then releases it again.

The `addPlan` method illustrates another excellent feature of the Java locking mechanism; that locks are *reentrant*. The method locks on the `requesterMapping` and calls the `getRequester` method which itself locks on the `requesterMapping`. This could seem like a deadlock, as the lock is already acquired. In Java, however, locks are reentrant. This means that the caller is identified as already holding the lock for the mapping, and a *lock-counter* is then incremented. When the caller releases the lock, in this case when `getRequester` returns, the counter is decremented. The lock is completely released when the counter reaches zero.

Notice that the `getRequester` call is not necessary within the synchronized block, but whenever an object acquires a lock, it has an overhead. This overhead, however, is minimized by calling the `getRequester` method within the synchronized block, as the lock is not reacquired but merely incremented.

Additionally, *thread confinement* is used to limit the scope of multi-thread accessible state. When state, or code, is confined to run in only *one* thread, it is per definition thread-safe. The system uses this technique by explicitly confining calculations, e.g. the graph searching algorithm, to be accessible only through a specially designed queue, see section 5.8.3, located in the `Planner`, see section 4.3.3.

To assist further development of the framework, classes being thread-safe or immutable have been documented as such. Either in their Javadoc or by using the annotations `Immutable` and `ThreadSafe`; borrowed from Java Concurrency in Practice[24].

5.1.2 Logging

Logging is used throughout the entire system. It provides an exceptional way of capturing the current state and context of the system when failures occur. This information can be invaluable, especially when hunting threading errors, which can be very hard to recreate because of their very nature. The log then may be the only way to catch the bug. Furthermore, the log can be used as a *trace* of the calls through the system; which can be an excellent way of checking the *correctness* of any part of the system. This feature, however, has a serious drawback for a system that has as abundant

```
    public PlanRequester getRequester(SuperLinkID id) {
        if (id == null) {
            throw new IllegalArgumentException("null not valid id");
        }
        PlanRequester returnPlanRequester = null;
        synchronized (requesterMapping) {
            returnPlanRequester = requesterMapping.get(id);
        }
        if (returnPlanRequester == null) {
            Repository.getPlanner().stop(id);
            return null;
        }
        return returnPlanRequester;
    }

    ...

    public void addPlan(Plan plan) {
        SuperLinkID id = plan.getId();
        synchronized (requesterMapping) {
            PlanRequester requester = getRequester(id);
            requester.setPlan(plan);
            requesterMapping.remove(id);
        }
    }
```

Snippet 5.2: The `getRequester` and `addPlan` methods from `BlackBoardEngine`.

a number of calculations as this.[2] The mere extent of produced data renders it useless, at least for human scrutiny, as well as slowing the calculations severely. The development of the framework extensively relied on testing as assurance of correctness, this is described in further detail in Chapter 8. Testing and logging are complementary; combined they ensure the correctness of an application as well as providing valuable information when failure inevitable occur.

The system makes use of the Commons Logging framework, which is a thin package bridging a number of different logging frameworks, and makes it possible to switch specific logging implementation without the need for recompiling. As specific logging implementation the framework use the Log4J package, which is perhaps the most widely used logging package for Java. See Appendix B for references.

5.1.3 Meta Data

The system is designed to be maintained and used by non software professionals, and as such most configurations can be done through meta data.

The system uses meta data extensively. Primarily this data exists in the form of Java `Properties` files. These files describe details that have been extracted from the source code, e.g. mathematical coefficients for desire calculations, and maximum and minimum temperature boundaries for rules regarding temperature. All rules of the system have a dedicated properties file to enter and alter rule specific data in.

The use of meta data enables on-the-spot tuning and configuration of the system, without the need for recompiling. Properties files were chosen for this task because they support both plain flat text files as well as XML files if needed, and they furthermore enables programmatically default values.

A natural extension of the system could be to extend it with a scripting language. Instead of using mere properties files, the calculations could be extracted out of the compiled code and done in a scripting language. The Java SE 6 release is extended with native scripting language capabilities.[25] This way, if e.g. a calculation seemed to be done the wrong way, the maintainers could alter the algorithms on-site. The framework would then become a mere shell to execute rules through.

[2] A test run of the entire system with trace log, once generated a 400MB+ log file.

5.1.4 A*

The A* algorithm is a graph search algorithm. The algorithm makes use of some rather brilliant things to achieve the effectiveness it has. It maintains a list of vertices which are yet to be studied. This *open list* is prioritized such that only vertices, which are on a candidate path, are extracted. This prioritization is done using two things. The first is an accumulated weight g of the cost of the shortest path to get to any vertex on the open list. The second is a heuristic h which estimates the cost from any vertex to the goal. When trying to find the shortest path it is important that the heuristic is admissible. An admissible heuristic for the A* algorithm always returns a value less than or equal to the cost of the actual shortest path from any vertex to the goal. If the heuristic is not admissible the algorithm may choose a suboptimal path.[26]

Besides the open list, A* employ a *closed list*. This list contains nodes already looked at by the algorithm. It is used to ensure that the same nodes are not checked over and over which can result in a drastic performance drop.

Now that the two list types used by A* have been described, the question of how they are filled remains. The first vertex to be put on the open list is the source vertex. First, the vertex is moved to the closed list, then all edges are retrieved and their endpoint added to the open list. Each vertex then has its g value updated if it is less than the one already stored. If this is the case the vertex also stores a reference to the start point vertex of the edge. This recipe is followed until the vertex removed from the open list is the goal vertex. When this happens the shortest path has been found.

Now a look at the different classes needed to get this algorithm up and running.

5.2 PlanRequester

The `PlanRequester` is an interface dictating which methods a client using the framework should implement. It has been developed to be as simple as possible because using the framework should be as simple as possible. Figure 5.1 shows the class diagram.

For a more thorough description on how to implement the `PlanRequester` parts see Section 6.1.1 in the Framework Extension chapter.

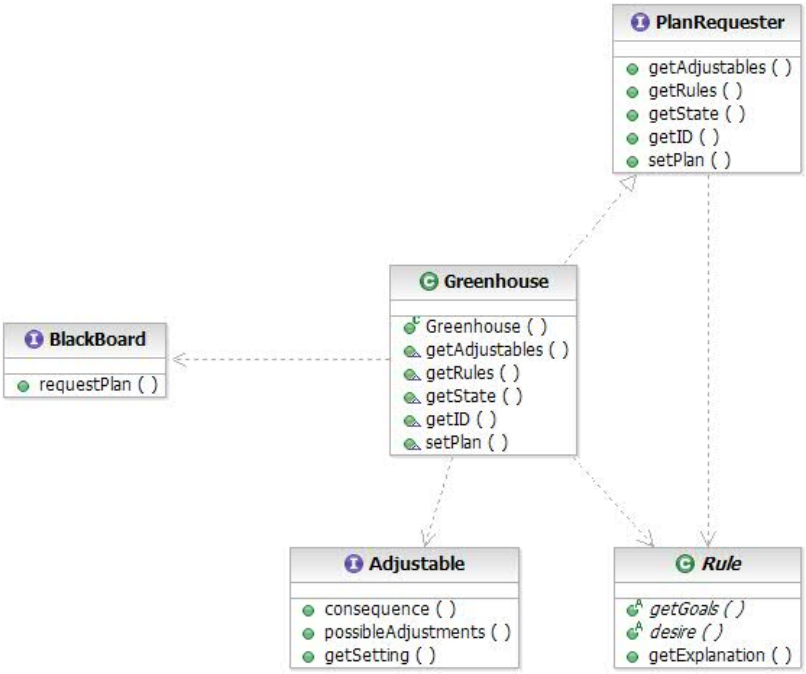

Figure 5.1: Class diagram of the `PlanRequester`

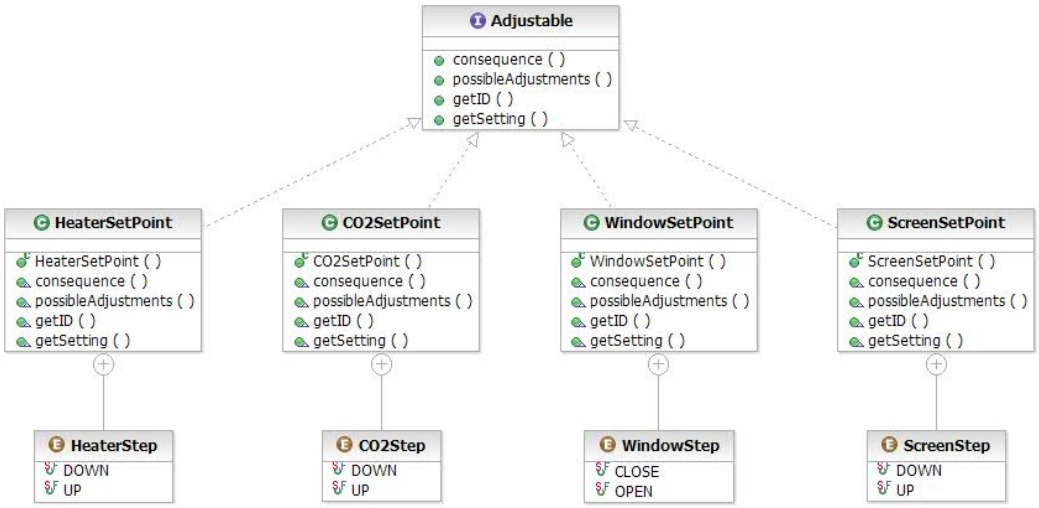

Figure 5.2: Class diagram of `Adjustable` and implementing setpoint classes.

5.3 Adjustable

An implementer of the `Adjustable` interface represents an element that can be adjusted and the result is a change of the environment. An adjustable must contain a setting and from this setting be able to calculate possible new settings. To avoid massive amounts of adjustables resulting from this calculation it is a good idea to divide adjustable settings into notches; notches could be implemented as *enumerations*. For instance, when adjusting the temperature setting, a notch could be represented by one degree up or down. This results in only two adjustables being returned when retrieving possible adjustments.

Classes implementing the interface must also contain a setting value, and override their `hashCode` and `equals` methods with respect to this setting. Besides this, the system assumes the implementing classes are immutable. All this is documented in the interface, through its Javadoc.

The classes implementing this interface are the `SetPoints`, which are discussed in Section 6.3.4. Figure 5.2 shows the class diagram.

```
public abstract class Rule {
    public abstract Collection<Goal> getGoals();
    public abstract double desire(State currentState, State newState, Step step);

    public String getExplanation() { ... }
    public SuperLinkID getParentID() { ... }
    public void setParentID(SuperLinkID parentID) { ... }
    protected double getSensorValue(State state, SuperLinkID sensorID) { ... }
    protected double getAdjustableSettingFromParent(SuperLinkID adjustableID) {
        ...
    }
    protected double getAdjustableSetting(Step step, SuperLinkID id) { ... }
}
```

Snippet 5.3: The API of the `Rule` class.

```
    protected double getAdjustableSetting(Step step, SuperLinkID id) {
        if (step == null) {
            return 0;
        }
        Collection<Adjustable> adjustables = step.getAdjustables();
        for (Adjustable adjustable : adjustables) {
            if (adjustable.getID().equals(id)) {
                return adjustable.getSetting();
            }
        }
        return 0;
    }
```

Snippet 5.4: The convenience method `getAdjustableSetting` from `Rule`.

5.4 Rule

The `Rule` is implemented as an abstract class. The reason that `Rule` is not an interface was because, during development of rules for the greenhouse, a lot of code duplication was discovered. Since this is unacceptable, the methods was implemented as convenience methods on the `Rule` class. The API of the `Rule` class is listed in Snippet 5.3. An example of this is the `getAdjustableSetting` which can be used to extract the setting of an adjustable from a given step. Snippet 5.4 shows how the method was implemented.

As mentioned in the previous chapter, descriptions of the concrete implementations are postponed to Section 6.3.2 in the Framework Extension chapter.

```
#Configuration file for MorningDropRule

temperature_sensor_id=temperature
time_sensor_id=time
dropTarget=14
dropDuration=120
minutesBeforeSunrise=30
```

Snippet 5.5: Property file from a `Rule`.

```
public SuperLinkID getID(String key) {
    return new SuperLinkID(properties.getProperty(key));
}
```

Snippet 5.6: The convenience method `getID` from `RuleProperty`.

RuleProperty

Rules can make use of an information container called `RuleProperty` where all information about id's and settings can be retrieved. All this information is stored in a `properties` file. An example of such a file can be seen in Snippet 5.5.

`RuleProperty` also contains a number of convenience methods that convert the retrieved data to the wanted type. E.g. when trying to retrieve the `time_sensor_id` property from the file in Snippet 5.5 using the `getID` method, the returned value is a `SuperLinkID` with the value *time*. The `getID` method is shown in Snippet 5.6.

5.5 GoalHandler

The job of the `GoalHandler` is to deliver the next goal to the planner. The `GoalHandler` ensures flexibility, but the basic implementation delivered with the framework should cover most needs. Before describing the goal handler in detail it is necessary to examine the goals.

5.5.1 Goal

The `Goal` class is basically described by three fields:

- `goalState`: A goal state field which defines the actual goal.

- `desire`: The desire level which refers to how much this goal wants to be reached, in other words, the desire of this `Goal` to become the goal of pathfinding.

- `origin`: This field connects the goal to a requester in the form of a `SuperLinkID`.

All these fields are of course `private` and only accessible through proper getter-methods.

To ensure that general use cannot alter the goal after it is instantiated, and to ensure thread safety, it is implemented as immutable. As all the fields of the goal are immutable themselves, immutability was easily archived.

Goals also implement the `Comparable` interface. This makes the desire levels of goals determine their importance in the goal handler by default. This can be overruled by a client, if it implements its own `Comparator` and passes it along to whatever container holds the goals.

By implementing goals this way, any goal can be represented, no matter what the task.

5.5.2 GoalHandlerEngine

The basic implementation of the `GoalHandler` interface included in the framework, is the `GoalHandlerEngine`. The `GoalHandlerEngine` also extends the `SidiousQueue`. (The `SidiousQueue` is described in Section 5.8.3.) It gets a request for a goal from the blackboard. This request, which passes a `SuperLinkID` along to identify the originating requester, is placed on the queue. This then wakes the queue thread and processing begins. It uses the *rule engine* to extract goals from the rules. The rule engine is described in Section 5.8.4. These goals are then added to a priorityqueue, which then sorts them, and, as shown in Snippet 5.7, the top priority goal is "peeked" from the queue. This goal is then delivered to the blackboard for further processing.

