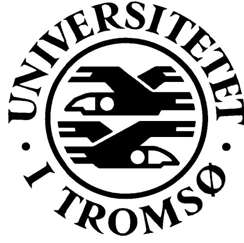


Master thesis in inf-3981

Framework for development of rule based sensor services in the Argos middleware platform

Tom Arild Jakobsen

Jan. 7, 2008

Faculty of science

Department of Computer Science

University of Tromsø, N-9037 Tromsø

ABSTRACT

Argos is a middleware platform developed at the University of Tromsø. It provides tailored, flexible and extensible middleware support. In this thesis we suggest a new approach to creating user services for Argos by using a rule engine to setup the program flow for components in Argos. The users are provided with a graphical tool where they can set up rules that can trigger an action. The input, called a fact, to the rule and the action that is triggered has to be picked from the methods of the components in Argos. These fact/ action-methods are component methods annotated with fact/action annotation which is part of the created rule engine system service for Argos.

The created rule engine system service also provides an API that is available to all programmers that want to use rules in their Argos components. There are many advantages to expressing functionality trough rules opposed to conventional declarative programming. By only telling the program what to do and not how to do it, rules are more easily understood by humans. This can benefit both the experienced programmer and the non-technical partner in a project.

Lifestyle diseases are a growing problem in Western Europe and North-America. An application, realized trough the rule editor tool, for monitoring a user's activity and give feedback will also be presented.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Arne Munch-Ellingsen for all his help. Together we brought this thesis to new heights trough our discussions. Also thanks to Gunnvald Bendix Svendsen (Telenor) for supplying me with specifications for the application where the Rule Editor can be used.

I would also like to thank my family, Hjørdis Solstad, Thorfinn Jakobsen and Mai Linn Jakobsen for all the love, support and food I've gotten while being busy with my thesis. Last I would like to thank my wonderful girlfriend Linda Kayseas for the inspiration in my heart to go and create something beautiful.

CONTENTS

1 Introduction 11
1.1 Background 11
1.2 Problem Description 12
1.3 Specifications and limitations 12
1.4 Method and approach 13
1.4.1 Scrum 13
1.5 Applying the system to a real task 14
1.5.1 The lifestyle project 15
1.5.2 The experiment 15
1.5.3 Using the rule editor to create this service 17
2 Related Work 19
2.1 Combining rule engines with sensors 19
2.2 Lifestyle 21
2.2.1 Major lifestyle problems 21
2.2.2 Problems with traditional intervention programs 22
2.2.3 Mobile persuasion 22
2.2.4 Some lifestyle tools available 24
3 Requirements 27
3.1 Drools system component 27
3.2 Drools user component 30
3.3 Real life application 33
3.4 Non-Functional requirements 35
4 Technology 37
4.1 Argos 37
4.1.1 Component model of Argos 37
4.1.2 Java Management Extensions (JMX) 39
4.1.3 Argos system services 39
4.1.4 Argus 40
4.2 Rules engines 40
4.2.1 Declarative programming 41
4.2.2 Rete algorithm 41

4.3 Drools 43
4.4 JSP 44
4.5 AJAX with Scriptaculous 44
5 Design 47
5.1 !!Drools system service 47
5.1.1 Drools role in Argos 47
5.2 !!DroolsComponent 48
5.2.1 Annotations to specify facts and actions 48
5.2.2 XML file to specify facts and actions 49
5.3 User services in general 50
5.4 Rule Editor user service 50
5.4.1 The graphical user interface (GUI) 51
6 Implementation 53
6.1 Drools system component 53
6.1.1 !!Drools 53
6.1.2 !!DroolsComponent 54
6.2 RuleEditor 55
6.2.1 User component with web-based interface 55
6.2.2 Component Icon 58
7 Evaluation 59
7.1 Functional requirement evaluation 59
7.2 Non-Functional requirement evaluation 61
7.3 Method - Scrum 62
7.4 Technical solutions 63
7.4.1 Rule Engines 63
7.4.2 Graphical User Interface 64
7.5 Remaining work 66
Hibernate 66
8 Conclusion 67
8.1 Achievements 67
8.2 Future work 67
8.3 Conclusion 68
9 Bibliography 71

LIST OF FIGURES

Example 4.1: Contents of a service jar 38
Example 4.2: Rule code 42
Example 4.3: Rete nodes 42
Example 5.1: Program flow in Argos with Drools 47
Example 5.2: Fact annotation 49
Example 5.3: Acton annotation 49
Example 5.4: action_components.xml 50
Example 6.1: Simplified architecture 54

Figure 1.1: The factors of motivation (7) 16
Figure 3.1: Use-case diagram for the system component 27
Figure 3.2: Use-case for user components 30
Figure 3.3: Use-case diagram from user view 33
Figure 3.4: Use-case diagram from Researcher perspective 34
Figure 4.1: Rule engine architecture 40
Figure 5.1: Component diagram of !!Drools and Rule Editor 51
Figure 6.1: Web based GUI 56

Table 1.1 Scrum terminology 14
Table 1.2: Description of elements 16
Table 2.1:Key design points of FACTS 19
Table 2.2: Fact properties in FACTS 20
Table 2.3: Applications for proactive health care 23
Table 2.4: Lifestyle improvement tools (7) 25
Table 3.1: DS-1: Run Drools in Argos 28
Table 3.2: DS-2: Add rules and facts 28
Table 3.3: DS-3: Add rules and facts 28
Table 3.4: DS-4: Facts and action-annotations 29
Table 3.5: DS-5: Add rules and facts 29

Table 3.6: DS-6: Persistence 30
Table 3.7: DU-1: Establish connection 31
Table 3.8: DU-2: Create rules 31
Table 3.9: DU-3: Add statements and actions........ 31
Table 3.10: DU-4: Remove facts, statements and acitons........ 32
Table 3.11: DU-5: Facts and action listings........ 32
Table 3.12: DU-6: Support a basic set of operations 33
Table 3.13: DU-7: Drag-and-drop GUI........ 33
Table 3.14: RL-1: Input data about person into system 34
Table 3.15: RL-2: Monitor user training........ 35
Table 3.16: RL-3: Motivate user 35
Table 3.17: NF-1: Generic 35
Table 3.18: NF-2: Simplicity 36
Table 3.19: NF-3: Use Argos functionality........ 36
Table 4.1: Files and file types found in a component 38
Table 4.2: Relevant system components 39
Table 5.1: Supported annotations........ 49
Table 5.2: Areas of interest in the GUI 52
Table 6.1: Files in the web folder........ 55
Table 6.2: Parameters in dropable.jsp 57
Table 6.3: Description of the parameters 58
Table 7.1: Evaluation of functional requirements for the Drools system service........ 60
Table 7.2:Evaluation of functional requirements for the Drools user service 60
Table 7.3:Evaluation of the Real life usage of the User service 61
Table 7.4:Evaluation of non-functional requirements........ 61
Table 8.1: Future work........ 68

1 INTRODUCTION

1.1 Background

Computer programmers write code to express how a program will work. There are multitudes of languages to choose from that lets us create very complex applications. Professionals in areas other than computer science, in general, do not have detailed insight in how to create computer programs. They do however have deep insight into the business logic of their trade. This is one area where rules and the rule engines can be of help (1). Rules engines simplify applications by separating business policy or rules logic from process, infrastructure, and presentation logic (2). Rules that are expressed in a rule language, such as drl, are simple enough for non-programmers to understand and verify. In the hands of an experienced programmer, expressing the business logic as rules as opposed to traditional code, can reduce the time used developing, deploying, modifying and managing a system.