5.6 The Planner

This section describes how the different parts that comprise the planner and parts used by the planner have been implemented. The `Planner` class consists of three inner classes that all extend the `SidiousQueue`, which will

```
public Goal getTopPriorityGoal(SuperLinkID id) {
    RuleEngine ruleEngine = Repository.getRuleEngine();
    PriorityQueue<Goal> queue = new PriorityQueue<Goal>(ruleEngine
            .extractGoals(id));
    if (queue == null) {
        return null;
    }
    Goal returnGoal = queue.peek();
    ...
    return returnGoal;
}
```

Snippet 5.7: `getTopPriorityGoal` from the `GoalHandlerEngine`

```
public interface Graph {
    public Collection<Edge> getEdges(Vertex v);
    public Vertex getSourceVertex();
    public Vertex getGoalVertex();
    public void setApproximateGoal(Vertex v);
    public SuperLinkID getId();
}
```

Snippet 5.8: The `Graph` interface

be described in Section 5.8.3. These queue up elements for asynchronous processing by the three main parts of the planner: the `GraphFactory`, the `Pathfinder`, and the `PlanGenerator`.

5.6.1 Graph Interface

The most essential part needed to search for a shortest path is the graph. This is implemented as an interface, only containing a few necessary functions to leave the rest of the abstract data type open for framework extenders to modify. To make it even more flexible the implementation of the vertices and edges are also kept open to modify. The `Graph` interface is shown in Snippet 5.8.

5.6.2 GraphFactory

The responsibilities of the static factory `GraphFactory` is to construct specific graphs. In the factory, graphs are built with only a heuristic, a source vertex, and a goal vertex. The reason for the graphs not containing more vertices is because, the remaining vertices will be calculated on-the-fly, but more on

```
public synchronized void update(Edge edgeToPredecessor, double cost) {
    assert cost >= 0 : "The cost was less than nil: cost = " + cost;
    assert edgeToPredecessor != null : "The edgeToPredecessor was null";
    if (cost < costCandidate) {
        this.edgeToPredecessor = edgeToPredecessor;
        costCandidate = cost;
        logger.debug(this);
    }
}
```

Snippet 5.9: The `update` method from `AStarVertex`.

this later. When the graph is constructed it is handed back to the planner for further processing.

5.6.3 AStarGraph

The framework already contains an implementation of the `Graph` interface, the `AStarGraph`. This graph is implemented such that when it is first constructed, nothing but the source vertex and the goal vertex is present. Then whenever the graph is asked for the edges of a specific vertex, the edges and their end point vertices are created on-the-fly. First a description of vertices and edges used in the A* graph implementation.

`AStarVertex` The vertices are implemented such that they contain a cost of getting to them, a heuristic value to store their calculated heuristic, and a predecessor edge used to store the edge to the vertex preceding it on the shortest path from the source to the vertex itself. The `AstarVertex` contains a method to update the predecessor edge. this method is called `update` and the implementation of it can be seen in Snippet 5.9. As can be seen in that Snippet there are two `assert` statements which are used for ensuring against things like negative edges.

`AStarEdge` The edges of the graph contain a step in which information about the setting of the system setpoints is stored (See Section 5.6.9). This information is stored to maintain information about what caused the state change represented by this edge.

The A* graph implementation maintains a list of already visited vertices. This list is maintained to ensure that no system states are represented more than once. The implementation does not maintain any list of edges. This

```
synchronized Vertex getVertexFromState(State state) {
    if (state == null) {
        throw new IllegalArgumentException(
                "Cannot create a Vertex from null");
    }
    AStarVertex newVertex = new AStarVertex(state, heuristic.h(state));
    int index = vertices.indexOf(newVertex);
    if (index >= 0) {
        Vertex returnVertex = vertices.get(index);
        return returnVertex;
    } else {
        boolean success = vertices.add(newVertex);
        if (!success) {
            logger.error("Vertex " + newVertex
                    + " was not added to list of vertices");
        }
        return newVertex;
    }
}
```

Snippet 5.10: The `getVertexFromState` method from `AStarGraph`.

is because only edges potentially part of a shortest path are interesting. These potential edges are stored in the vertices themselves in the previously mentioned predecessor edge field.

The on-the-fly calculation of edges and vertices is done by calculating the consequence of each possible step from the current vertex. How these steps are retrieved and the mentioned consequence is calculated, is described in Section 5.6.9 where `Step` is explained. For each possible step s, the consequence of applying s to the state the vertex V represents. Then the method `getVertexFromState`, seen in Snippet 5.10, is used to check if the new state is already represented by another vertex in the graph, if this is the case it returns that vertex, if not a new vertex is created and added to the list of vertices in the graph. This found vertex is then used as the end point when creating an edge from V. The Snippet 5.11 shows the above described calculation.

5.6.4 Heuristic

The heuristic is, as the other parts, implemented as an interface to maintain flexibility. The reason for keeping the heuristic flexible is that the implementation is closely related to how the rules work. Furthermore, it is an integral part of the calculation time reduction. Some ideas for approaches to the heuristic can be seen in Section 6.3.5 in the Framework Extension chap-

```
public Collection<Edge> getEdges(Vertex v) {
    ...
    for (Step step : possibleNewSteps) {
        State state = step.consequence(oldState);
        Vertex newVertex = getVertexFromState(state);
        double cost = calculateCost(oldState, state, step);
        Edge newEdge = new AStarEdge(v, newVertex, step, cost);
        edgesForV.add(newEdge);
    }
    ...
}
```

Snippet 5.11: The on-the-fly calculation of edges and vertices from `AStarGraph`.

ter. The heuristic is one of the parts that can, and must, be tweaked when optimizing the framework for any task, not just the greenhouse extension.

The parts used in the path-finding algorithm have been explained, now its time to look at the `Pathfinder` itself.

5.6.5 Pathfinder

The pathfinder algorithm was implemented as a *strategy pattern*. This enables system with the flexibility to interchange the underlying algorithm. However, the best-suited algorithm for traversing a graph fast is the A* algorithm. (See Section 5.1.4 for a description of the algorithm) As such, the only strategy implemented for the pathfinder is the `AStarAlgorithm` class.

AStarAlgorithm

The A* algorithm is implemented using a `PriorityQueue` to represent the open list. In this queue, vertices are prioritized with regard to the sum of their cost and calculated heuristic. The `AStarVertex` class, that represent the vertices in the graph, implements the `Comparable` interface, which enables the priority queue to prioritize the vertices without further interference. The closed list, that is maintained to keep track of which nodes have already been visited, is implemented by an ordinary `List` implementation.

To start up the algorithm the source node is placed on the open list, and then a series of actions are performed:

1. Remove the vertex V with the lowest priority from the priority queue.

2. Check if V is the goal vertex. Exit if it is.
3. Retrieve all edges that emanate from V.
4. For each edge E
   (a) Add the weight of the edge to the cost of E.
   (b) Retrieve the end point vertex U, of the edge, with the calculated new cost.
   (c) Call the `update` method If the new cost is lower than the already stored cost, the predecessor edge is set to E.
   (d) If the end vertex is not on the closed list it is added to the open list.
5. Start over from step 1.

The implementation of the above enumeration posed no greater challenges and can be seen in its entirety in Snippet 5.12.

The check for equality is actually checking for partial equality. The reason for this is that the goal state is retrieved from a `Rule`. This rule does not necessarily possess knowledge about all the different state descriptors in the current environment. The check itself is actually performed by the state itself, which is the subject the following section.

5.6.6 State

To adhere to the philosophy of flexibility `State` is an interface open for implementation by users of the framework. Snippet 5.13 shows the interface. Some basic rules for how states must be implemented do apply. A `State` represents a snapshot of the environment. This can be in the form of a list of *state descriptors*. The state descriptors or `SensorInput` are the subject of Section 5.8.2. These state descriptors are an important part of a method called `impact`. The input parameter for the `impact` method is a list of states. The method must from the list of states and the state itself calculate a the combined or resulting state. This can be done through one of many statistical methods like calculating the average of each sensor input. This method is important for use in the planner when, the consequence of a `Step` is calculated. The `Step` class is described in Section 5.6.9.

```
    private void jAStar(Graph graph) {
        Collection<Vertex> closedList = new ArrayList<Vertex>();
        Queue<Vertex> openList = new PriorityQueue<Vertex>();
        Vertex currentVertex;
        Vertex goalVertex = graph.getGoalVertex();
        Collection<Edge> edges = null;
        openList.offer(graph.getSourceVertex());
        int vertexCount = 0;
        while (!openList.isEmpty() && !cancelled) {
            currentVertex = openList.poll();
            SidiousOutput.getInstance().addVertex(currentVertex);
            if (currentVertex.partiallyEquals(goalVertex)) {
                graph.setApproximateGoal(currentVertex);
                break;
            }
            edges = graph.getEdges(currentVertex);
            vertexCount += edges.size();
            for (Edge edge : edges) {
                Vertex endVertex = edge.getEndVertex();
                double sum = edge.getCost() + currentVertex.g();
                endVertex.update(edge, sum);
                if (endVertex.g() >= Double.MAX_VALUE) {
                    throw new IllegalStateException(
                            "Cost exceeded Double.MAX_VALUE value");
                }
                if (!openList.contains(endVertex)
                        && !closedList.contains(endVertex)) {
                    openList.offer(endVertex);
                }
            }
            closedList.add(currentVertex);
        }
    }
```

Snippet 5.12: The `jAStar` method.

```
public interface State {
    public State impact(Collection<State> states);
    public boolean sameStateSpace(State other);
    public boolean partiallyEquals(State state);
}
```

Snippet 5.13: The `State` interface

To make sure that states are not altered, as they are used in vertices which represent a given state, and to ensure thread-safety, it is important that `State` is implemented as immutable.

5.6.7 Plan

A `Plan` is the end result when the Planner has finished its calculations. A plan is associated with a specific requester through the stored `SuperLinkID`. Besides that, it contains a `Stack` where the `Step` elements, which comprise the plan, are stored. It is accessed as a Last In First Out (LIFO) stack because when the plan is generated in the `PlanGenerator` the steps are added in reverse order. When the requester needs a step from the plan it can just pop the top element from the stack. When there are no more elements on the stack, the plan has been performed.

5.6.8 PlanGenerator

To construct the plan the graph has to be traversed backwards, from goal to source. This is because each vertex on the path contains an edge to the vertex immediately preceding it. This is the job of the `PlanGenerator`. It traverses the graph backwards and extracts the steps one by one. They are added to a plan as they are extracted . This plan is then returned.

5.6.9 Step

A `Step` represents a "state changer". The step is a collection of all the system adjustables which each contain a setting. (See section 5.3 for more on adjustables)

The `Step` class contains two, for the planner, very important methods:

- `getSteps`: This method, which is depicted in its entirety in Snippet 5.14, generates a list of all possible new states with this step as the

```
public Collection<Step> getSteps() {
    Collection<Step> steps = new ArrayList<Step>();
    ArrayList<Adjustable> proxyAdjustables = new ArrayList<Adjustable>(
            adjustables);
    for (Adjustable adj : adjustables) {
        proxyAdjustables.remove(adj);
        Collection<Adjustable> resultingAdjustables = adj
                .possibleAdjustments();
        for (Adjustable newAdj : resultingAdjustables) {
            proxyAdjustables.add(newAdj);
            steps.add(new Step(proxyAdjustables));
            proxyAdjustables.remove(newAdj);
        }
        proxyAdjustables.add(adj);
    }
    steps.add(this);// "virgin" step
    return steps;
}
```

Snippet 5.14: The `getSteps` method of `Step`.

source. This is done by generating new adjustables, with altered settings. For each new possible setting of an adjustable, a step is generated. Lastly an unaltered step is added to the list. This is done because some implementations of adjustables can result in changes that can not be fulfilled in a single timestep.

- `consequence`: Snippet 5.15 shows a method which is used to calculate the consequence of this step on the current state. That is the resulting changes to the environment caused by the adjustable settings. This could be the heater adjusting the temperature in the environment. The consequence of each adjustable is calculated and stored in a list which is then combined into on single state by using the `impact` method, described in Section 5.6.6, on the state.

A step also contains an *explanation* which is a description of why this step was taken. These explanations are set by the `RuleEngine` during the calculation of the edge weigths in the graph. The following section explains the explanations.

5.7 Explanatory Capabilities

The system is equipped with a simple way of reasoning about the choices made during generation of a plan. As described in the previous sections,

```
public State consequence(State state) {
    if (logger.isDebugEnabled()) {
        logger.debug("consequence(State state=" + state + ") - start");
    }
    if (state == null) {
        return null;
    }
    Collection<State> states = new ArrayList<State>();
    for (Adjustable adj : adjustables) {
        states.add(adj.consequence(state));
    }
    State returnState = state.impact(states);
    if (logger.isDebugEnabled()) {
        logger.debug("consequence(State state=" + state
                + ") - end - return value=" + returnState);
    }
    return returnState;
}
```

Snippet 5.15: The `consequence` method of `Step`.

the system evaluates all the rules provided by the requesting client when the pathfinder algorithm searches for through the graph. When these evaluations are made, the system stores the rules and their associated desire values, as `ExplanationEntry` objects, in the designated `Explanation`. This object is stored on the `Step` object attached on each `Edge` in the graph. This design enables the system to retrieve the rules and their calculated desire for each step in the finished plan. Since it is the rules with the lowest *desire-for-change* that dictates the direction of the search, these values reveal the rules that "decided" each step; hence the reason for each step. Snippet 5.16 shows how the explanations are retrieved in the `RuleEngine`.

Explanation The `Explanation` object is a simple container in which to store and retrieve explanation entries. The class is shown in Snippet 5.1 on page 41.

ExplanationEntry The `ExplanationEntry` object wraps the name of a rule along with its desire value.

Both classes are immutable to facilitate free sharing among threads. They can be found in the `dk.deepthought.sidious.explanation` package.

```
double evaluateRules(Collection<Rule> rules, State current, State next,
        Step step) {
        ...
        List<ExplanationEntry> explanations;
        ExplanationEntry entry;
        explanations = new ArrayList<ExplanationEntry>();
        for (Rule rule : rules) {
            double desire = rule.desire(current, next, step);
            evaluation += desire;
            entry = new ExplanationEntry(rule.getExplanation(), desire);
            explanations.add(entry);
        }
        step.setExplanation(new Explanation(explanations));
        ...
}
```

Snippet 5.16: The retrieval of the `Explanation` from the `RuleEngine`.

5.8 General Components

This section contains implementation descriptions of the more general components used throughout the framework.

5.8.1 SuperlinkID

As described in Section 4.4.4, the `SuperLinkID` encapsulates a unique identifier in the system. The class is implemented as an immutable container of a `String` instance; it is a simple *decoration* of the string. The reason for encapsulating the string was to keep the implementation of identifiers of all components flexible and in one place. The `hashCode` and `equals` methods have been overridden to facilitate instances being equal on the encapsulated string, and not on the enclosing object reference.

5.8.2 SensorInput

The state descriptors or `SensorInput` are the ones that are changed based on consequence calculations. A sensor input consists of two fields, an id and a value. To encapsulate the `SuperLinkID` of the sensors, the `SensorInput` class provides a static factory method, from which it is possible to create a new instance with a new value but with the same id. This is important when calculating new states for use in the planner.

Besides the check whether two sensors are identical, which is used when

```
    private class InternalThread implements Runnable {
        public void run() {
            while (true) {
                T item = null;
                synchronized (queue) {
                    while (queue.isEmpty() && !interrupted) {
                        try {
                            queue.wait();
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                    if (interrupted) {
                        return;
                    }
                    item = queue.remove(0);
                }
                process(item);
            ...
    }
```

Snippet 5.17: The internal thread class of `SidiousQueue`.

checking state equality, it was also necessary to check whether two sensors have the same `SuperLinkID`. This functionality is used when calculating `Step` consequences and `State` impacts.

5.8.3 SidiousQueue

The `SidiousQueue` is a queue used to handle asynchronous requests throughout the system. A thread handles the entire processing. The thread waits until an item is enqueued before it then dequeues the item and starts processing it. This waiting is upheld by a lock object which is the queue itself. The internal thread is shown in Snippet 5.17.

Note that the `wait` statement is surrounded with a while loop checking whether the queue is empty or not. This is done to avoid processing to start without any items in the queue. This could happen if a `notifyAll` call is made, which results in all waiting threads to awaken. To avoid this the `enqueue` waking of the thread is done using the `notify` method instead of `notifyAll` on the lock object.

Another important thing to note is that enqueueing and dequeueing is not possible at the same time, but processing and enqueueing is however possible. This allows for a sequential enqueueing while the queue thread is "busy" processing an item.

```
public Collection<Goal> extractGoals(SuperLinkID requesterID) {
    PlanRequester requester = Repository.getBlackboard().getRequester(
            requesterID);
    if (requester == null) {
        return new ArrayList<Goal>();
    }
    Collection<Rule> rules = requester.getRules();
    Collection<Goal> returnCollection = extractGoalsFromRules(rules);
    ...
    return returnCollection;
}
```

Snippet 5.18: The retrieval of goals in the `RuleEngine`.

As seen in the above description the `SidiousQueue` is actually very similar to the *producer-consumer* pattern.

Because this class has multiple uses and the only thing that differes is the processing; the `process` method is abstract. The `SidiousQueue` is used in four different variants: the `GoalHandler` is a `SidiousQueue`, and the `Planner` class contains three inner classes which extend the `SidiousQueue`.

5.8.4 RuleEngine

The `RuleEngine` is the liaison between the rules and the rest of the system. It is an interface in case the implementation has to be replaced. However, the standard implementation that is included in the framework will do for most extensions.

The rule engine has two main tasks; which are extracting the goals from the rules and evaluating the combined desire of the rules. The first important task of the `RuleEngine` is extracting the goals from the rules. This is done when needed by the goal handler. It works by first retrieving the rules from the requester and then asking each rule for their goals. This interaction can be seen in Snippet 5.18. The rules themselves handle the creation and formulation of goals as will be described in the Framework Extension chapter.

The second task of evaluating the combined desire of the rules is accomplished by asking each rule for their desire given three things; two states and a step. The two steps are connected in such a way that the step applied to the first state yields the second state. The rules then return a desire based on this and the `RuleEngine` returns this value.

5.8.5 ServiceEngine

When using services, like the planner framework, as plugins in a project it was thought as a good idea to have a central place to ask for the assistance from other plugin services. The `ServiceEngine` was thought to be the mediator of such plugin services when the project was extended into the *Avian Game Platform*.[3] Inside the framework it is used as a means to get hold of the environment state and other tasks that involve outside help from elements such as a `PlanRequester`.

5.9 Development Milestones and History

This section concerns itself with the major implementation milestones, as well as the major refactorings, during development.

5.9.1 In the Beginning...

During the startup the program was very similar to IntelliGrow with regards to the different components. There were three kinds of components to deliver setpoints and costs.

5.9.2 Stepping Away From Intelligrow

To get rid of the conflicts that where possible in the IntelliGrow application it was decided that instead of setpoints some other representation was needed. This is where the climatic state came into being. Instead of directly calculating the setpoints their impact on the environment would have to be calculated. This was the first step, albeit small, away from IntelliGrow.

5.9.3 There and Back Again - A Gamecoders' Tale

The first major refactoring of the code was when the project was adopted into the Avian Game Platform of Rising Tide. The system was to be used to control the behavior of the non-player agents in the game.

[3]The system was bought by Rising Tide in mid November 2006. The company developed a children's computer game in Java.

Refactoring to game

The refactoring to the game required a complete rethinking of the design. This was where the brilliant idea of *rules* came into play. Instead of having separate components for cost, restrictions, etc. a general abstraction was needed. That general abstraction was the rule; it encapsulates the general idea behind the components nicely. The transformation was fairly easy, since each component was a rule in disguise anyway. This refactoring also implied the concurrency model to be revised; especially the passing of messages was reworked. Clearly, a bunch of other minor tweaks had to be performed to fit the system to its new assignment.

Re-Refactoring to greenhouse

To get the framework back on track for greenhouse calculations, the previously mentioned components were then refactored to rules. The simplicity of this refactoring made the flexibility of the framework, as it is now, very obvious. Some leftovers of the game still persist in the current system. However, the design seemed to justify their survival. Among the survivors are the rule engine and the service engine.

5.10 Implementation Details

The system utilizes some of the features added to the Java™5.0 platform - such as *annotations*, the `Queue` and `PriorityQueue` classes, and the *for-each* iteration idiom - and therefore will not compile with previous versions of the platform.

The system is developed and tested on MS Windows XP Professional Version 2002 Service Pack 2, and has been continuously tested on a Debian GNU/Linux 3.1 distribution; acting as build machine, see Section 7.3 for details.

5.10.1 Requirements for Running

Following is the requirements for running the system.

- Java™JRE 5.0 (or later)

5.10.2 Requirements for Development

Following is the requirements for further development of the system.

Requirements to compile:

- Java™JDK 5.0 (or later)
- JUnit 3.8 (or later) *
- Log4J 1.2.13 (or later) *
- Commons Logging 1.1 (or later) *
- JCIP Annotations[24] *

Auxiliary requirements for development:

- Ant 1.7 (or later)
- Jakarta ORO 2.0.8 *
- Commons Net 1.4.1 *
- Java2HTML 5.0 *
- EMMA 2.0 (or later) *
- JDepend 2.9.1 (or later) *
- Java NCSS 28.49 (or later) *

All requirements annotated with an * are bundled with the project in the `vendor/lib` or `resources` directories. Appendix B accounts for the use of each library and explains where to obtain it.

The required libraries are handled by the enclosed Ant script, and as such, the project requires no other system setup or altering of system *classpath*, besides installation of Ant.

5.11 Code Quality

The following explains some of the decisions made in this project regarding the quality of the code base.

5.11.1 Overloading

Overloading has generally been avoided throughout the project. Wherever possible, overloading of constructors has been replaced by *static factory methods* instead, and overloading of instance methods has been avoided completely. This decision was based on the notion that overloading seems counterintuitive. The selection among overloading methods is *static*, made at compile time, as apposed to *overriding*, for which the selection is *dynamic*, made at runtime, which renders overloading somewhat counterintuitive.

5.11.2 Overriding

All classes inherit methods from `java.lang.Object`. Most classes should also override at least some of these methods.

All classes representing value objects have had their `equals` and `hashCode` methods overridden if necessary. Several algorithms used in the project rely on objects being equal on more that just the object reference. The `PlannerEngine` e.g. uses a `HashMap` to map a `SuperLinkID` to a `PlanRequester`. If `SuperLinkID` did not override `equals` this mapping would fail miserably. This is because the `hashCode` method relies on that equal objects has the same hash code.

Furthermore, all classes have had their `toString` method overridden. This serves primarily to make the classes more pleasant to use, but also simplifies writing log messages significantly.

5.11.3 Code Conventions

The entire code base generally adheres to the *Code Conventions for the Java™ Programming Language*.[4] Especially all variable, method, and class names adhere strictly to the Naming Conventions. Throughout the system, every method and class name has been carefully thought through, and is chosen to make the most sense relative to its context.

5.11.4 Documentation

All packages, classes, interfaces, methods, and instance variables are documented in their respective *Javadoc*. Methods implementing interfaces or

[4] See: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

abstract methods, or methods overridden from `java.lang.Object` are only documented where the methods are described. This is to avoid the duplicate work of maintaining the same documentation in more that one place. Appendix A.2 describes where the generated Javadoc is located.

5.11.5 Crash Early

The system is developed to crash early in case of failures. This is managed by catching all checked exceptions, plus a couple of unchecked ones as well, and at the very least writing to the log before throwing a new exception or rethrowing the original. If the input parameters to a method are seriously erroneous, e.g. passing a `null` reference, the method awards this by writing the error to the log and throwing an `IllegalArgumentException`.

5.11.6 General

In general, the system is developed to be as resilient as possible. Some of the mechanisms used to promote this goal are:

- Wherever possible instead of returning `null`, methods will return an empty array.
- Wherever possible interfaces are used to define types, not the implementing classes.
- Functionality and data are not duplicated, but kept in the one place where it logically belongs.
- Encapsulation is enforced; all member variables are private and can only be accessed through methods.
- All non-used variables, instance as well as local, are completely removed.

Furthermore, several static code analysis tools have been applied to monitor the quality of the code base. The results of these tools can be found through Appendix A.4.

5.11.7 Look Ma, No Warnings!

A final note is that compiling the sidious project does not generate **any** warnings from the compiler - despite the fact that the compiler is running in the most pedantic mode possible. Through the Ant script, the compiler is run with `-Xlint:all` flag, which checks for software "lint" and then generates a warning if any such lint is present in the code. Examples of lint are the use of deprecated methods, or finally clauses that cannot complete normally.[5]

[5]See: http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javac.html for a complete description.

Chapter 6

Framework Use and Extension

> "Know where to find the information and how to use it - That's the secret of success."
>
> — Albert Einstein (1879-1955)

"Application Development with the Planner Framework" is the sub-title of this chapter; this is the how-to chapter. How to use the framework is addressed in a tutorial section. For developers that need to extend, augment, or maintain the framework, a section is concerned with greenhouse specific tasks, and another section is dedicated to the traps and pitfalls in extension development.

6.1 Tutorial

The framework is designed to be used as a service. In principle, only one method is ever needed to be called by a client to have a viable plan delivered; thats the `requestPlan` method. This section explains the prerequisites for attaining that goal.

6.1.1 Prerequisites

The driving forces of the planner are the rules and adjustables. They describe the domain of the clients; what can be manipulated by the planner and how to do it. Clients of the framework must therefore implement proper rules

and adjustables accordingly. Furthermore, two additional concepts are of importance; namely the notion of *state* and the *requester of a plan*.

Rules

Clients must implement rules describing the constraints of the domain. The concepts of rules are described in section 4.1.3. All rules implemented must extend the abstract `Rule` class, which besides containing several convenience methods, forces the extending class to have certain functionality expected by the framework. The API of the `Rule` class can be seen in Snippet 5.3 on page 49. The abstract methods should behave as follows:

- `getGoals` should return the immediate whishes of the rule. This means returning the objective of the concept the rule describes, appended with a desire for that objective.
- `desire` should return a measure for how "happy" the rule would be, if it were to change state from `currentState` to `newState`.

The rules already implemented for the greenhouse can serve as guidelines for developing new rules, these are described in Section 6.3.2. As an example, if the client was a chess game, the basic rules would be:

- Rules describing the movement constraints of the chess pieces (a rook can move in a straight line, horizontally or vertically)
- Rules describing the boundaries of the chessboard (64 squares)
- Rules describing the goal of the game (how to win)

Section 6.4.2 covers in greater detail how to deal with the complexity of writing rules and their desire functions.

Adjustables

For the planner to be able to devise a plan, it must have some way of adjusting the state of the environment. This task is handled by classes implementing the `Adjustable` interface. The adjustables of a client describes the possible ways that the client can manipulate its environment. Snippet 6.1 shows the `Adjustable` method set.

The following functionality is expected of implementations of `Adjustable`:

```
public interface Adjustable {
    public State consequence(State state);
    public Collection<Adjustable> possibleAdjustments();
    public SuperLinkID getID();
    public double getSetting();
}
```

Snippet 6.1: The `Adjustable` interface.

- `possibleAdjustments` should return all possible settings the adjustable can enter in the subsequent step. This is represented as a collection of new instances of the adjustable with the new setting. For example, with the chess game in mind, a lonely pawn would return only one new adjustable; one step forward. Actually, it would return two adjustables; also the adjustable for not doing anything at all.

- `consequence` should return the consequence of applying the implementing adjustable to the given input state. In a chess game, if a pawn is standing right diagonally in front of an opposing piece, the consequence of the "one-step-diagonally-forward" adjustable would be taking the opposing piece.

- `getSetting` should return the setting of the adjustable.

To reuse the example from above, the adjustables for a chess game would be the position of each chess piece. More accurately, they would be the means of which to manipulate the position; i.e. it is possible to move a chess piece (adjust its position), according to its movement rules.

PlanRequester

The client of the framework must implement the `PlanRequester` interface. This interface guarantees the framework certain methods are available on the requesting client. Snippet 6.2 shows the `PlanRequester` interface.

The methods of the interface are primarily self-explanatory. However, a quick outline would be:

- `setPlan` is the callback hook for the framework to deliver the finished plan.