Argos is a personal middleware system with focus on tailored, flexible and extendable solutions. The Argos platform is a lightweight microkernel implemented in Java. It offers a service and component model, component lifecycle management and hot deployment of new service into the system. The services are categorized as system services and user services, where a system service extends the Argos core and a user service can use one or more system services to create user applications. The Argos component model combined with a rule engine gives a very natural dataflow trough a service. First a component in Argos takes in some sensor data that is pushed into the rule engine as a fact. The rule engine will then do an evaluation of the rules to see if the new facts trigger an action. If an action is triggered the rule engine will trough Argos call a method in another component that gives the user some output. By combining the simplicity and expressiveness of rules with the easy access of input and output sources in Argos, component programmers can more effectively create new functionality. This is particularly true if the programmer is working with non-technical partners, who will now more easily gain a deeper understanding of the inner workings of the component, since rules are much easier to understand than regular program code. (3)

1.2 Problem Description

The main goal of this work is to develop a system service for Argos that combines the rule engine with the existing functionality to get data from sensors to an integrated system. The goal is that it should be simpler to create services that combine sensors and rules in a natural program flow. There will also be developed user service in Argos that offers access to the to the Rule engine trough a simplified GUI.

1.3 Specifications and limitations

Due to time limitations, the software will be designed and developed as a prototype. One can divide the project into three parts. The system component offers Drools functionality to other components. The user component is the Rule Editor tool that can be used to create a generic rule service. The third part of the project is setting up the Rule Editor so it can be used by Researchers at Telenor R&I for a specific project. The system component is the central part in integrating Drools into Argos, but it will to a large extend be an overlay over existing functionality in Drools, so it will only require a smaller piece of the combined workload for the project. The Rule Editor will be the most work intensive part of the project. Setting up the system for use by Telenor R&I will not be a priority unless time allows for it. The Rule Editor will be created in such a way that it should be easy to create a vide range of new services based on rules. The difficulty in setting up a system for Telenor is that it relies on there being other components in the system to provide input and output (facts and actions). Creating such other components will not be considered a priority.

Neither the system-component nor the user-component will support the entire functionality of Drools. They will provide a basic set of operations, but the implementation will be generic enough to extend the functionality at a later point if the need for more complex services arises.

1.4 Method and approach

1.4.1 Scrum

The chosen method for managing the programming process in this thesis is Scrum. Scrum is described more closely in *Agile Software Development with Scrum* (4). Scrum was first described by Takeuchi and Nonaka in (5) where they pointed out that projects using small, cross-functional teams historically produce the best results. The name Scrum refers to a strategy in rugby for getting an out-of-play ball back into play. In the 1990's several different projects helped develop Scrum further. Ken Schwaber used it in his company *Advanced Development Methods*, and Jeff Sutherland, John Scumniotales, and Jeff McKenna developed a similar approach at *Easel Corporation*. Schwaber and Sutherland collaborated during the following years to merge their experiences, and industry best practices into what is now known as Scrum. Scrum is a management and control process that cuts trough complexity to focus on building software that meets business needs. Scrum is an empirical[1] approach that improves flexibility, adaptability and productivity in a project (4) (6).

Concept	*Description*
Team	In scrum terminology a team is a self-organizing autonomous group working together. In the case of this thesis there was only one person in the team.
Task	A task is one isolated piece of work that has to be done. A task should not be longer then 8-10 hours.
Product backlog	This is an evolving prioritized queue of business and technical functionality that need to be developed into a system.
Sprint backlog	The sprint backlog contains the items taken from the product backlog that has to be done within this sprint.
Sprint	A sprint is a given time in which the tasks of the sprint backlog are to be done. The sprint time used in this thesis was two weeks.
Scrum master	The scrum master is the person responsible of a project. In a company he will represent the management and the team to each

[1] based on observation or experience

	other. His job is to make sure that nothing is keeping the developer from doing his job. In this thesis my supervisor has taken the role as scrum master.
Daily scrum	The daily scrum is a short meeting between the scrum master and team. It is used to shed light on things that might be keeping the team from doing their jobs. The daily scrum has in this project been held at least once a week (so it hasn't been daily).

Table 1.1 Scrum terminology

To help managing Scrum a program called S*crumworks*[2] has been used to setup the tasks in the backlogs and sprints. It is also used by the scrum master to monitor the progress of the team. The whole project started with thinking of as many as possible tasks that had to be done in the duration of the thesis period. These tasks where added to the "product backlog". Items were then picked from the product backlog into the sprint. And there you go, the tasks for the first sprint are ready, and the sprint is ready to begin. As time goes by more tasks that must be done will show up. The important thing here is that these tasks go into the product backlog and not into the ongoing sprint. To give predictability and continuity trough the sprint it must be kept the same, except for when all the tasks in a sprint is done, then you can of course go and pick more items from the top of the product backlog. It is generally the task of the scrum master to keep the product backlog sorted by importance. The team can then go and pick from the top to get the most important task to be done first. Architecture and design emerges in Scrum across multiple sprints, rather then being developed completely during the first sprints. At the end of a sprint the team and scrum master (and management if any) comes together to review the progress of the sprint. The backlog is then revised and sorted by importance before picking tasks for the next sprint backlog.

1.5 Applying the system to a real task

The university collaborates with Telenor R&I and this thesis will be used as part of a research project at Telenor R&I called "Lifestyle change by divine intervention". After a series of talks with the psychologist at Telenor we have together found an application that the Rule Editor

[2] http://www.danube.com/scrumworks

service is well suited for. The goal is that a researcher should be able to use the Rule Editor to describe a service in Argos without having to bother with any programming, only using a simple tool to set up the connections between the different components running in Argos.

1.5.1 The lifestyle project

The main assumption behind the “Lifestyle change by divine intervention” project is that users receive information trough mobile phones and that this will help them in changing their lifestyle. Mobile phones are supposed to be more effective than PC in this regard, because they are with the user all the time and thus are able to target the behaviour in question at appropriate times during the day. A further assumption is that automatic gathering of information, e.g. via step counters or other activity measures, will be fruitful in helping users be more active. The assumption made is that it’s not easy to keep track of and have a conceptual grip on ones own physical activity over a period of time. So by giving the user a tool to measure actively and summarize the activity, it can be used as a goal for change. Measurements like this will help the user change his behaviour by giving him a traceable handle to his own behaviour (7).

1.5.2 The experiment

A simplified explanation of the flow of the experiment will be described in this sub-chapter. A user will start by answering a standardized questionnaire containing 20 questions. The answers from this give tell something about the importance of each of the five elements in Figure 1.1. Later when the user takes the questionnaire again the results can be compared to the initial one to see where the motivation is slipping and what feedback is necessary to get the user back on track again. Which feedback the user will get is connected to what is happening with the five elements the user has chosen from a list in advance, or that the user himself has written the feedback message to make it more personalized (7).

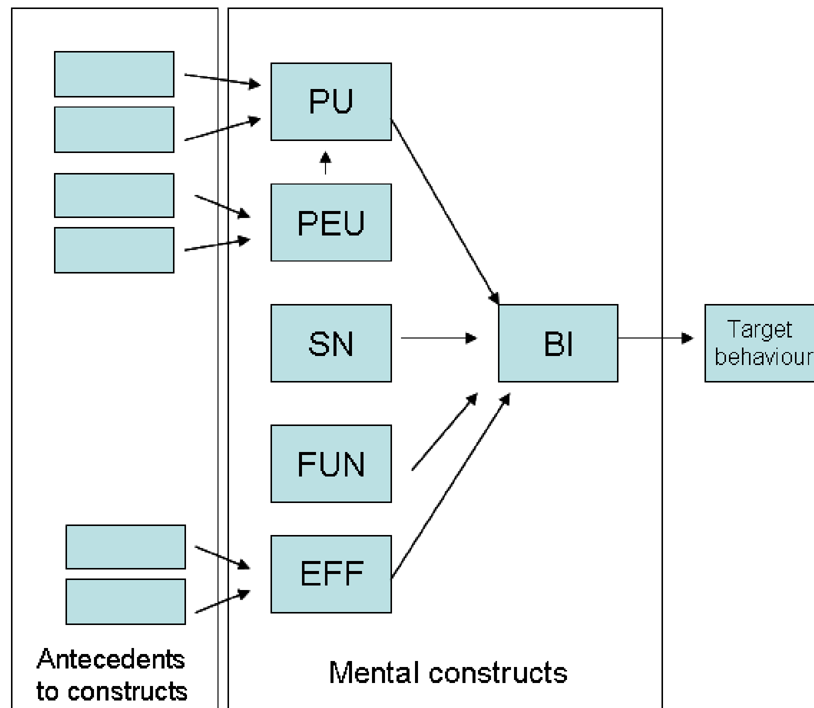


Figure 1.1: The factors of motivation (7)

Element	*Description*
BI	The users stated behavioral intention to act according to a set of goals.
PU	The user's opinion of the usefulness for the user to act in this way.
PEU	The user's opinion of how easy it is for her or him to act in this way.
SN	The user's opinion of what persons important to her or him think (s)he ought to act.
FUN	The user's opinion of how fun it is to act in this way.
EFF	The users opinion of his or hers ability to change the target behaviour

Table 1.2: Description of elements

There must be some kind of measure for the behaviour of the user. This measure is time stamped and might be aggregated over different time periods. If the target behaviour is physical activity, the behaviour will both be measured by user reports and a sensor (e.g. strep counter). The user has a set of goals that he wants to reach. This goal is connected to the measurement done in some way. So that if a step counter is used the goal has to be that the user should walk x number of steps in a given time. To keep track of the goals the user has a daily schedule. SMS on a mobile phone will be used to give the user the questionnaire to

answer, and also to give feedbacks to the user reminding him of his ultimate goals if the goals are not reached. Computers can also be used to give the user the questionnaire form (7).

1.5.3 Using the rule editor to create this service

The question is then: can the simplest possible generic rule editor help create such a service as the on outlined above? Well, hopefully throughout this thesis we will get a little closer to answering that question. By having a specific service in mind when making the rule editor one can discover things that the system will need to be useful in the creation of real services.

2 RELATED WORK

2.1 Combining rule engines with sensors

There are other systems than ours out there that are using rule-based middleware architecture with sensors. One such system is FACTS (8) where they have used rule-engine middleware in a sensor network. Since it was built to be run on nodes in a sensor network the whole premises of the system is different then that of the one in this thesis, but the motivation behind is very much the same. Both projects are using a rule engine to react to sensor input.

The developers of FACTS have also created a new rule-based language tailored to their needs. Their objective is to combine event-centric processing with rule-based execution while also preserving low resource usage. The main criteria for the design of the FACTS system can be listed with these key points:

Design feature	*Motivation*
Event-centric architecture	Sensor nodes detect changes in the environment and react to them. With no changes it should stay in low-power mode. This is a triggered action, thus a push mechanism is favorable.
Rule-based language	Captures trigger/action semantics and allowing local decision to be made.
Minimalistic architecture	The architecture should be minimalistic. To extend functionality the system can be enhanced at runtime according to application specific services.
Distributed shared memory	The fact repository supports software relying on any kind of grouping or clustering mechanism.
Cross-layer optimization	Algorithms operating on different layers according to the ISO/OSI layering model[3] can also exchange information to use it as a means to coordinate and tune themselves.

Table 2.1:Key design points of FACTS

[3] The Open Systems Interconnection Basic Reference Model (OSI Model) is a layered, abstract description for communications and computer network protocol design. It has seven layers: Physical, Data link, Network, Transport, Session, Presentation and Application.

FACTS use rules to express algorithms and reactions to external events in the system. A rule is a named structure containing both a set of conditions and an ordered list of statements[4]. A statement modifies the fact repository or generally interacts with the rest of the system. A rule will fire a statement/list of statements when all conditions are evaluated to true and at least one of the facts used in the conditions are tagged as modified. The modify flag is set whenever one property of a fact is changed, either caused by external events or the execution of statements of a previous run of the rule engine. Facts are the central means of representing any kind of data in the system. Facts are not only used inside the rule engine, but are also the central means of transmitting information between nodes of the sensor network. A fact has a set of read-only properties that are set by the system. These properties are:

Property	*Type*	*Description*
owner	int	The network wide unique ID of the sensor node that was the last to modify this fact.
time	int	The time at when the fact was last modified.
id	String	The network wide unique ID of this fact. It is implemented as the dot-separated concatenation of the IF of the owning sensor node and the time of last modification (which is unique to the local node).
modified	Boolean	This Boolean flag indicates whether a fact has been modified since the last run of the rule engine.

Table 2.2: Fact properties in FACTS

Unlike the Drools rule engine used in the thesis, FACTS does not support local variables to which a specific fact cam be bound within a rule. This would be too expensive in terms of memory usage. FACTS use its own system with something they call slots-filtering to provide the same functionality. (8)

[4] Note that there is a difference in the terminology used in FACTS and the rest of this thesis. A statement in FACTS is what we call an action. And what FACTS call a condition is what we call a statement.

2.2 Lifestyle

There has been done some research about the use of technology to change people's lifestyle. In (9) Telenor and Tromsø Telemedicine Laboratory[5] (TTL), thay have looked into how technology can help improve a person's lifestyle. This is very interesting for the part of this thesis where we look at the application area of the Rule Editor (see chapter 5.4 for more details).

2.2.1 Major lifestyle problems

Lifestyle diseases are a growing problem in Western Europe and North-America. They include:

- Alzheimer's disease
- Atherosclerosis
- Cancer
- Chronic liver disease or cirrhosis
- Chronic Obstructive Pulmonary Disease
- Type 2 diabetes
- Heart disease
- Nephritis or Chronic renal failure
- Osteoporosis
- Acne
- Stroke
- Depression
- Obesity

Factors concerning diet and lifestyle are thought to influence susceptibility to the diseases listed above. The focus in this chapter will be lifestyle diseases based on overweight and obesity. Studies indicate that more then 50% of adults are defined as either being overweight or obese in ten western OECD countries. The economic costs of obesity have been assessed in several developed countries to be in the range 2 – 7% of the total health costs. These are conservative estimates but it clearly shows that obesity represents one of the largest items of expenditure in national health care budgets. There is also studies telling us that overweight among children and adolescents are becoming more and more common. One of the reasons for the drastic increase on obesity, both among children and adults is that people live a more

[5] http://www.telemed.no/ttl/

inactive lifestyle today then we did before and that the time pressure makes people choose fast and inexpensive food. This food often contains more fat and carbohydrates then is recommended on a daily basis (9).

2.2.2 Problems with traditional intervention programs

A health intervention is an effort to promote good health behaviour such as physical exercise or to prevent bad health behaviour, e.g. promoting healthier eating and be more physical active. Traditional health intervention programs have traditionally focused on behaviour change in specific populations such as a work place or school. This means that it doesn't reach out to everyone, but only a narrow part of the population. Critiques have pointed out that the long term effects are missing because people can't adhere to the programs. So the billions of dollars put into such programs give very little result. One of the flaws behind many health intervention programs is that they have forgotten that there are so many factors that control a persons behaviour. Desired behaviours must be modelled, rehearsed and reinforced. Tailoring has been the answer to much of this criticism. Tailoring is basically to individually assess a person to find out which type of information and change strategy will give the wanted outcome for that individual. (9).

2.2.3 Mobile persuasion

After the introduction of the Internet, early in the 1990's, technology, and in particular computers has started to be used as a device to change health behaviour. More recently one has started to try out mobile phones in the same area. The advantage with mobile phones is that they are more accessible to most people and easy to use. (10) The effect of tailoring is often bigger because it's easier to reach people trough their phones then trough their computer. Research has been done in trying to use your mobile phone as your lifestyle coach by help of sensors, Internet, and various applications. It is also clamed that the use of mobile phones will increase adherence even more by tailored messages to users. (9)

The timing of messages to the user is also of great importance. The system can use sensors and user input to determine when to present messages to motivate healthy behaviour. The medical systems of many other countries face an impending crisis in a few years: how to pay

for an aging population. To ease the burden on the health care system it is necessary to try to help people stay healthy and living independently of the medical system as long as possible. Applications for proactive health care can be organised as in Table 2.3 (11).

System	*Description*
System that detect crisis	Detecting crisis require good sensors, like biometric sensors on the body, that can get help if a critical situation comes up.
System that detects declines in health	Detecting a gradual change in health status will generally require multiple, multi-modal sensors. A sensor system that can detect changes in everyday activity in the home would enable a new generation of home-based and institutionally based services for the aging. Changes in everyday activity can often precede decline in health.
Systems that motivate healthy behaviour	If a computer can identify everyday activity, then it can not only monitor changes, but it can also proactively present information that may lead to behaviour changes that help people stay healthy.