- `getState` should return the current state of the environment relative to the requester.

```
public interface PlanRequester {
    public Collection<Adjustable> getAdjustables();
    public Collection<Rule> getRules();
    public State getState();
    public SuperLinkID getID();
    public void setPlan(Plan plan);
}
```

Snippet 6.2: The `PlanRequester` interface.

- `getRules`, `getAdjustables`, and `getID` returns the rules, adjustables and the id respectively, associated with the requester.

Henceforward, when referring to a client of the framework, it will implicitly mean an implementer of the `PlanRequester` interface.

State

The `State` interface is another valuable part of the framework. It encapsulates the environment of the client, which is quantified into a set of *state descriptors* called sensor input, implemented by the `SensorInput` class. A sensor input holds an id and a value. In the domain of the greenhouse, a sensor input is some sensor from the greenhouse, and in the chess example, it would indicate whether a square is occupied or not. Snippet 5.13 on page 58 shows the `State` interface.

The implementation class should behave as follows:

- `impact` takes a set of states and calculates the aggregated impact of applying the states to the current state. (See section 6.3.3 for a description of the implemented `ClimaticState`.)

- `sameStateSpace` takes another state and verifies if the two states belong to the same state space.

- `partiallyEquals` takes an input state and checks whether the state contains all state descriptors of the input state.

When all the described prerequisites are in place, it is time to fire the machinery up. The following section deals with how to use the framework.

6.1.2 Getting a Plan

If the previously described necessities behave as prescribed, the framework should respond to a request for a plan with a reasonable candidate plan. The delivered plan will consist of a sequence of *steps* to be carried out by the client. (See section 5.6.9 and 5.6.7 for a description of steps and plans respectively)

A client of the system should, as explained in the previous sections, implement the `PlanRequester` interface. When submitting a request for a plan to the system, the client provides a reference to itself along with the request. This reference is used by the system to query for the rules and adjustables, and the current state of the environment, of the client. It is furthermore used as a callback hook for delivery of the finished plan. The entire client workflow is comprised of only three steps:

1. call the `requestPlan` method
2. respond to the requests for data
3. receive the plan when it is delivered

It should be noted that the framework does not impose any directions of how to implement a plan once it has been delivered. The client is free to dispense with the plan as it sees fit.

6.2 How-To's

This section gives quick guidelines to checking out the project from its repository and importing it into an IDE. This how-to section is given to make it simple and straightforward to get the framework up-and-running, and verify the correctness and viability of the system.

6.2.1 Check Out from Repository

The project lives in a Subversion (SVN) repository on a server at the University of Southern Denmark. The following describes how to check the project out from that repository. The description assumes a viable Subversion client is installed.

Performing a checkout of the project is done by changing to the directory where the project should be placed and executing the checkout command.

```
>cd workdir
>svn co project_path
```

Where `project_path` is the path of the repository. This can be found in Appendix A.1. When the project is retrieved from the repository, it is time to get it into an IDE.

6.2.2 And Into an IDE

The following briefly describes the import of the project into some of the currently available IDEs.

Eclipse and Friends

The framework has been developed using the *Eclipse* IDE[1]. The project is bundled with an Eclipse `.project` file and `.classpath` file, and is therefore easily imported into eclipse, using the Import Wizard.

The project is similarly imported into the *IBM Rational Software Modeler* and the *IBM Rational Software Architect* IDEs.

NetBeans and Friends

The project has been successfully imported into the *NetBeans IDE*, and the *Sun Java Studio Enterprise IDE*. This can be achieved by using the IDE's New Project wizard, and choosing "Java Project with Existing Ant Script" or "Java Project with Existing Sources".

6.3 Greenhouse Extension

This section defines and explains the parts of the framework that are relevant when extending with respect to the greenhouse. The section also explains the components that have already been developed for the greenhouse.

6.3.1 How to Extend

The greenhouse can be extended with new technology and discoveries as they arise in the future. Most of it will be in the form of rules and setpoints. This

[1] The Eclipse IDE can be found at http://www.eclipse.org/

is why the descriptions of the already implemented parts for the greenhouse application in the following sections, can serve as additional guidelines for further development. A kind of *sandbox*.

6.3.2 Rules

The rules used in the greenhouse extension are all, except `PhotosynthesisRule`, not necessarily reflecting the real world. They are an attempt at demonstrating how it can be done. They are made without great domain knowledge and may therefore not represent reality that well. In other words they are guesstimates at best. This does, however not detract from the value of the example they all pose. They all demonstrate different approaches and ways of using rules in the framework.

Another class exists that extends the `Rule` base class; the `ConstrainingRule`. This class, however, solves a special problem, which will be addressed in Section 6.4.3.

HeatExpensesRule

The `HeatExpensesRule` is a rule which is not goal oriented. The rule will instead only influence the plan when it is being generated in the planner. The way it works is when the temperature setpoint has a value that is higher than the current temperature weight is added to the edge. The way this weight is calculated can be seen in Snippet 6.3.

Determining the value of the edge weight is a simplified version. In a real world situation, many other factors may come into play when determining what generates heat expenses. Nevertheless, it is a good example of a rule used to limit expenses.

MorningDropRule

The morning drop rule is a rule that uses the time of day to determine when it has a high desire for change. This rule dictates the need for a temperature decrease in the early hours of the day, then holding it for a duration and then heating the greenhouse back to normal operating temperature again.

When the plan is being generated, it adds weight to the edges when time is within the period where it operates. This results in, no matter which rule "won", an attempt at keeping a low temperature. This illustrates very

```
public double desire(State currentState, State newState, Step step) {
    ...
    Adjustable heater = null;
    SensorInput temperature = null;
    ...
    double returndouble = 0;
    double diff = 0;
    if (heater != null && temperature != null) {
        diff = heater.getSetting() - temperature.getValue();
    }
    if (diff < 0) {
        returndouble = 0;
    } else if (diff < 5) {
        returndouble = (diff / 5);
    } else {
        returndouble = 1;
    }
    return returndouble;
}
```

Snippet 6.3: The desire function of `HeatExpensesRule`

well how rules can influence the choices made in the plan only when it is of concern for the rule.

PhotosynthesisRule

The photosynthesis rule is making use of a model already built. The model used is a, to Java, translated version, of the component used in Intelligrow. This rule ensures that a photosynthesis rate of 80% or more is maintained. This is done by adding weight if the plan tries to stray from the 80% rate. The desire calculation, as can be seen in Snippet 6.4, is rather ineffecient.

This is because each time the rate has to be calculated it runs through two loops with calculations to determine the maximum rate. This quickly adds up to many calculations when a plan is generated. This problem is described in greater detail in Section 6.4.

The rule is a prime example of how easy it is to use already built models in the framework. It, however, also demonstrates one of the pitfalls that can arise.

TemperatureRule

The temperature is governed by the `TemperatureRule`. This rule ensures that the temperature stays inside the temperature boundaries. The bound-

```
double calculateMax(double shade, double sun,
        ArrayList<ArrayList<Double>> matrix) {
    LeafPhotosynthesisModel model = new LeafPhotosynthesisModel();
    double max = 0;
    for (int temperature = T_START; temperature < T_END; temperature++) {
        ArrayList<Double> inner = new ArrayList<Double>();
        for (int co2 = CO2_START; co2 < CO2_END; co2 += CO2_INCREMENT) {
            double val = model.calculate(temperature, co2, sun,
                    GLASS_FACTOR, SHADE_FACTOR, shade);
            inner.add(val);
            max = Math.max(val, max);
        }
        matrix.add(inner);
    }
    return max;
}
```

Snippet 6.4: The `PhotosynthesisRule` maximum rate calculation.

aries are determined as the highest and lowest, for the plants, non-lethal temperatures. Keeping the temperature inside the boundaries is done by adding massive weight to edges that lead a plan outside the temperature boundaries.

Another aspect of this rule is that it attempts to hold a mean temperature within the greenhouse. In addition, if the temperature is getting close to the boundaries, it will have a high desire valued goal, aiming for the mean temperature.

This rule is a demonstration on how several related things can be combined into one rule. The last rule that extend the `Rule` base class is the `ConstrainingRule`; the description of this class is, however, postponed to Section 6.4.3. How state is represented in the greenhouse extension is described next.

6.3.3 ClimaticState

The implementation of the `State` interface for the greenhouse extension is the `ClimaticState`. As required by the interface the implementation is immutable.

The `ClimaticState` is an example of how methods only used within the class is given "package" (or default) visibility, to facilitate testing the methods thoroughly. The class holds four such methods, which all are used in the calculations done by the `impact` method:

```
public State impact(Collection<State> states) {
    // Current implementation calculates the average.
    ClimaticState returnState = average(toClimaticStateList(states));
    returnState = returnState.incrementTime();
    ...
    return returnState;
}
```

Snippet 6.5: The `impact` method of `ClimaticState`.

- `sum`: takes a climatic state and calculates the sum of the matching sensors, which then are returned as a new climatic state.
- `total`: takes a list of climatic states and calculates the total sum of the matching sensors. This is done recursively.
- `average`: takes a collection of climatic states and calculates the average value of the matching sensors.
- `toClimaticStateList`: convenience method to convert the input states to climatic states.

The `impact` method is the one used to combine several consequences into one `ClimaticState`. The method is used when calculating each new state to be represented in the graph. This method is also responsible for incrementing time. Snippet 6.5 shows the `impact` method.

Another method used in `ClimaticState` is the `partiallyEquals` method, which looks at the input state and determines whether or not it contains all the `SensorInput` this `ClimaticState` contains. If that is the case the method returns true, as the two states are then partially equal.

New `ClimaticState` instances are generated by setpoints when they calculate consequences. Setpointe are the subject of the following section.

6.3.4 Setpoints

In the greenhouse domain, the adjustables are the *setpoints*. The concept of setpoints is described in Section 1.1.1. To facilitate this, several implementations of the `Adjustable` interface have been developed.

A general feature of all implemented setpoints is that they are all immutable, and contains an internal *enumeration* that holds the possible adjustments for that particular setpoint. The enumeration abstracts the di-

rections and the increment-size of each setpoint; e.g., because of the servo engine controlling it, a window can perhaps be opened 5% and closed 5% at a time, so the enumeration of that window setpoint would contain two instances, namely `OPEN(5)` and `CLOSE(5)`. This is in fact exactly what the implemented `WindowSetPoint` does. The following describes the individual implementations in further detail.

CO2SetPoint

This is the abstraction of the CO_2 setpoint in the greenhouse. The `consequence` method of this setpoint works like this:

1. retrieves the CO_2 sensor value from the input state
2. looks at the difference between its own setting and the retrieved sensor value
3. adjusts the sensor value according to the maximum increase or decrease capabilities per time-step
4. appends the new value to a constructed list, containing the other untouched sensors
5. constructs a new `ClimaticState` and returns it

HeaterSetPoint

This class represents a setpoint for the heater or thermostat in the greenhouse. Remembering that increase in temperature result in a decrease in relative humidity, the `consequence` method of this setpoint works like this:

1. retrieves the values of the temperature and humidity sensors from the input state
2. calculates the difference between its setting and the retrieved temperature sensor value
3. calculates the possible temperature change according to the approximated rate of change
4. appends the new value to a constructed list, containing the other untouched sensors

```
public State consequence(State state) {
    ...
    Collection<SensorInput> sensors = climaticState.getSensors();
    Collection<SensorInput> newSensors = new ArrayList<SensorInput>();
    // When temperature increases, humidity decreases
    for (SensorInput input : sensors) {
        if (input.getID().equals(temperatureID)) {
            double temperature = input.getValue();
            double delta = setting - temperature;
            double newValue = temperature + timestep * FACTOR_PER_MINUTE
                    * delta;
            newSensors.add(input.newInstanceWithNewValue(newValue));
        } else if (input.getID().equals(humidityID)) {
            double decreaseFactor = decreaseFactor(input.getValue());
            newSensors.add(input.newInstanceWithNewValue(decreaseFactor));
        } else {
            newSensors.add(input);
        }
    }
    ...
}
```

Snippet 6.6: The consequence method of the `HeaterSetPoint`

5. adjusts the retrieved humidity sensor value from the input state for the temperature change

6. constructs a new `ClimaticState` and returns it

The primary part of the `consequence` method is shown in Snippet 6.6.

ScreenSetPoint

This setpoint is for controlling the sunsscreens in the greenhouse. The `consequence` method of this setpoint works like this:

1. retrieves the irradiance sensor value from the input state

2. reduces the irradiance with the percentage the screens are closed

3. appends the new value to a constructed list, containing the other untouched sensors

4. constructs a new `ClimaticState` and returns it

```
public State consequence(State state) {
    ...
    double deltaTemperature = outsideTemperature.getValue()
            - temperature.getValue();
    double deltaHumidity = outsideHumidity.getValue() - humidity.getValue();
    double newFactor = windowAreaPercentage * (setting / 100)
            * SystemSettings.getTimestep();
    if (Math.abs(newFactor) > 1) {
        newFactor = Math.signum(newFactor);
    }
    double newTemperatureValue = temperature.getValue() + deltaTemperature
            * newFactor;
    newSensorList.add(temperature
            .newInstanceWithNewValue(newTemperatureValue));
    double newHumidityValue = humidity.getValue() + deltaHumidity
            * newFactor;
    newSensorList.add(humidity.newInstanceWithNewValue(newHumidityValue));

    State returnState = new ClimaticState(newSensorList);
    return returnState;
    ...
}
```

Snippet 6.7: Excerpt from `consequence` method of the `WindowSetpoint`

WindowSetPoint

This setpoint represents the opening and closing of the windows in the greenhouse. The `consequence` method of this setpoint works like this:

1. retrieves the values of the inside and outside temperature and humidity sensors from the input state

2. looks at the difference between the retrieved inside and outside sensor values

3. adjusts the inside sensor values towards the outside sensor values with a factor depending on the window sizes

4. appends the new value to a constructed list, containing the other untouched sensors

5. constructs a new `ClimaticState` and returns it

The adjustment of the inside sensor values can be seen in Snippet 6.7

6.3.5 GreenhouseHeuristic

The `GreenHouseHeuristic` is an implementation of `Heuristic` that is used in the `AStarAlgorithm`. During the course of developing the greenhouse extension the heuristic has been calculated in many different ways. Some of the ways are explained below.