Table 2.3: Applications for proactive health care

Let's look a bit closer on the third kind of systems: systems that motivate healthy behaviour. Researchers in a number of fields have demonstrated the power of point-of-decision messaging (e.g. improving safety at the workplace, encouraging seat belt use, increasing public recycling, reducing electricity consumption and encouraging exercise in public spaces). Studies have show that context-sensitive information presentation can make a difference. There are four components to effectively motivate behaviour change:

1. Present a simple message that is easy to understand
2. At just the right time
3. At just the right place
4. In a non-annoying way

To present the messages at just the right time requires computational sensing that can infer context from sensor data. It also requires that the user have some device that gives the output where the user is, like a mobile phone or a PDA. To present the information in a non-annoying way means that the information must be relevant given the context and the

presentation of information must not disrupt ongoing activity. It may be better to present more subtle information that the user is receptive to. Rather then attempting to command the user what to do, the system can give positive reinforcement. If the user took a unusually long walk the system can inform the user of the health benefits. This is perceived as less annoying then having a system that tells the user that he/she hasn't taken a walk in a long time. (11)

2.2.4 Some lifestyle tools available

You can see some of the lifestyle tools on the marked listed up in Table 2.4.

Lifestyle tool	*Description*
BeWell Mobile	*BeWell* Mobile develops and sells wireless mobile technology to improve patient's health. They focus on diabetes, smoking cessation and asthma. The company is doing research on physical activity measurement.
Qualcomm	*Qualcomm* produce wireless devices which can be used for personal entertainment, productivity and lifestyle. When the mobile phone is equipped with committed sensors it can monitor both biological and physiological information about a user. The phone can also remind a person of his activities trough the day.
myFoodPhone	*myFoodPhone Nutrition, Inc.* is a mobile health application service provider. Its flagship product is myFoodPhone, is a camera-phone food-journaling feedback service. When the user eats something they must first take a picture of it and send it in, so that their food intake can be monitored, and feedback given on the bases of that.
Card Guard	*Card Guard AG* is developing health care delivery platforms for consumers, high-risk and cronically ill patients, and to home health, disease management, eHealth and wellness/fitness markets. Their flagship is the HealthePod that measures 1-Lead ECG, heart rate, body fat, body temperature and blood glucose levels. Each parameter that is measured can be uploaded to a wireless handheld devise for display and to be transmitted to a web-based server for analysis by professionals.

HealthPia	*HealthPia is a prototype system of a diabetes phone. It is the world's first all-in-one glucometer cell phone and service for managing diabetes remotely. It provides patients and professionals with 24/7 support and emergency intervention trough a GlucoPhone subscription.*

Table 2.4: Lifestyle improvement tools (9)

3 REQUIREMENTS

In this chapter we will look into the requirements for our “framework for development of rule based sensor services”. One of the preconditions of the software that will be created is that it will run in Argos. Another precondition is that the application is to be split into two parts. One generic part where the rule engine is hosted and can be used by other components needing rule engine support, and one part that uses that functionality to give the user a way to design rules for Argos trough a GUI. The first part will be a system component, since it only renders functionality to other components while the second part will be a user component since it actually gives functionality to the user. We will now take a look at the requirements for the system components, then we will look at the user component, a real life scenario for the user component and last we will look at non-functional requirements.

3.1 Drools system component

The Drools system component is meant to give other components easily access to a rule engine. It will offer an interface to add facts as well as adding and removing rules in the working memory of the rule engine. In Figure 3.1 you can see a use-case diagram from a user components view as it uses the system component. You can also see functionality in the system component that Argos needs.

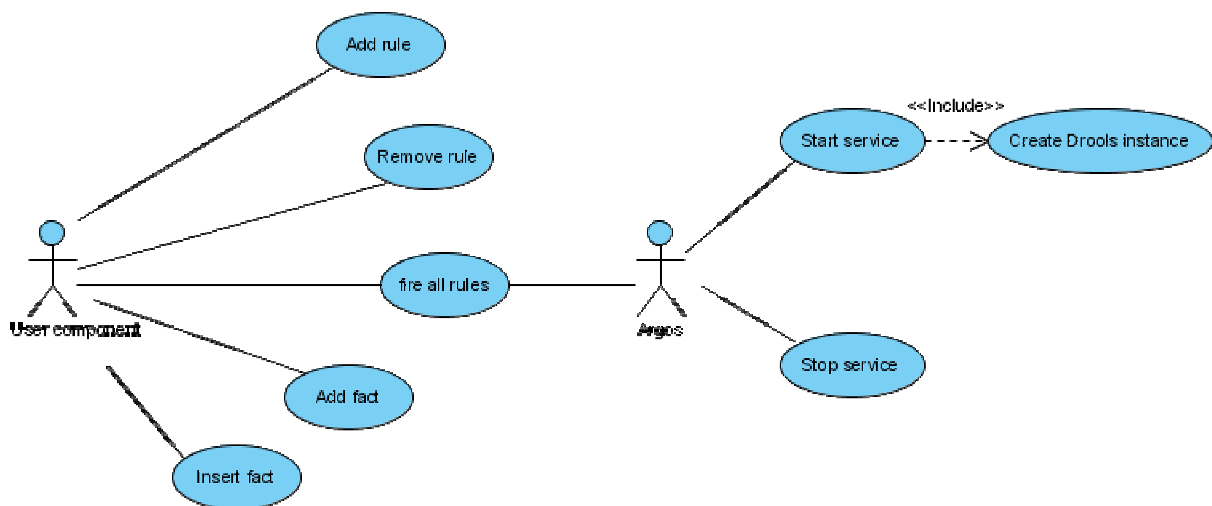


Figure 3.1: Use-case diagram for the system component

Requirement Id	*DS-1*
Requirement name	Run Drools in Argos
Priority	High
Goal	Have instance of a drools rule engine running in Argos
Testing	*See requirement D-2*
Description	Set up the core system component for Argos. Set up a rule engine with one working memory that rules can be run in.

Table 3.1: DS-1: Run Drools in Argos

Requirement Id	*DS-2*
Requirement name	Add rules and facts
Priority	High
Goal	Insert a rule base and facts into Drools in Argos
Testing	1. Run insert simple test rule base 2. Insert fact 3. Get correct response
Description	We need to be able to add a rule base and facts into Drools running under Argos.

Table 3.2: DS-2: Add rules and facts

Requirement Id	*DS-3*
Requirement name	Remove rules
Priority	Medium
Goal	Remove rules from the rule engine
Testing	1. Delete a rule 2. Check if it still exists
Description	Sometimes there is a need to modify the rule base as we go along; this makes the user component able to remove a unwanted rule.

Table 3.3: DS-3: Add rules and facts

Requirement Id	*DS-4*
Requirement name	Facts and action-annotations
Priority	Medium
Goal	Use annotations to show which components can be used as facts and actions in Drools applications.
Testing	1. Annotate demo component 2. Check that the demo component is in fact registered
Description	Make it possible to annotate components that can be used as facts or actions in Drools context.

Table 3.4: DS-4: Facts and action-annotations

Requirement Id	*DS-5*
Requirement name	Specify facts and action components
Priority	Medium
Goal	Setup an already existing component in Argos to be able to be a fact or action in Drools.
Testing	1. Create configuration file specifying a component to be a fact or action 2. Check if the component was in fact registered as such
Description	Use a configuration file to specify already existing components in Argos to be facts or actions in Drools

Table 3.5: DS-5: Add rules and facts

Requirement Id	*DS-6*
Requirement name	Persistence
Priority	Low
Goal	Save working memory persistent
Testing	1. Create rules in the component 2. Restart Argos 3. See if the rules still exist

Description	*There is a need to save the working memory between restarts of Argos so that users will not loose the rules that has been made in a session.*

Table 3.6: DS-6: Persistence

3.2 Drools user component

The user component that will be implemented to use the system component is a rule editor. This rule editor will enable a user to set up simple rules that uses the methods found in other components. In Figure 3.2 we can see the use case diagram of the rule editor. It is viewed from a user's perspective, and shows the functionality that the user needs to set up simple rules.