Rule Desires

Using the rules as a means for generating the heuristic value was the first option tried. The heuristic for a vertex V was calculated by assuming a direct edge from V to the goal vertex. This however gave rise to some complications, because with few rules the heuristic value often ended up close to zero for almost all vertices. The problem with a heuristic value close to zero is that the algorithm "loses its sense of direction" that is, it does not search towards the optimal path, but rather just circling outwards. This can severely damage performance because many more vertices are searched.

Euclidian and Normalizing

Another approach came with the emerge of the `ConstrainingRule`, which is explained in Section 6.4.3. It is based on the Euclidian distance between the states. The value returned from the Euclidian distance often amounted to values far exceeding the value of the actual shortest path. That made this approach seem like an unwise path because the heuristic would be inadmissible. But by using the Euclidian distance from the source to the goal as a divisor, the distance was normalized. This approach also yielded the, by far, fastest calculation time. The code that calculates the euclidian distance can be seen in Snippet 6.8

6.3.6 SystemSettings

A number of settings are needed throughout the system. These settings are primarily related to the different sensor ids needed by the rules. The `SystemSettings` class is a container of such settings and holds all the `SuperLinkID` instances needed by the greenhouse example rules. The individual ids are retrieved by the class from a properties file. This was done to facilitate configuration of the system on-site. (See section 5.1.3 for a description of

```
...
double euclidianDistanceToGoal(ClimaticState state) {
    if (state.partiallyEquals(goal)) {
        return 0;
    }
    double total = 0;
    for (SensorInput sensorFromGoal : goal.getSensors()) {
        double sensorValue = sensorFromGoal.getValue();
        for (SensorInput sensorFromState : state.getSensors()) {
            if (sensorFromGoal.equalsOnSuperLinkID(sensorFromState)) {
                sensorValue -= sensorFromState.getValue();
                total += (sensorValue * sensorValue);
            }
        }

    }
    return Math.sqrt(total);
}
...
```

Snippet 6.8: The `euclidianDistanceToGoal` method in `GreenhouseHeuristic`.

how the system uses metadata) This approach in general also promotes not duplicating code, since all retrieval of ids is located in one place.

For testing purposes, the class facilitates setting the ids programmatically. This eases the testing of especially the rules considerably, since the test can use a general state-template and just set the necessary ids on the `SystemSettings` object.

The class furthermore holds the method, `isTestMode()` for checking if the system is in test mode. (This mode is described further in section 8.1.2)

6.4 Traps, Pitfalls, and Corner Cases

This section reveal some of the traps and pitfalls of the framework. When developing extensions for use in the framework many things can wreak havoc on the planner engine. This is because large parts if the engine are exposed and therefore configurable by the user. The following guidelines, however, cannot cover all corner cases of what can be done with the framework, but they are excellent pointers for what to consider when extending the planner framework. This section will also elaborate on the consequences that can arise when the different aspects of the extension have not been carefully considered.

6.4.1 State Bulk

When constructing a state it is important to consider the state descriptors used. The floating point precision of these descriptors is directly related to how many possible states there are. This is because, when dealing with state spaces where the amount of states is not finite, the consequence functions generate a massive volume of states. This can result in an enormous bulk of states if these, like in the greenhouse extension, are calculated on-the-fly.

An example of the consequences of not having reduced the number of states comes from the greenhouse extension. Decreasing the floating point precision from two to one in the `SensorInput` value, resulted in exponential decrease in plan-length to calculation-time.

The short advice here is; reduce the possible states of the system to minimize the calculation time. This advice is sound no matter if the graph is calculated on-the-fly and potentially infinite or has a finite amount of states. This is because limiting the potential amount of states for the pathfinder to search, will always reduce the calculation time.

6.4.2 Rules and Desire Functions

When developing and implementing rules, there are some things that have to be avoided and some good general practices to follow.

The lower the better and vice versa

When implementing desire functions the output value is the weight added to the edge. This means that the desire function should return a value that is low if the state change is towards a, for this rule, desirable state.

If the same desire function is used for calculating the desire for use when delivering goals to the goal handler, bear in mind that the desire needs to be high if the rule wants change. That is the higher the desire, the higher the probability that this rule decides what to do. In other words, here it represents the *desire for change*.

Avoid directly conflicting rules when possible

When developing rules it is a good idea to avoid direct conflicts between the rules. The best way to illustrate this is by example. Consider the following two rules

temperature limits This non-goal oriented rule will attempt to keep the temperature in between two fixed values, e.g. 15-20 degrees. When the temperature is outside these limits, the weight added to the edges by this rule is immense.

mean temperature This goal oriented rule attempts to keep a steady temperature of e.g. 25 degrees. The further the temperature sways from this value the larger the value added to the edge weights.

Assume the current temperature is 18 degrees and the goal for the planner is set to 25 degrees. Each time the pathfinder tries to go beyond the limits of the *temperature limit rule* it hits a metaphorical brick wall. This can result in the pathfinder never finding a path, or that it will need obscure calculation time to reach it.

Avoid a fixed value as desire function output

If the value output from the desire function of a rule is a fixed value or constant, the rule may not have any influence in the plan.

To illustrate this let us look at the example used above. Assume that the starting temperature is 28 degrees and the goal of the *mean temperature rule* is 30 degrees. Furthermore assume that the value output by the *temperature limit rule* is a fixed value. When the pathfinder then runs, all the edges emanating from the source will be penalized by the same weight. The plan the pathfinder produces will, in most cases, be the same plan as one generated without the use of *temperature limit rule*. So when using fixed values make sure that all other rules primarily operate within the area where fixed values are not applied.

Avoid inverse proportional desire functions

When two, or more, desire functions combined are inverse proportional, they will cancel each other out when calculating a plan. In other words if the combined desire of a set of rules is almost always the same, then they will have no influence on the calculated plan.

Use smooth surfaced desire functions

When deciding upon a desire function it is important to consider the shape of its output graph. The best desire functions are not jagged with big spikes here

and there, but smooth. The smooth surface makes the pathfinder avoid the high penalty areas much better, and this results in a much faster pathfinding.

Avoid local minima in desire functions

When selecting mathematical functions for use as desire functions it is important to avoid functions with local minima and only one global minimum. The reason for this is that a rule with a desire function that has more local minima can make the pathfinder hover around the local minima, which can result in very long, or even infinite, calculation times.

When using functions with only one global minimum and following the above advice of using smooth functions for your desire functions the speed of the pathfinder will be rather good.

Use desire functions with output in same interval

When using rules it is important that they have the same power over the plan. This is accomplished by making sure that the desire functions used by the rules, all generate values within a specific interval. In the greenhouse extension, this was upheld by ensuring that all the desire functions returned values between zero and one. The only time this does not apply is when limits are enforced. The level of enforcement is determined by the size of the value.

Test your desire functions

One of the lessons learned during the greenhouse extension was that it is a good idea, as is the rule when developing software, to test the desire functions for unexpected behavior. The photosynthesis rule is a prime example of this. At first the desire function was only tested with values beneath 80% photosynthesis rate, it did not do anything unexpected. But when putting it into action and a lengthy bug tracking later, it was discovered that the function did not behave very well with values higher than 85%. This was remedied and tests were written to avoid this from happening again.

Sometimes the rules and their desire are not enough. This is the *raison d'être* of the `ConstrainingRule`.

```
    private double normalizedDistance(SensorInput currentSensor,
            SensorInput newSensor) {
        double newValue = newSensor == null ? 0 : newSensor.getValue();
        double currentValue = currentSensor == null ? 0 : currentSensor
                .getValue();
        double normalizer;
        double distance = Math.abs(newValue - currentValue);
        // Normalizes to log 10 value
        if (Math.abs(newValue) > 0 && Math.abs(currentValue) > 0
                && Math.abs(distance) > 0) {
            double flooredNorm = Math.floor(Math.min(Math.log10(distance), Math
                    .min(Math.log10(newValue), Math.log10(currentValue))));
            normalizer = flooredNorm > 0 ? flooredNorm : 0;
        } else {
            normalizer = 0;
        }
        if (normalizer >= 0) {
            normalizer += 1;
            double div = Math.pow(10, normalizer);
            newValue /= div;
            currentValue /= div;
            distance = Math.abs(newValue - currentValue);
        }
        if (distance > 1) {
            distance = 1;
        }
        return distance;
    }
```

Snippet 6.9: The `normalize` method in the `ConstrainingRule`.

6.4.3 ConstrainingRule

A problem appeared, when the greenhouse extension was developed. This problem was that in some cases pathfinder tried to adjust only one setpoint which just kept rising and rising. After lengthy debugging sessions the reason became apparent. The problem was that no rule was governing the setpoint adjustment or the consequence of adjusting it. This brought the idea of the `ConstrainingRule` into play.

The `ConstrainingRule` is a rule that ensures that any change in the environment is connected to a cost. The way it works is that when constructing it the `SuperLinkID` values of the sensors that are to be constrained are given as parameters. The desire function of the rule then observes these sensor values in any state. If any changes happen it then normalizes this change to a value between zero and one and returns that. The rather intricate math to do this can be seen in Snippet 6.9. It works by using the logarithm to reduce the values to something with a difference between zero and one.

6.4.4 Adjustable Combinatorial Explosion

When developing adjustables it is important to be mindful about the abundance of possible settings. If there are many possible adjustments or many possible settings, the amount of calculations done on a per vertex level, can skyrocket.

An example is a temperature adjustable, which can be adjusted within a tenth of a degree. Then raising the temperature two degrees would require a plan of at least 20 actions in size. Furthermore, under the assumption that the pathfinder chooses the shortest path directly, each desire function of each rule has to be calculated once for each possible adjustment of all adjustables. To reduce the amount of calculations significantly, the precision of the setting could be lowered, such that it was only possible to adjust half a degree at a time or even one degree.

When reducing the amount of possible settings it is a good idea to consider what is possible. Even though it is possible to measure temperature in the greenhouse with enormous precision, it does not make sense to have setpoints with such values. It simply does not make sense to control the temperature in such a degree because it does not imply any benefit.

6.4.5 Adjustable Consequences

It is also important that the consequence of an adjustment should have a measurable effect. Measurable means a change that results in a state change. It is important to keep the amount of states in mind here. If the resulting consequence is not measurable, it is possible that the planner cannot find any plan to solve the given problem, because it will run out of vertices to look at.

6.4.6 Long Runtime

Throughout this section, we have looked at ways to reduce the calculation time. That is because it is possible for the planner to take an immense amount of time calculating plans. When not properly calibrated, calculating time of fortyfive minutes and more have been observed. These long calculation times can make it difficult to discern between infinite loops and "just" long calculations.

Also if time is of the essence then following the above given guidelines can mean the difference between success and failure.

The advice here is reducing the calculation time by any means necessary. Remember that reducing the calculation time will often result in a less precise plan or behavior. Heed to make reasonable compromises when doing so.

Chapter 7

Development Process

> "However beautiful the strategy, you should occasionally look at the results."
>
> — Winston Churchill (1874-1965)

This chapter is dedicated to the development process of the project. The methodology and strategies that were used are described. Furthermore, the rationale for fully automating all aspects of the process is explained, along with an account for how this was accomplished.

7.1 Agile

This project has been a reconnaissance mission. The goal was to explore the possibilities of introducing game AI technology into the world of the greenhouse. As such, it would have been foolish to adopt an old-fashioned rigid development process, like the Waterfall process or Rational Unified Process (RUP),[27] because they preach a strict flow in development. Each cycle of development is finished before the goal, or sub-goals, can be modified to fit changes in requirements or new discoveries. Much like navigating a minefield, only based on a year old aerial photo; probably not a good idea.

Because we did not know exactly where we were going, we knew the design of the system probably needed to be revised several times, which of course it did. Therefore, instead of pursuing a rigid methodology, we chose to follow the path of *agile*.

The notion of agile spawned in the mid 1990's as a *lightweight* contrast to the *heavyweight* methodologies dominant at the time. The focus moved from the highly pre-planned development processes, where progress is measured in terms of deliverable artifacts - requirement specifications, design documents, test plans, code reviews and the like - to a highly adaptable process, where working software is delivered frequently and is the principle measure of progress. The essence of agile development can be found in the *Agile Manifesto*.[28]

Agile methods are highly diversified, comprising a family of development processes. This project was developed as a fusion of several agile methods. First and foremost, all coding was done as a pair-programming effort. The design of the system has emerged from ruthless and continuously testing of all functionality. Some ideas from the *Crystal Clear* process was originally used, but when the development focus switched to the Avian Game Platform, *SCRUM* was adopted.

7.1.1 Methodology

A quick overview of the two methodologies adapted within the development of the project.

Crystal Clear

The primary artifacts of the Crystal Clear development process are:[29]

- frequent delivery: of useable code to users
- osmotic communication: developers are located close to each other to facilitate communication
- reflective improvement: keep track of problems and solutions, and reflect on them

A nice technique from Crystal Clear is *blitz planning*. The essence of blitz planning is brainstorming. The different tasks of a project, and their interdependencies, is identified and laid out on a table. This procedure simplifies planning, as its makes missing items immediately evident and displays the "natural" order in which to approach the tasks.

SCRUM

As mentioned, the development adapted some elements from SCRUM. The most important was the notion of a *backlog*. The set of tasks that are remaining in a project is the *product backlog*. An iteration in SCRUM is called a *sprint*, and the remaining tasks of a sprint is the *sprint backlog*.

When a project is broken down into simple and manageable tasks, it helps you focus on the tasks at hand. Keeping the focus is great way of speeding up development. Another feature of the backlog is that it illustrates the development rate.

SCRUM also dictates daily *SCRUM meetings*[1], where each person present answers the same three questions:

1. What did you do yesterday?
2. What have you planned for today?
3. Is anything impeding your work?

It is of course ridiculous to have a daily meeting when there is only the two of us, and we are doing pair-programming, but during the time we worked at Rising Tide, it was a very useful method for eliminating duplicate work, getting help quick, and sharing ideas.

7.1.2 Pair Programming

The idea of pair-programming is stolen from *eXtreme Programming* (XP).[30] The benefits of programming in pairs are numerous; just to name a few:

- Higher discipline: one tends to "do the right thing" first time around.
- Better design: constantly bouncing design ideas off the partner results in a more well thought out design.
- Collective code ownership: every developer has working knowledge of the entire code base.
- Mentoring: the sharing of knowledge is fluent.

[1] In Crystal Clear it is just called the daily *stand-up meeting*.

The entire project is done as a pair-programming effort. The ease of which any design issue, code problem, or algorithm quirk is discussed and solved, is simply not possible in any other way.

Even pair programmers doing agile have to use some tools to make everything work like a well oiled machine though. Let's look at a few of those tools and techniques.

7.2 Version Control

Version control is a time machine! It's that simple.

It gives the developer the freedom and confidence to go with any idea that comes to his mind, no matter how wild it is: everything can always be rolled back if it fails.

The project was given a Subversion repository on a server at the University of Southern Denmark. Besides perhaps the `javac` compiler, this has been the most valuable asset, bar none, for this project.

In addition to the obvious Java source files, **everything** needed to build the project has been checked into version control, and has been updated with newer versions as they came along. This includes build scripts and configuration scripts, such as Ant's `build.xml` and the CruiseControl `config.xml`, as well as external libraries, such as JUnit and Log4j. Section 7.3 gives the rationale for versioning external libraries that in principle just as easily could have been downloaded again.

All non-autogenerated documentation was also checked into version control, along with articles, papers, licenses, and ideas scribbled down in notepad. This of course includes this writing as well. Since it is written in LaTeX the "source" is plain text files, which are "easy as pie" for any version control system to handle. The same freedom applies to ordinary text as to Java source files; you are never worried about changing or deleting anything, since it can always be rolled back.

7.2.1 Structure

The repository is structured as follows. The top folder is the project name, *sidious*. That folder is split up into three different, but equally important folders, each with specific responsibilities. Figure 7.1 shows the structure of the repository.

- `trunk` holds the main development line of the project.
- `branches` holds the release branches of this project; which as of this writing is empty, but when this project is delivered, will hold the thesis delivery.
- `tags` holds tags of the project; e.g. when demos and large refactorings took place.

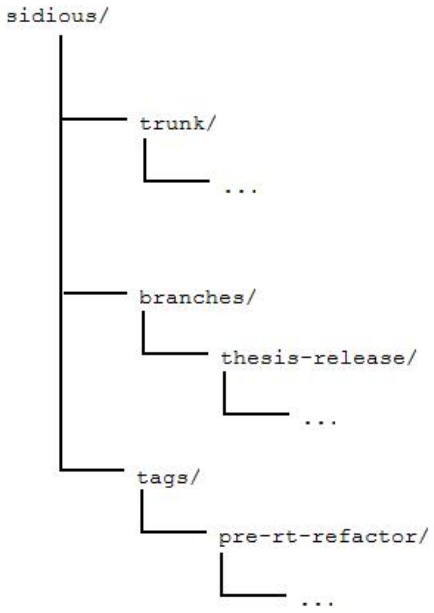

Figure 7.1: The sidious project trunk and branches

7.3 Automation

One of the non-functional requirements of this project was that **all** aspects of its development was to be automated. From compiling and deployment, through testing and instrumentation, to generation of documentation, everything can be, and is, automated. A dedicated Ant script is responsible for executing all the above tasks.

The project has a specific directory structure, which enables separation of test and production[2] source files into different directories but mirrored by the Java package structure. This means that test files and production source files are in the same Java package but in different directories. The structure is shown in Figure 7.2. The `src` directory contains the production source code, the `test` directory contains the test files, and the `vendor` directory contains all external libraries needed, e.g. jar files for junit, log4j etc.

The `build` directory, which is left outside version control, contains all artifacts that were built in the process. The reason for this directory is explained in a moment.

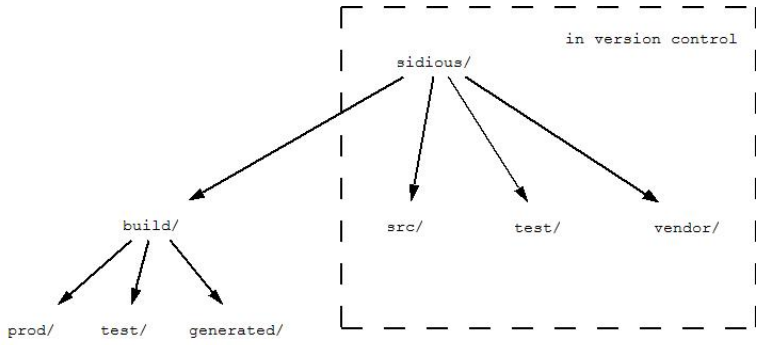

Figure 7.2: The directory structure of the sidious project

One of the reasons to automate the entire process is to make it *complete* and *repeatable*. Completeness means that the process is self-sufficient; the developer only needs to "turn the crank" and everything is executed as the recipe prescribes.[3] Repeatability ensures that the exact same build can be generated again at any time in the future. To ensure completeness and repeatability, the entire above structure is placed under version control. Section 7.2 descripes in further detail how version control was used in this project. Even though all external libraries can be downloaded again, placing them under version control helps ensure that the build is complete and repeatable, because the build is then not dependable of a specific version of a library being available still.

[2]The distinction between production files and test files is to have clear separation of concerns.

[3]It can be a good analogy to think of the script as a recipe for baking crisp new builds.

Ant scripts provide the notion of *targets*, which are a way of specifying which parts of the script to be run. Each target can have dependencies to other targets, which then are run before the specified target is. This chain-of-dependencies enables the script to make the process complete and repeatable.

7.3.1 Building

When the script is run, all source files in the src and test directories are compiled into class files and placed in the `build/src` and `build/test` directories respectively. Again, the production files and test files are separated. Keeping these files separate simplifies shipping of the finished product. The build-target also includes generation of documentation; e.g. Javadoc, HTML view of the source code, and code analysis.

These artifacts are built into the `build/generated` directory. That way compiled class files and generated documentation, test results, and code analysis results are kept apart.

7.3.2 Testing

The default target of this project is `test`. The chain of dependencies of the test-target ensures that all production and test files are compiled, and that all unit tests are executed. See Snippet 7.1.

As the Snippet shows the integration tests are left out. This was done to make the execution of the unit tests as fast as possible; if executing the tests becomes too slow, developers tend to "forget" to run them.

7.3.3 Deploying

Automating deployment of a project would normally include generation of a jar or zip file, or perhaps deploying the product on an application server. This project does not require any explicit deployment. However, since the documentation of the project is automatically generated, and since this writing is done in LaTeX, which also needs compiling, the `deploy` target of the Ant script does the following:

1. Ensures that all production and test source code is compiled.
2. Ensures that all static code analysis is run.

```
<property name="generated.doc.junit"
    location="${generated.doc.dir}/junit" />
<target name="test" depends="compile-tests"
    description="Runs all tests">
    <delete dir="${generated.doc.junit}" />
    <mkdir dir="${generated.doc.junit}" />
    <junit
        haltonfailure="true"
        fork="yes"
        printsummary="on"
        forkmode="perTest">
        <classpath refid="project.classpath" />
        <jvmarg value="-ea" />
        <formatter type="xml"/>
        <batchtest todir="${generated.doc.junit}">
            <fileset dir="${build.test.dir}"
                excludes="**/*IntegrationTest.class"
                includes="**/*Test.class" />
        </batchtest>
    </junit>
    <junitreport todir="${generated.doc.junit}">
        <fileset dir="${generated.doc.junit}">
          <include name="TEST-*.xml" />
        </fileset>
        <report todir="${generated.doc.junit}" />
      </junitreport>
</target>
```

Snippet 7.1: Test target in the Ant build script. (build.xml)

3. Ensures that all documentation is generated.

4. Compiles this writing into a PDF file.

5. Copies all the above generated artifacts into the designated `delivery` folder, where it is then ready to be burned to a CD-ROM, which is enclosed with this document.

6. Copies the contents of the `delivery` folder to the server hosting the project homepage, where it is then readily available.

7.3.4 Continuous Building

Although maybe overkill for this project[4], a dedicated build machine was set up, to continuously build the project and run the tests. Nevertheless, it adds to the confidence that everything is working all the time. The machine[5] was setup with CruiseControl to run the scheduled building and testing. CruiseControl is a framework for running continuous integration. See Appendix B for a reference. It supports scheduled execution of Ant tasks, various possibilities for publishing the results, combined with source control tools. It was configured to wake up every thirty minutes, checkout a fresh copy of the source code from the repository, build it, and run all the tests. If no changes was made to the repository, it merely went back to sleep. If the build was successful, the developer who made the last commit was notified by email; if the build failed all developers (both of us) were notified by email. All necessary scripts to setup and configure CruiseControl is located in the `automation` folder in the project.

[4]Not least because the project is developed as a pair-programming effort.
[5]A very old Pentium III 1GHz with 256MB RAM

Chapter 8

Testing

"I have not failed. I've just found 10,000 ways that won't work."

— Thomas Alva Edison (1847-1931)

This chapter is started with the incentive for testing the way we have done, our experience with writing the tests, and some arguments for "loosening" a good object-oriented programming practice. This is followed by a walkthrough of the main test setups and their results.

8.1 Coding With Confidence

The following lines explain the importance of writing, running, and maintaining tests.

8.1.1 Unit Testing

Unit testing has been an invaluable friend and reliable companion throughout the entire development of the system. When all unit tests run, it automatically induces confidence in whatever task the developer has finished really works and is valid, and additionally leads to a higher appreciation of the code base as a hole.

When "the bar is green"[1] it feels liberating. You can let go of the previous

[1]"Keep the bar green, to keep the code clean" is a saying when using JUnit. When all the unit tests run without failure, the GUI version of JUnit shows a green bar.

task, because you know it works, or at least doesn't break anything. That provides the necessary mental capacity to solve the next task without having to deal with any "skeletons in the closet".

Proof

Unit tests are the developer's way of proving that a piece of code actually does what it's supposed to do – or at least what the developer thought it should.

Collateral Damage

When developing an application the natural way is to add features one at a time, incrementally. When a developer is adding a feature or correcting a bug, he is changing the code base and, consequential, also risks creating a new bug or destroying some other possibly unrelated part of the system. Unit tests ensure that everything works at least as well as before adding the new feature or fixing the bug.

Additionally, if a developer discovers a bug and fixes it, he should always write a new unit test that catches that bug so bugs only pop up once!

Executable documentation

Another way of thinking about unit testing is as *executable documentation*. When writing a unit test to accompany a piece of code, the developer at the same time states the intended use of that piece of code. This statement also goes for an entire API. When an API is delivered with the unit tests as chaperon they serve as examples of how to use the API.

Trust

In the authors' experience unit testing has the effect that one simply does not trust a piece of code that doesn't have any unit tests. Ones own code or anybody else's.

The Privates are Showing

Good object-oriented practice dictates classes to "hide their privates". However, to be able to test thoroughly it is often useful to have access to otherwise

private methods; e.g. methods used in composite calculations, where only the composite is the public exposed method. Java does not provide a straightforward way to test private member methods[2]. A possible solution is to make the test case a private member class. This would expose the private methods to the test case class, but it would also introduce horrible code clutter to the class as well as coupling those classes forever.

This project was build with a parallel package structure, allowing the test case classes to be within the same Java package as the classes they are testing, but in another physical folder. The details are explained in section 7.3. The "private" methods needed testing had then their access level set to *default*. This approach allows the methods to be accessed from classes within the package, and hence, to be accessed from the test case classes, but the advantage is that it does not propagate those methods to the public exposed API.

8.1.2 Integration Testing

As mentioned earlier in the discussion of automation, integration tests were a separate target from the default `test` target of the Ant script. The integration testing of the system entailed exercising the entire system; from the first call to `requestPlan` to a plan is finally delivered. This is potentially very long calculations, and therefore not feasible for the default test target as it is paramount that the tests can be executed often. The integration tests was of course a part of the continuously build process on the dedicated build machine.

Integration testing can be done at a more fine-grained level, but for this system, the unit tests could fully cover testing the integration issues between packages.

Furthermore, to ease the complexity of writing the tests several *mock* objects were developed. A mock object is an object that imitates the behavior of another object, but is more easily used or can be controlled more specific. All developed mock objects are located in the `dk.deepthought.sidious.mock` package.

[2]C++ provides the notion of *friends*, which allows for testing of private methods and members. Nevertheless, they must be stated as a friend of the class to be tested, and hence add additional code and coupling to the class. C# provides the "internal" access modifier, which is roughly equivalent to Java's default level.

To further assist the integration tests, a special *test mode* can be set. This mode is set on the `SystemSettings` object and is used to limit the generation of plans beyond the `GoalHandler`. This has the side-effect that the initiating `PlanRequester` is not removed from the `requesterMapping` located on the `BlackBoardEngine`, and later requests for that specific requester will then not fail.

8.2 Test Overview

The following gives an overview of how the system was tested, from the small unit tests, only testing simple functionality, to the integration tests, that fire up the entire system.

Only a small excerpt of all the tests is shown in this section. This was decided, since the tests are written in the same general manner throughout the project, which will be explained in the next section, and since all tests can be easily found, as they have the same name as the class they are testing, with "Test" appended; e.g. the test case for `Step` is named `StepTest`. This excerpt poses only to give a general outline of the tests written.

As mentioned in Section 7.3, the test and production source files are placed in separate physical folders, but in the same logical Java package. This means that the test classes are to be found in the `test` folder, relative to the project root.

Furthermore, since JUnit are set to deliver the results of the tests as HTML files, all test results can be browsed. Appendix A explains were the results are located.

8.2.1 Unit Tests

This section describes some of the unit tests made for the system.

As the code coverage clearly shows all parts of the system has been "touched" by the tests. Unfortunately, this only shows that all parts of the system has been exercised by the tests, but not if the methods has been tested thoroughly.

By thoroughly we mean not only testing the "*on a calm sunny day, in the middle of the road*"-case, but whenever possible checking the *Right-BICEP*.[31] See Table 8.1 for the acronym.

Right	Are the results **right**?
B	Are the **boundary** conditions correct?
I	Can the **inverse** relationship be checked?
C	Can the results be **cross-checked** using other means?
E	Can **error conditions** be forced to happen?
P	Are **performance** characteristics within bounds?

Table 8.1: The Right-BICEP

It has been the goal of all tests written to abide to the Right-BICEP principle whenever possible. The following are an exemplary subset of tests that were written.

ClimaticStateTest

This test example shows the `average` method of the `ClimaticState`. As Snippet 8.1 shows, the method is first tested for the **right** condition by evaluating to a known average, then tested **error conditions** by passing `null` and an empty list, and then for **boundary conditions** by passing states with no sensors. The method does not have any grounds for testing the inverse relationship or performance characteristics.