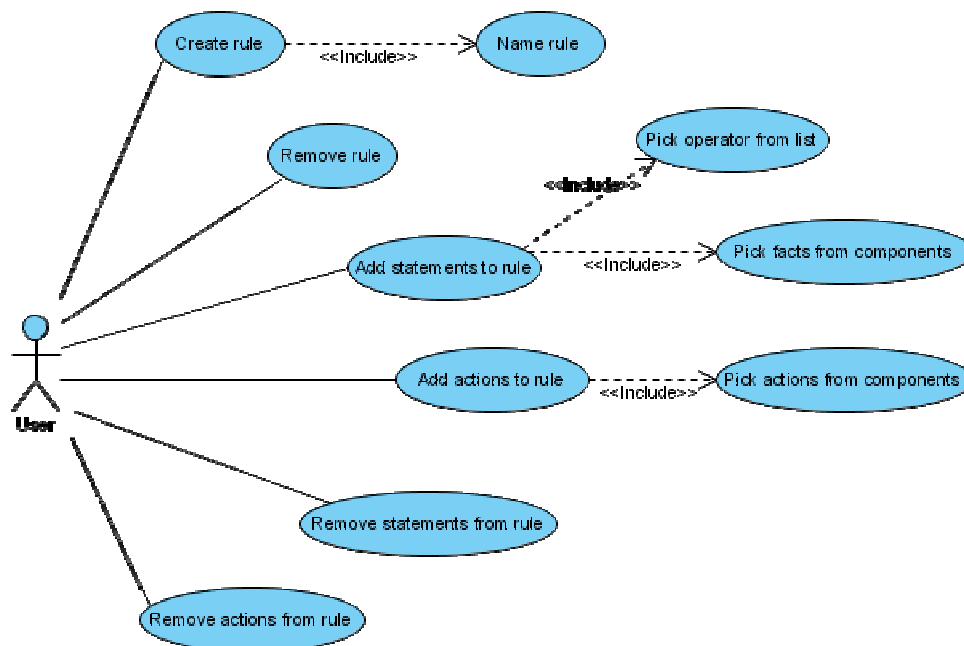


Figure 3.2: Use-case for user components

Requirement Id	*DU-1*
Requirement name	Establish connection
Priority	High
Goal	User component accesses system component for Drools functionality.

Testing	1. Have user component connect to system component 2. User component creates rule and fact 3. Get correct response
Description	The user component should connect to the system component using JMX to get access to the Drools functionality trough that.

Table 3.7: DU-1: Establish connection

Requirement Id	*DU-2*
Requirement name	Create rules
Priority	High
Goal	The user must be able to create a new rule
Testing	1. Create new rule 2. Transfer rule to system component 3. Put fact into the system component 4. Get correct response
Description	The user should be able to create a new rule with a unique name, one or more statements that has to be true before one or more actions will trigger.

Table 3.8: DU-2: Create rules

Requirement Id	*DU-3*
Requirement name	Add statements[6] and actions
Priority	High
Goal	Can add statements and actions to a rule.
Testing	1. Add a statement or an action 2. Check with the rule base in the system component if it was successful
Description	It is possible to add more statements and actions to a single rule.

Table 3.9: DU-3: Add statements and actions

[6] With a statement we here mean a Boolean expression that is part of a rule. A statement consists of two facts and an operation. There can several statements in a rule.

Requirement Id	*DU-4*
Requirement name	Remove facts, statements and actions
Priority	High
Goal	The user can remove facts, statements and actions.
Testing	1. Remove a fact, statement and action 2. Check with the rule base in the system component if it was successful
Description	The user should be able to remove facts that he doesn't need. He should also be able to remove statements and actions in facts that are unneeded.

Table 3.10: DU-4: Remove facts, statements and acitons

Requirement Id	*DU-5*
Requirement name	Facts and action listings
Priority	High
Goal	Get lists of the suitable components that are suitable as actions and facts.
Testing	3. Query Drools system component for lists of facts and actions 4. Display the list
Description	We need to get lists of which components are specified as facts and actions in the Drools component. These will be listed up so that the user can pick which to use with the drag-and-drop GUI.

Table 3.11: DU-5: Facts and action listings

Requirement Id	*DU-6*
Requirement name	Support a basic set of operations
Priority	High
Goal	The rules should be able to include operations such as *equals*, *greater then*, *smaller then*, *not*.

Description	*To be able to create rules the system must support at least a basic set of operations to enable the user to create most rules.*

Table 3.12: DU-6: Support a basic set of operations

Requirement Id	*DU-7*
Requirement name	Drag-and-drop GUI
Priority	High
Goal	The user is be able to build rules by drag and drop
Testing	1. Drag and drop a rule 2. Execute it 3. Get correct response
Description	The user of the component should be able to create new rules in the Argos system by using the drag-and-drop interface.

Table 3.13: DU-7: Drag-and-drop GUI

3.3 Real life application

We will now take a look at one application area of the rule editor. The following specifications are to be used by the Telenor R&I project “Lifestyle change by divine intervention (7)”. The goal here is to see if the rule editor is powerful enough to be used in a real scenario. In Figure 3.3 we can see a use case diagram showing the application from the user’s perspective.

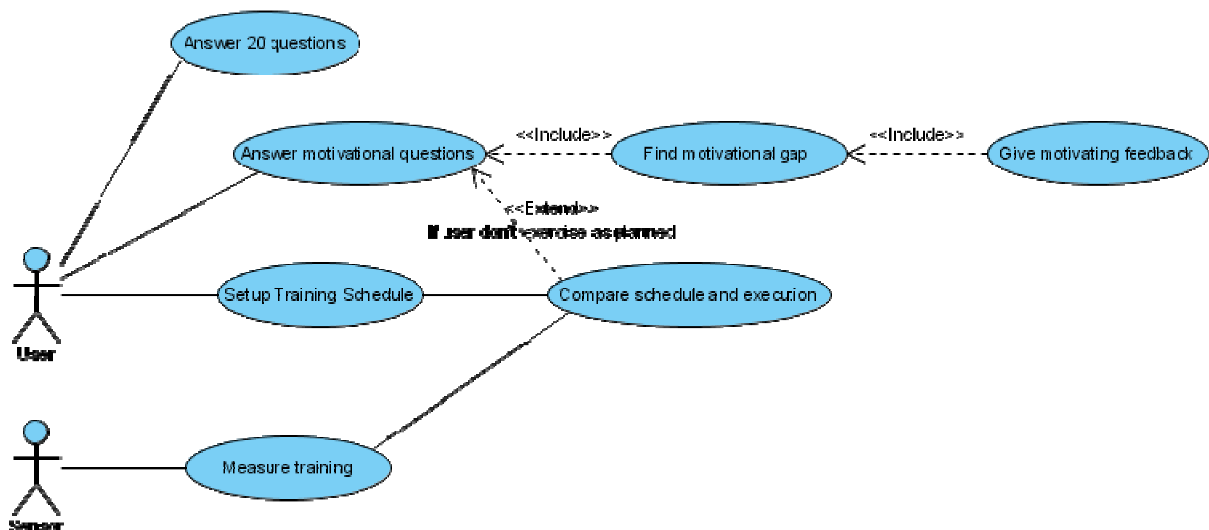


Figure 3.3: Use-case diagram from user view

In Figure 3.4 you can see a use-case diagram for the same application but this time from the researches perspective. This functionality can be viewed as an overlay over the *Rule Editor* component. In addition there is need for some input (facts) and out (actions) that is gained trough other components. This functionality must either be programmed as a new component or found in an already existing component and specified in the way described in requirement DS-5.

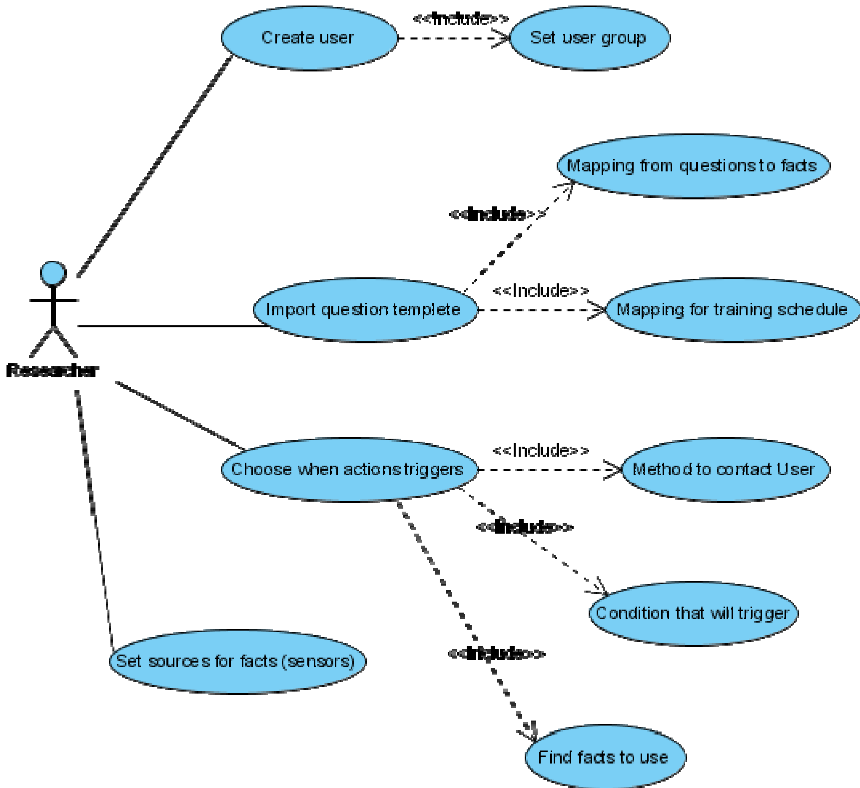


Figure 3.4: Use-case diagram from Researcher perspective

Requirement Id	*RL-1*
Requirement name	Input data about person into system
Priority	Low
Goal	Get user data with his group and personal preferences (20 questions) specified.
Description	Need to get the personal data of a person into the rule engine to be compared against rules.

Table 3.14: RL-1: Input data about person into system

Requirement Id	*RL-2*
Requirement name	Monitor users training
Priority	Low
Goal	Receive sensor data about the users training.
Description	Use a sensor (e.g. step counter) to monitor if the user does his training and put result into rule engine.

Table 3.15: RL-2: Monitor user training

Requirement Id	*RL-3*
Requirement name	Motivate user
Priority	Low
Goal	Motivates user based on his personal specifications
Description	Use the person's specifications and send motivational text-message based on whether he has done his training or not this day.

Table 3.16: RL-3: Motivate user

3.4 Non-Functional requirements

The non-functional requirements will here describe the other non-functional aspects of the system. These requirements are valid for all the parts of the system.

Requirement Id	*NF-1*
Requirement name	Generic
Priority	Medium
Goal	Don't limit the usage of components to one scenario, but make them generic
Description	Both the system and user service should be as generic as possible. The system component should be able to be used by a wide range of components, while the user component should be able to make many very different scenarios.

Table 3.17: NF-1: Generic

Requirement Id	*NF-2*
Requirement name	Simplicity
Priority	Medium
Goal	Save time by choosing the technology that is easiest to use
Description	When it comes to technology choices the simplest solution of several suiting choices should be picked.

Table 3.18: NF-2: Simplicity

Requirement Id	*NF-3*
Requirement name	Use existing Argos functionality
Priority	Medium
Goal	Save time by using already existing functionality in Argos
Description	If there is several choices of technology to use, and one is already supported by Argos, don't spend time on adding new functionality when the existing can be used.

Table 3.19: NF-3: Use Argos functionality

4 TECHNOLOGY

In this section we will describe the different technologies used in this thesis. We will first take a look at Argos which is the framework the applications are running in. Then we will take a look at rule engines in general and the Drools rule engine in particular. And last there will be a little bit about JSP and AJAX with Scriptaculous, which is used for the graphical user interfaces.

4.1 Argos

The Argos platform is a personal middleware platform implemented in Java and developed at the University of Tromsø. The difference between Argos and enterprise systems such as JBoss or Enterprise Java Beans (EJB) is that it was created to host smaller and more personal services. This lets it disregard the strict requirements of scalability found in enterprise systems. This results in more freedom for developers when creating components, e.g. creating threads, reading files and loading native libraries. Argos provides tailored, flexible and extensible middleware support using reflection, dependency injection, Java Management Extensions (JMX) notifications and hot deployment (3).

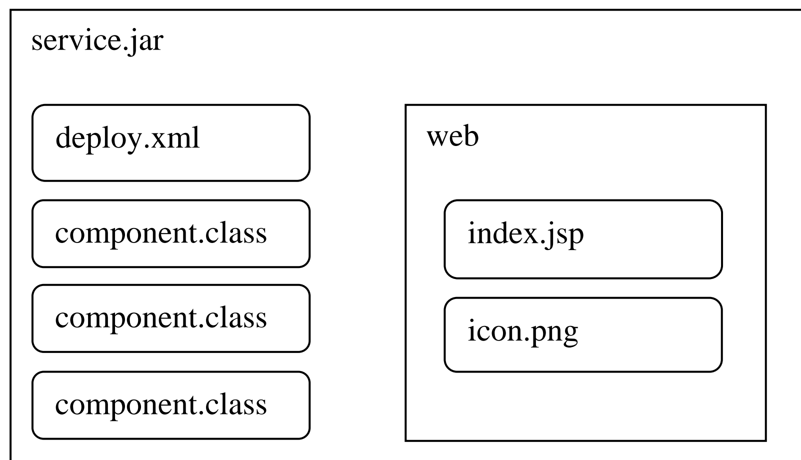
4.1.1 Component model of Argos

Argos is often referred to as a container. This is because it is the “box” that we put services and components into. A component is a POJO (Plain Old Java Object) where the application programmer can use annotations on operations and attributes to change how a component will behave. It is up to the Argos container to extract annotations from components and handle the components as specified by the annotations. A service may consist of zero or more components, a deployment descriptor, external applications, desktop widget and dashboards (see Table 4.1 for common file types found in an Argos component) (3).

Name	*File type*
Components	Class files
Deployment descriptor	XML file
Web content (in the "web" folder)	JSP files,
Graphical representation of the component	PNG file (in web-folder)

Table 4.1: Files and file types found in a component

System components may also extend this component model with more functionality. An example of this is how the *!!Jetty6* system component deploys the contents of the "web" directory in a service in a Jetty6 web container. The files included in a service are packed in a Jar file called the service file. To deploy a service the Jar file has to be copied to Argos' "deploy" folder. Argos will discover and deploy it automatically. See Example 4.1 for the layout of a service file.



Example 4.1: Contents of a service jar

As already mentioned Argos has looser restrictions then those found in enterprise systems (such as EJB). For instance the Argos component model allows components to create threads, open sockets and read from files. This is a real benefit when collecting sensor data. It can simplify the application programmers' job when he is able to create threads, open sockets and read from and write to local files. (3).

4.1.2 Java Management Extensions (JMX)

Argos is built around Java Management Extensions (JMX) which is the Java standard for management of application resources (12). JMX helps with monitoring and management of the components. Communication to and between components in Argos is also done using JMX. There is a notification model that lets the Argos components broadcast and receive notifications internally in the container or even between containers (3). A managed bean (MBean) is an application or system resource that has been instrumented to be manageable trough JMX. In Argos all the components become MBeans. All MBeans follow a set of specifications, but this is not something the Argos component programmer has to worry about. Argos will make the component an MBean by interpreting the deployment descriptor and the annotations in the component files. (3) (12)

4.1.3 Argos system services

Argos also comes with a number of ready to use functionality called system services. These services are notated with a *!!* (bangbang) in the front of the name. Table 4.2 shows the most relevant system components.