TemperatureRuleTest

The `TemperatureRuleTest` follows the Right-BICEP principle, but it also shows how to circumvent the *rule properties* construction. As it can be seen in Snippet 8.2, the property object of the rule can be passed along to a *static factory* method on the rule. This allows for the flexibility of properties files describing the rules, and at the same time makes the testing a lot easier.

The temperature rule furthermore acts as a boundary guardian for the environment. As such, its desire method return VERY high values when the system evaluates outside the boundaries. This is also reflected in the test, by securing that the desire method returns the specified value.

AStarAlgorithmTest

When testing the `AStarAlgorithm` class a completely new graph was implemented. This graph, called `TestGraph`, was used as both a means to prove

```
public final void testStateAverage() {
    ArrayList<ClimaticState> listOfCS = TestBuilder.buildClimaticStateList(
            10, 10, 0, 1);
    // Right: know the average is 4.5
    ClimaticState returnState = (ClimaticState.average(listOfCS));
    for (SensorInput input : returnState.getSensors()) {
        assertEquals(4.5, input.getValue());
    }
    // Test for null
    try {
        ClimaticState.average(null);
        fail("null was accepted");
    } catch (IllegalArgumentException e) {
        assertTrue(true);
    }
    // Test for list with no sensors
    try {
        ClimaticState.average(TestBuilder.buildEmptyStateList());
        fail("empty was accepted");
    } catch (IllegalArgumentException e) {
        assertTrue(true);
    }
    ClimaticState emptyState = TestBuilder.buildClimaticState(0, 0);
    int sensorAmount = 0;
    ArrayList<ClimaticState> stateList = TestBuilder
            .buildClimaticStateList(10, sensorAmount, 0, 2);
    // Test for empty sensor list
    assertEquals(emptyState, ClimaticState.average(stateList));
}
```

Snippet 8.1: The test of the `average` method of `ClimaticState`.

```
public final void testCalculateDesire() {
    SuperLinkID ID = new SuperLinkID("testCalculateDesire");
    Properties prop = new Properties();
    prop.setProperty("sensor_id", "sensor0");
    prop.setProperty("t_min", "5");
    prop.setProperty("k_min", "0.5");
    prop.setProperty("t_max", "30");
    prop.setProperty("k_max", "0.5");
    prop.setProperty("t_mean", "18");
    // Test happy day for static factory
    TemperatureRule tempRule = TemperatureRule.constructTemperatureRule(ID, prop);
    // Happy day
    double desire = tempRule.calculateDesire(20);
    assertTrue(desire <= 1 || desire >= 0);
    // Mean temperature test
    desire = tempRule.calculateDesire(18);
    assertEquals(0.0, desire);
    // Should return 1000 when outside boundaries
    desire = tempRule.calculateDesire(35);
    assertEquals(1000.0, desire);
    desire = tempRule.calculateDesire(-3);
    assertEquals(1000.0, desire);
    // Further away from mean implies lower desire
    double desire1 = tempRule.calculateDesire(22);
    double desire2 = tempRule.calculateDesire(25);
    assertTrue(desire1 <= desire2);
}
```

Snippet 8.2: Test of the `desire` method of the `TemperatureRule`.

```
public void testJAStar4Vertices4Edges() {
    // 4 Vertex 4 edge Graph with shortest path 3 vertices long
    Collection<TestAdjacent> adList4 = new ArrayList<TestAdjacent>();
    adList4.add(new TestAdjacent(new TestState(0, 0), new TestState(1, 1),
            2));
    adList4.add(new TestAdjacent(new TestState(0, 0), new TestState(2, 1),
            5));
    adList4.add(new TestAdjacent(new TestState(1, 1), new TestState(2, 1),
            2));
    adList4.add(new TestAdjacent(new TestState(1, 1), new TestState(2, 2),
            4));
    TestGraph g4 = new TestGraph(new TestState(2, 1), new TestState(0, 0),
            adList4);
    Pathfinder p4 = new AStarAlgorithm();
    p4.search(g4);
    assertEquals("2 edges from start to goal failed", 4.0, g4.getGoalVertex()
            .g());
}
```

Snippet 8.3: Test of the `search` method of the `AStarAlgorithm`.

that the algorithm had been implemented correctly, but also as proof that any graph implementation could be used when pathfinding.

The `TestGraph` makes use of a class called `TestAdjacent`, which is a representation of an adjacency list. In `TestGraph` the vertices represent a state called `TestState` which represents a set of coordinates (x, y). All the tests work as seen in Snippet 8.3. First an adjacency list is created, then the graph is contructed, and finally the graph is searched. The cost of the path found is then checked by extracting the cost value from the goal vertex.

8.2.2 Integration Tests

With the parts of the system quality tested, it is time to put it all together and see all wheels turning in the so-called *integration tests*. Integration testing is done to verify that all parts of the system interact and that performance is up to specs. There are different approaches to integration testing. One approach where only parts of the system is integrated and tested. The approach chosen to test the greenhouse extension is one where the entire framework and parts of the extension is tested at once. At the beginning, it is a good idea to only test simple problems and then gradually increase the difficulty level.

A wide array of different setpoints and rules has been tested against each other, and the following is only a small excerpt of the tests that have been run. These excerpts have been chosen to demonstrate that the essentials

```
private void finished(Greenhouse greenhouse) throws InterruptedException {
    long startTime = System.currentTimeMillis();
    while (!greenhouse.isFinished()) {
        Thread.sleep(10);
    }
    long end = System.currentTimeMillis();
    Plan plan = greenhouse.getCurrentPlan();
    assertTrue("Plan was empty", plan.size() > 0);
    System.out.println("-----------------------------------------------------");
    System.out.println("-----------------------------------------------------");
    System.out.println("Plan of size=" + plan.size() + " finished in "
            + (end - startTime) + " milliseconds");
    System.out.println("-----------------------------------------------------");
    Stack<Step> steps = plan.getSteps();
    while (!steps.isEmpty()) {
        Step step = steps.pop();
        System.out.println(step);
    }
}
```

Snippet 8.4: The `finished` method of the integration test.

work. The reader is encouraged to alter the values and run the tests to check the validity. This should be fairly easy, and is explained in the How-To section of Chapter 6.

The `finished` Method

Because the system runs in multiple threads, and JUnit does not provide a native way of testing with multiple threads, the `finished` method was conceived. When a plan is delivered back to a `GreenHouse`, a `finished` flag is raised on that `GreenHouse`. The `finished` method just sleeps for 10 milliseconds, wakes up and checks that flag; if the flag is not yet set it just goes back to sleep, and if the flag is set the rest of the method is executed: The plan is retrieved, checked to be non-empty, and then printed to the console. The method is shown in Snippet 8.4.

The following test cases are all located in the `PathfinderIntegrationTest` class, which is located in the `dk.deepthought.sidious.planner` package.

testIntegrationSimple

The first integration test is a simple test involving the `HeaterSetpoint` and the `TemperatureRule`. The test itself can be seen in Snippet 8.5.

The test itself is started with adding the `TemperatureRule` and the

```
public void testIntegrationSimple() throws Exception {
    SuperLinkID id = new SuperLinkID("testIntegrationSimple");
    // Add adjustables
    Collection<Adjustable> adjs = new ArrayList<Adjustable>();
    adjs.add(new HeaterSetPoint(18));
    // Add rules
    Collection<Rule> rules = new ArrayList<Rule>();
    TemperatureRule temperatureRule = new TemperatureRule(id);
    rules.add(temperatureRule);
    rules.add(new ConstrainingRule(SystemSettings.getHumidityID(),
            SystemSettings.getTimeID()));
    // Creating current state
    SensorInput temperature = new SensorInput(SystemSettings
            .getTemperatureID(), 18);
    SensorInput hum = new SensorInput(SystemSettings.getHumidityID(), 50);
    SensorInput time = new SensorInput(SystemSettings.getTimeID(), 0);
    ClimaticState source = new ClimaticState(Arrays.asList(temperature,
            hum, time));
    // Setting sensors on service engine
    ServiceEngine.setSensorList(source.getSensors());
    // Request plan
    Greenhouse req = new Greenhouse(id, adjs, rules);
    BlackBoardEngine.getInstance().requestPlan(req);
    finished(req);
}
```

Snippet 8.5: Simple Integration test.

`HeaterSetpoint`, with a setting of 18, to the greenhouse. The environment is also initialized with a humidity of 50% and a temperature of 18 degrees. Then a plan is requested by the greenhouse and the system is running. In the system the `TemperatureRule` produces a goal of 22 degrees and then a plan is generated. The output, in a simplified form, can be seen in Snippet 8.6

From the output, it can be seen that the HeaterSetpoint is not entirely correctly modeled. However, this does not detract from the fact that the plan found is sound as the temperature setpoint is rising.

testIntegrationWithConflictingRules Involving Morningdrop

These test show what happens when running the planner with two conflicting rules. The `MorningDropRule` and the `TemperatureRule`. The `MorningDropRule` is a rule trying to achieve a drop in temperature early in the morning. The `TemperatureRule` on the other hand tries to achieve a mean temperature that is a lot higher. The test is initialized just like in the test above with the exception that the `HeaterSetpoint` setting and the environment tempera-

```
Plan of size=8 finished in 93 milliseconds
Step[adjustables=[HeaterSetPoint[setting=19.0]]...]
Step[adjustables=[HeaterSetPoint[setting=20.0]]...]
Step[adjustables=[HeaterSetPoint[setting=21.0]]...]
Step[adjustables=[HeaterSetPoint[setting=22.0]]...]
Step[adjustables=[HeaterSetPoint[setting=23.0]]...]
Step[adjustables=[HeaterSetPoint[setting=24.0]]...]
Step[adjustables=[HeaterSetPoint[setting=24.0]]...]
Step[adjustables=[HeaterSetPoint[setting=24.0]]...]
```

Snippet 8.6: Simple Integration test.

```
Plan of size=10 finished in 9609 milliseconds
Step[adjustables=[HeaterSetPoint[setting=23.0], CO2SetPoint[setting=850.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=24.0]]...]
Step[adjustables=[HeaterSetPoint[setting=24.0], CO2SetPoint[setting=950.0]]...]
Step[adjustables=[CO2SetPoint[setting=950.0], HeaterSetPoint[setting=25.0]]...]
Step[adjustables=[CO2SetPoint[setting=950.0], HeaterSetPoint[setting=26.0]]...]
Step[adjustables=[CO2SetPoint[setting=950.0], HeaterSetPoint[setting=27.0]]...]
Step[adjustables=[HeaterSetPoint[setting=27.0], CO2SetPoint[setting=850.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=28.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=28.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=29.0]]...]
```

Snippet 8.7: Integration test with conflict between `MorningDropRule` and `TemperatureRule`.

ture are 14 degrees. Furthermore, the `MorningDropRule` is added in one the tests. The output of the two tests can be seen in a simplified form in Snippet 8.7 with the conflict and in 8.8 without.

It can be seen that the `MorningDropRule` forces the `HeaterSetpoint` to have a lower setting longer. Namely it stays at 24 degrees for two steps in the plan with `MorningDropRule` compared to the one without. It is also visible that the calculation time without a conflict is shorter.

testIntegrationWithManyAdjustablesAndConflictingRules

In this final test the `PhotosynthesisRule` is tested. This shows whether or not the system will work with the more complex rules and goals posed by more complicated rules. This test also succeeds, but shows some of the flaws in the rule making at the same time. The model in the `PhotosynthesisRule` must be calculated many times, which is the primary reason for the long calculation time, of beyond a minute on a very fast dual core machine. The

```
Plan of size=10 finished in 344 milliseconds
```

```
Step[adjustables=[HeaterSetPoint[setting=23.0], CO2SetPoint[setting=850.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=24.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=25.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=26.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=27.0]]...]
Step[adjustables=[HeaterSetPoint[setting=27.0], CO2SetPoint[setting=950.0]]...]
Step[adjustables=[HeaterSetPoint[setting=27.0], CO2SetPoint[setting=850.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=28.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=28.0]]...]
Step[adjustables=[CO2SetPoint[setting=850.0], HeaterSetPoint[setting=27.0]]...]
```

Snippet 8.8: Integration test without conflict between `MorningDropRule` and `TemperatureRule`

test generates a plan consisting of 16 steps.

For a pointer to where test results can be found, see Appendix A.3.

Chapter 9

Discussion and Conclusion

"A conclusion is the place where you got tired of thinking."

— Arthur Bloch (1948 -)

This final chapter discusses the outcome of the project, the major lessons learned, where the project could go from here, and finishes with the conclusion. The time for reflection has come.

9.1 Discussion

Throughout the development, some unforeseen issues presented themselves. Some of the smaller issues and possible workarounds are described in Section 6.4.2. Large-scale issues have been considered, discussed, and addressed throughout the development of the framework. For instance much consideration has been put into how much of the core functionality, of the framework, should be exposed.

9.1.1 Extendability vs. Usability

When exposing much of the inner workings and logic to developers who wish to use the framework there is the advantage of being able to use it for things never anticipated by the original developers. This is a great advantage but it also gives rise to some major drawbacks. First, there is the problem that extension developers may find the framework overwhelming when first

approaching it. This can be countered by having some examples and default values to tinker with. The greenhouse extension is an excellent sandbox for developers to tamper with to see just how far they can go when using the framework. Besides that, they serve as excellent templates for how to build the different extension components.

Another problem with exposing a lot of the core functionality to the developers of framework extensions is; they can break the system in ways never anticipated by the original developers. This is because, when you expose the logic, you also expose the possibility to create flawed logic, which may cause deadlocks, horrible performance, or no performance at all. To counter this, specific parts of the system can be encapsulated. The first parts that are prime candidates for encapsulation are the parts most prone to be implemented erroneously. This will of course give rise to the debate about which are the most delicate parts, and eventually the framework may end up being very inflexible. To avoid inflexibility there will be times when encapsulation is not an option. This is where good documentation saves the day. As stated earlier, all elements of the framework, exposed as well as private, are well documented through Javadoc. Furthermore, this writing provides answers to potential "challenges", or at least cautions against them. The *Traps, Pitfalls, and Corner Cases* section from Chapter 6, explains where it is likely to encounter these challenges, and how to conjure counter spells.