Name	*Functionality*
!!jetty6	Jetty 6 is a web container that has support for normal HTML pages as well as JSP and servlets. It will automatically deploy the contents of the "web" directory in a component to the Uniform Resource Locator (URL) http://localhost:8080/ServiceName (3).
!!jmxconnector	Provides functionality to use JMX.
!!hibernate	Hibernate is an object/relational mapping (ORM) tool for Java environments. ORM is a technique of mapping a data representation from an object model to a relational data model with a SQL-based schema (13).

Table 4.2: Relevant system components

4.1.4 Argus

Argus[7] is a graphical management tool for JMX implemented using Java Swing. It can be used to view the components inside the Argos container, and the also to call the attributes and operations of those components. Argus is generic enough to work with all JMX enabled systems. Argus can be a very useful tool when debugging.

4.2 Rules engines

The three most important aspects of a rule engine are facts, pattern and the rules themselves. The rules and facts are taken into the rules engine and if the rules are proved true by the facts, a list of actions will be triggered.

A rule engine gives a programmer a way to separate the decision-making logic from the rest of the program. One can think of it as hiding the imperative programming (telling it how), and focus more on declarative programming (telling it what). The most prominent reason for using a rules engine is that it provides the programmer with a very quick and simple and cost efficient way to change the logic of a program, without having to dive deeply into the code. So instead of using long chains of `if...else` statements, using a rule engine lets you express it by declarative programming which is more structured, intelligible and flexible (14).

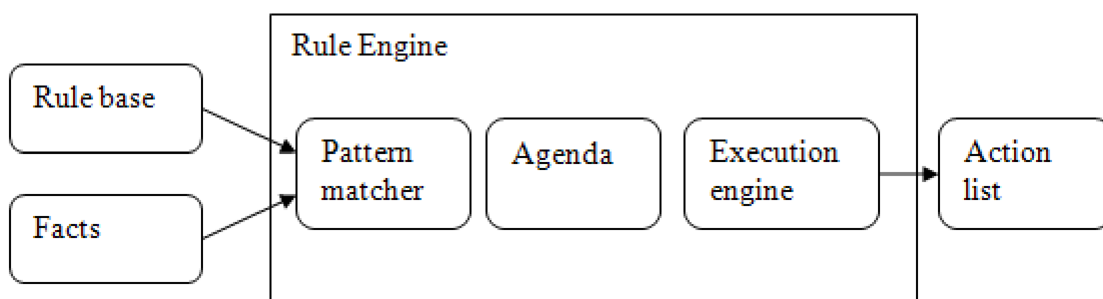


Figure 4.1: Rule engine architecture

[7] http://sourceforge.net/projects/argusjmx

4.2.1 Declarative programming

In *procedural* programming, the programmer tells the computer what to do, how to do it, and in what order. Procedural programming is well suited for problems in which the inputs are well specified and for which a known set of steps can be carried out to solve the problem. Mathematical computations, for example, are best written procedurally. In a purely *declarative* program the programmer tells the program what to do, and omits much of the instructions on how to do it. This means that declarative programs must be executed in some kind of runtime system that knows how to fill in the blanks and use the declarative information to solve the problems.

Since declarative programs only include important details of the solution they can more easily be understood by humans then their procedural counterparts. Declarative programs are particularly good at solving problems without clear algorithmic solutions like control, diagnosis, prediction, classification, pattern recognition, or situational awareness (15).For most simple Argos services procedural programming can be quite enough. But when met with a service that takes input from a number of different sources, and where the relationship between these sources are very complex or difficult to solve with an algorithm, then declarative programming can save you a lot of unnecessary work.

4.2.2 Rete algorithm

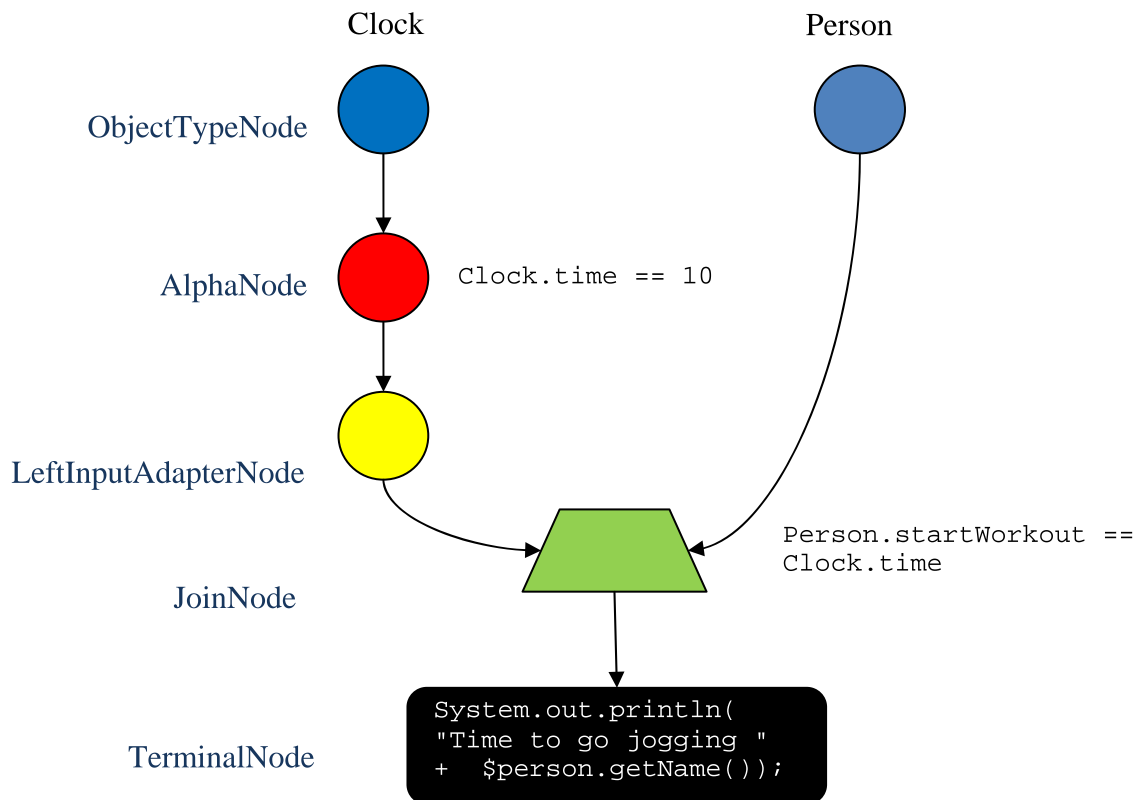
Rete is a pattern matching algorithm for implementing production rule systems and was designed by Dr. Charles Forgy at Carnegie Mellon University in 1979. It is by far the most efficient algorithm for production rule systems ever written. Rete is Latin for *net*. The Rete algorithm is implemented by building a network of interconnecting nodes. Every node representing one or more tests found on the statement part (when-part) of a rule. Facts that are being added or removed from the *working memory* are processed by this network of nodes. You can say that it filters data as it propagates trough the network (15) (16).

Let's take a quick look at an example of how the nodes are created from a rule. In Example 4.2 you can see the code for a simple rule. We have two types of facts here: *Clock* and *Person*. What is expressed in this rule is that if the time of the clock is 10 the check if the

person is supposed to start a workout at 10. If this is true, print out "`Time to go jogging <name>`".

```
rule
when
      Clock( $time : time == 10)
      $person : Person ( startWorkout == $time )
then
      System.out.println("Time to go jogging " +
      $person.getName());
end
```

Example 4.2: Rule code



Example 4.3: Rete nodes

In Example 4.3 we can see Example 4.2 drawn up as Rete nodes. We can see that we have two *ObjectTypeNodes*: *Clock* and *Person*. The *Clock ObjectTypeNode* is propagated into an *AlphaNode* that is used to evaluate a literal expression. In this case the literal expression is

"`Clock.time == 10`", and the literal expression must be satisfied before it can proceed to the next node. The *JoinNode* is used to compare two Objects, their fields, to each other. By convention we call these two inputs left and right. The left input is usually a list of Objects[8], while the right is a single object. This list of Objects is created in the *LeftInputNodeAdapter*. A *TerminalNode* is used to indicate that a single rule has matched all its conditions. We say that the rule has a "full match" at this point (16).