During the development of the framework, keeping the above advice in mind at all times was attempted; combining encapsulation and good documentation, to help extension developers find a balance between risk and flexibility. These lessons are not only applicable for this framework but also important for framework developers in general. The intended users of these frameworks will then reap the benefits.

Refactoring to Game

During the refactoring to the Avian Game Platform, many lessons on how to purvey the usage of the framework were learned. This was because it was the first direct contact with people supposed to use and extend the framework. The questions asked by the team at Rising Tide gave invaluable insight on what, and how, framework extension developers think, and what they expect from an AI framework.

The need for a highly flexible framework was expressed. This is one of

the main reasons for the framework ending up as open and flexible as it is. The advantages of the highly flexible design far outweighed the risks, as the framework developers would always be close by to answer questions and help debugging the extension.

9.1.2 Other Approaches

During development, it became clear that other approaches could have yielded good results too. One of the major problems was that the rules and especially their desire functions tended to require much more tweaking than first anticipated. One way this could be avoided would have been to go with a fully-fledged expert system. This would then probably reduce the amount of time spent tweaking the rules. The benefits gotten from using a planner however far outweigh the benefits of an expert system.

Because formulating the rules for the system can be a complicated task, it could be interesting to apply evolutionary algorithms to the greenhouse problem domain. As described in Section 3.2, genetic algorithms can be used to mine data and generate fuzzy rules. One of the ongoing tasks in the greenhouse domain is to collect data sets, coupling the climatic state of the plants through their growth period, to the end-quality of the same plants; and to have it done automatically. As viable collected data amounts, genetic algorithms could be used to mine the data and generate fuzzy rules for an expert system. This expert system could then be applied as the *rule engine* for the planner. This approach would complete a circle, where artificial intelligence were monitoring, assessing, and controlling the growth cycle of plant life.

9.2 Conclusion

The Goal

This project was developed with the ambition that the result should be a well-tested, easy maintainable, rock-solid piece of software; that applied game AI to solve the climate control problem.

The Challenges

While fulfilling this ambition, we encountered a few challenges.

The formulation of rules and their desire functions proved to be particularly challenging; if not formulated meticulously, the result can be extreme calculation-time. Additionally, by combining multiple rules and adjustables, the result can be non-deterministic.

The Solution

The primary solution to the challenges of the framework is to be VERY careful when formulating the rules of the system. Excellent guidelines for succeeding in rule building are outlined in the Traps, Pitfalls, and Corner Cases section. However, the best solution is to "be mindful of your rules".

The Success

The project clearly proves that the framework can be used to solve the climate control problem. It provides basic solutions to the drawbacks of the IntelliGrow system. With aid from climate control professionals, the system can function as a fully-fledged decision support system on a real production line. Since this was the ultimate goal of the thesis, it is the authors' opinion that the mission was accomplished.

The Future

The grand vision of IntelliGrow seems to be within grasp, with a little help from this framework.

The End

The authors do hope you have enjoyed the show; it has been a great pleasure making it.

Appendix A

Resources

Easy Access

The root folder of the enclosed CD-ROM and the project homepage:
`http://www.deepthought.dk/sidious/`
both contain an `all.html` file. This file contains links to whatever generated resource needed, and is provided to simplify browsing of the resources of the project.

A.1 Source Code

The source code is available by the following means:

- On the enclosed CD-ROM, in the directory: `src`
- In its SVN repository at:
  `https://svn.mip.sdu.dk/master-mb/sidious/branches/thesis`[1]
- As browseable HTML pages at:
  `http://www.deepthought.dk/sidious/html`

The above copies of the source code will be mirrors of each other, and will represent a frozen snapshot of the project as of: 1. May 2007.

[1] This repository requires username and password, which can be acquired by contacting any of the authors.

The snapshot will be branched off the trunk of the project, so that further development can be made to the trunk, a.k.a. working copy, without disturbing the delivered copy. (Section 7.2 outlines the structure of the repository.)

A.1.1 Working Copy

The working copy of the source code in the repository will not be frozen at the designated time, and will be subject to change thereafter. The working copy can be found at:
`https://svn.mip.sdu.dk/master-mb/sidious/trunk`

A.2 Documentation

The full Javadoc of project sidious is available by the following means:

- On the enclosed CD-ROM, in the directory: `doc`.
- On the project homepage at:
  `http://www.deepthought.dk/sidious/doc`

The documentation will reflect the snapshot of the source code as described in the previous section.

A.3 Test Results

All test results are available both on the enclosed CD-ROM and online on the project homepage. The `test` folder contains an `index.html` file, which lets you browse through all the tests.

A.3.1 Integration Test Results

The results of the integration tests are found in the `PathfinderIntegrationTest`. This can be browsed to via the `index.html` page in the `test` folder as mentioned above. The generated plans can be seen by clicking the `System.out`-link in the bottom right corner.

A.4 Static Code Analysis

The results of the static code analysis tools executed upon the source code of the project can be found in their respective folders, relative to the root folder of both the CD-ROM and the project homepage. They are:

JDepend : `/jdepend/report.html`

JavaNCSS : `/ncss/report.html`

EMMA : `/coverage/coverage.html`

Again, it would be easier to browse to them via the `all.html` of the root folder.

Appendix B

External Libraries

The following accounts for the various external libraries used in development of this project. It covers what the library was used for and where to get it.

Ant the well known building tool for Java.
Can be found at: `http://ant.apache.org/`

JUnit is a testing framework for regression testing in Java.
It can be found at: `http://www.junit.org/index.htm`

Log4J is a logging package for Java.
Can be found at: `http://logging.apache.org/log4j/docs/`

JCIP Annotations is a annotation package for documenting thread-safety policies. It can be found at:
`http://www.javaconcurrencyinpractice.com/`

CommonsLogging is a logging package that acts as a thin bridge between different logging frameworks.
It can be found at: `http://jakarta.apache.org/commons/logging/`

Jakarta ORO is a set of regular expression classes for text processing. The package is used by Commons Net.
Can be found at: `http://jakarta.apache.org/oro/`

Commons Net is an implementation of, among others, client side FTP protocol. It is used by Ant to provide FTP capabilities.
It can be found at: `http://jakarta.apache.org/commons/net/`

Java2HTML is a package for converting Java source files into browsable HTML pages.
Can be found at: `http://www.java2html.de/`

JDepend is a tool for generating design quality metrics in Java.
It can be found at:
`http://www.clarkware.com/software/JDepend.html`

JavaNCSS is a suite for source measurements in Java.
It can be found at: `http://www.kclee.de/clemens/java/javancss/`

EMMA is a code coverage tool for Java.
It can be found at: `http://emma.sourceforge.net/`

CruiseControl is a framework for running continuous integration.
It can be found at: `http://cruisecontrol.sourceforge.net/`

Appendix C

Larger Versions

C.1 Sequence Diagram of the Planner

This figure is the enlarged version of Figure 4.7 on page 38.

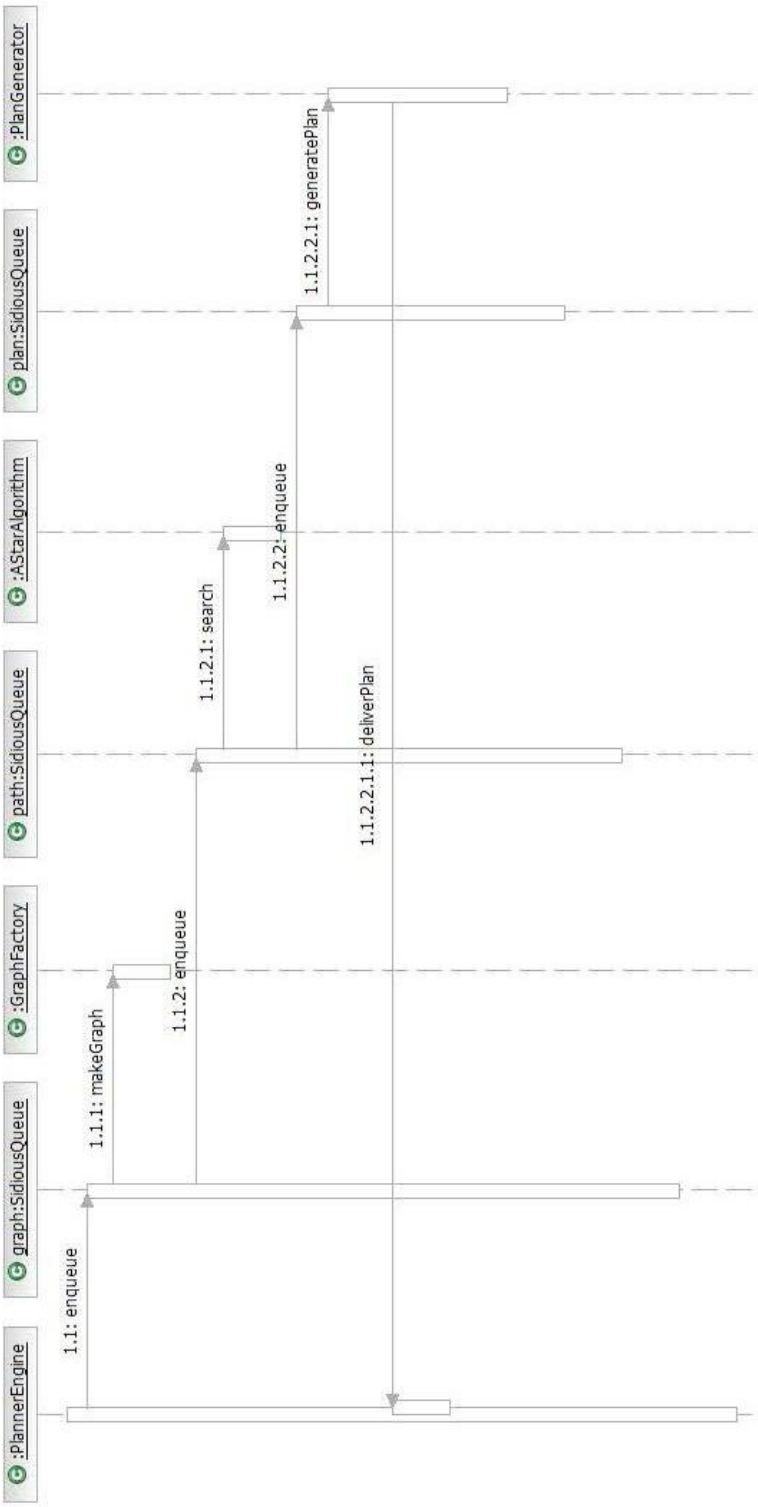

Figure C.1: Sequence diagram of the planning

Appendix D

Co-Author Statement

Thesis: Rule Based Planner Framework – Game AI Technology Applied to Climate Control

Aurthors:	Brian Johnsen (BJ) Morten Lykkegaard Kristiansen (MLK)	

The undersigned hereby state that the above authors contributed to the thesis as stated below:

Intellectual input:	BJ	50%
	MLK	50%
Experimental results:	BJ	50%
	MLK	50%
Writing process:	BJ	50%
	MLK	50%

Brian Johnsen

Morten Lykkegaard Kristiansen

Bibliography

[1] Den Kongelige Veterinær og Landbohøjskole, Danmarks JordbrugsForskning, Dansk Erhvervsgartnerforening, and DGT-volmatic. Intelligrow. http://www.intelligrow.dk, 1999-2004.

[2] Blizzard Entertainment. *Diablo*. http://blizzard.com/diablo/, 1996.

[3] Westwood/Electronic Arts. *Command & Conquer*. http://www.ea.com/official/cc/firstdecade/us/index.jsp, 1995.

[4] G. van Straten, H. Challa, and F. Buwalda. Towards user accepted optimal control of greenhouse climate. *Computers and Electronics in Agriculture*, 26:221–238, 2000.

[5] Microsoft Game Studios. *Halo: Combat Evolved*. http://www.microsoft.com/games/halo/, 2007.

[6] Mat Buckland. *Programming Game AI by Example*. Woodware Publishing Inc., 1 edition, 2005.

[7] Michael Negnevitsky. *Artificial Intelligence - A Guide to Intelligent Systems*. Addison Wesley, 2 edition, 2005.

[8] Electronic Arts. *Earl Weaver Baseball*. http://www.sportplanet.com/features/articles/ewb/, 1987.

[9] IBM Research. *IBM's Deep Blue*. http://www.research.ibm.com/deepblue/, 1997.

[10] Warren S. Sarle. *ai-faq/neural-nets*. ftp://ftp.sas.com/pub/neural/FAQ.html, 2007.

[11] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

[12] Marvin Minsky and Seymour Papert. Perceptron: An introduction to computational geometry. *MIT Press*, 1969.

[13] A.L. Blum and R. L. Rivest. Training a 3-node neural network is np-complete. *Neural Networks, Vol. 5*, 1992.

[14] Penny Sweetser. How to build neural networks for games. *AI Game Programming Wisdom 2*, 2004.

[15] A. J. Champandard. The dark art of nerual networks. *AI Game Programming Wisdom*, 2002.

[16] Lionhead Studios/Electronic Arts. Black & white. http://www.bwgame.com, 2001.

[17] R. Evans. *AI in Games: A Personal View*. http://gameai.com/blackandwhite.html, 2001.

[18] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[19] F. D. Laramée. Genetic algorithms: Evolving the perfect troll. *AI Game Programming Wisdom*, 2002.

[20] Jeff Orkin. Applying goal-oriented action planning to games. *AI Game Programming Wisdom 2*, 2004.

[21] Lotfi A. Zadeh. *BISC - The Berkeley Initiative in Soft Computing*. http://www-bisc.cs.berkeley.edu/, 1997.

[22] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[23] Senmatic DGT-Volmatic. *SuperLink PC Program*. http://www.senmatic.com/volmatic, 2007.

[24] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 1 edition, 2006.

[25] Sun Microsystems. *JSR-000223 Scripting for the Java Platform*. http://jcp.org/aboutJava/communityprocess/pr/jsr223/index.html, 2007.

[26] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality af a*. *J. ACM*, 32(3):505–536, 1985.

[27] Ivar Jacobsen, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1 edition, 1999.

[28] Manifesto for agile software development. http://agilemanifesto.org.

[29] Alistair Cockburn. *Crystal Clear : A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 1 edition, 2004.

[30] Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.

[31] Andrew Hunt and David Thomas. *Pragmatic Unit Testing in Java with JUnit*. Pragmatic Bookshelf, 1 edition, 2003.