4.3 Drools

The rule engine chosen for this thesis is the Drools (also called JBoss Rules[9]) rule engine. The Drools project was started by Bob McWhirter in 2001 and registered a SourceForge[10]. Drools 1.0 was never released as the limitations of a brute force linear search approach were soon realised, and work was started on Drools 2.0, which was loosely based on the Rete Algorithm (see chapter 4.2.2), and the project was moved to Codehaus[11]. In October 2005 Drools was federated into JBoss and rebranded JBoss Rules, but in May 2007 it was found that the community was still predominantly calling the system Drools. So even if it is technically called JBoss Rules today, we will also in this thesis refer to it as Drools as the rest of the community does. Having become federated by JBoss and with financial backing version 4.0[12] of Drools was rewritten with a complete and enhanced Rete implementation with a GUI tool. Note that version 4.0 was not backwards compatible with older versions (17).

The Drools rule engine gives us great way to collect complex decision-making logic and work with data sets too large for humans to effectively use (2). Drools is a forward chaining rule engine, also called a production rule system (17). Forward chaining starts with the available data and uses inference rules to extract more data until an optimal goal is reached. It will search trough the rules until it finds one where the "when-part" is true, then the "then-part" is

---

[8] In Drools this is called a tuple
[9] http://www.jboss.com/products/rules
[10] http://sourceforge.net/projects/drools/
[11] Codehaus in an open-source project repository with a strong emphasis on Java. See http://codehaus.org for more information.
[12] The current version of Drools (Jboss Rules) is 4.0.3

triggered. This can result in new facts for the working memory, or an external action to be taken (18).

4.4 JSP

To display the Graphical User Interface (GUI) of the Rule Editor user component, the choice fell on making it web-based using JavaServer Pages (JSP). There where two main motivations behind choosing a web-based approach. Firstly web-pages are an easy way to make quite complex input mechanisms, and secondly there is already a built in web-server in Argos.

JSP technology provides an easy way to create to create dynamic web pages and simplify the task of building web applications (19). Argos can supply us with a finished web-server component: *!!Jetty* . Jetty is a full featured web server written in Java. The web server also includes a servlet container based on Tomcats Jasper engine (3). Since all the things needed to run JSP pages are readily available we can go right to the programming of the pages, instead of having setup anything more in Argos. The only thing the component programmer has to do is to put the JSP files a folder named “web” in the component to have the pages deployed in Argos. The pages can be found at *http://localhost:8080/ServiceName*. Another clear advantage by using a web interface compared to a traditional GUI (e.g. Java Swing) for the service is that it can be accessed from remote computers. This makes the user able to work against the service from anywhere in the world with an internet connection.

4.5 AJAX with Scriptaculous

JSP is not in itself enough to create responsive web pages. Asynchronous JavaScript Technology and XML (Ajax) can be used to create more responsive web-interfaces. With Ajax it is possible to make web applications that look and act very similar to traditional desktop applications without relying on plug-ins or browser specific features. Web applications have traditionally been a set of HTML pages that must be reloaded to change any portion of the content. Technologies such as JavaScript and cascading style sheets (CSS) have

matured to the point where they can be used efficiently to create very dynamic web applications that will work on all of the major browsers (20).

Scriptaculous is a JavaScript library that can be used in combination with Ajax to give advanced responsive GUI functionality. A developer can import the Scriptaculous libraries into a web page and use JavaScript to access a simple API that with a few lines of code visual effects such as drag-and-drop (21).

5 DESIGN

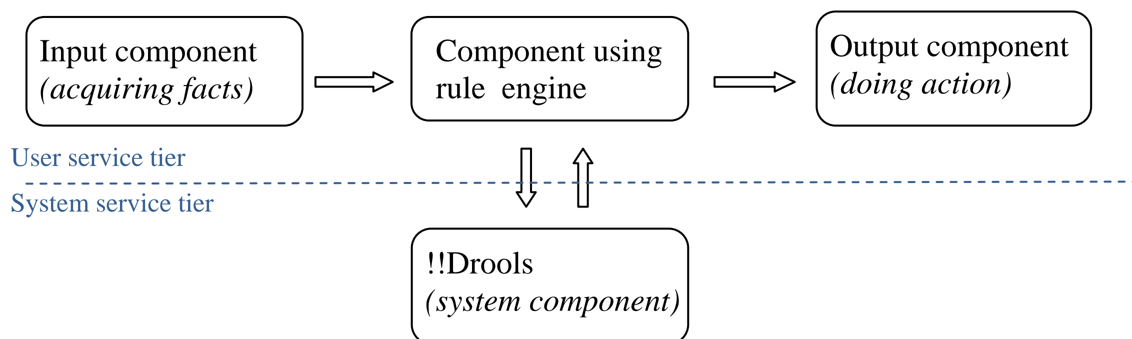
In this section we will first look at the Drools system service which provides user services with rule engine support. Note that “*!!Drools*” is both the name of the system service and the main component in that service. The *!!Drools* system service also contains the *!!DroolsCompont* component. It is responsible for listing up the components compatible with Drools. We will then look at the two parts of such a user service; the Rule Editor user service and the web-based Rule Editor client that belongs to it.

5.1 !!Drools system service

The *!!Drools* system service provides other Argos components with a Drools rule engine. The component is built so that it will give components access to one instance of one *drools rule base* that again is used to initialize one *drools working memory*. A rule base is a set of rules specified in the drl rule language. It is against these rules that the facts are tested to see if any action need to be taken. When the *rule base* has been updated it is realized as a drools *working memory*. It is in this *working memory* that facts are inserted into to try to fire the rules.

5.1.1 Drools role in Argos

In Argos a rule engine simplifies the data flow trough a service (see fig.Example 5.1) by getting the input from other components, checking if this new input makes a rules become true, and if it does start an action operation in an other component.



Example 5.1: Program flow in Argos with Drools

5.2 !!DroolsComponent

For certain types of components that uses the *!!Drools* there will be a need to list up other suitable components that can be used as facts or actions. If the component developer knows which components he wants to use, there is no need for the *!!DroolsComonent* functionality, but in the cases where he needs to either let the end user pick the input or output, or they are handled in a dynamic way, there will be a need to list up all the components in Argos that can be used for this. A component that can supply the input to a service using drools we will call a *fact-component*, since it supplies us with a fact for the rules engine. In the same way a component that supplies us with an output method we will call an *action-component*, since it will supply the drools component with an action that can be taken when new facts fire a rule. This will create a very natural flow of data trough the system. Input comes from a sensor to a component in Argos; this input will be a fact that can trigger predefined rules in Drools that will cause an action. This action will be a method that may give the user some output, and is also implemented as another component in Argos. We can here see that the component using drools in a very natural way, takes us from sensor input, trough rules, to a response to the user by invoking another component.

5.2.1 Annotations to specify facts and actions

In principle any method at all that returns a value may be used as a fact, same as every method that “does” something can be used as an action. But if we imagine that the Argos container can theoretically contain thousands of components, it will be quite unruly to pick out which are suitable as rules or actions. This is why *!!DroolsComponent* contains a method to get out a list of components and their methods that are suitable as facts or actions. To populate this list with components a set of java annotations has been created so that component programmers easily can annotate their code to pick out drools compatible methods in their components (see Table 5.1 for supported annotations).

Annotation	*Meaning*
@DroolsFact	This class has methods that can be used as Drools facts
@DroolsFactMethod	This method can be used as a Drools fact
@DroolsAction	This class has methods that can be used as Drools actions
@DroolsActionMethod.	This method can be used as a Drools action

Table 5.1: Supported annotations

As of Java release 5.0, the platform has a general purpose annotation facility that permits you to define and use your own annotation types. Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time (22). In Argos annotations are read as the component is loaded in. By use of notifications *!!DroolsComponent* is notified that a new component is loaded and checks if any relevant annotations are in the file. If it is it keeps record over which components can be used as facts and actions based on the annotations. In example Example 5.2 and Example 5.3 you can see how the annotations have been used in a java-file to show that a class' methods can be used either as facts or as actions.

```
@DroolsFact
public class MyComponent{
   @DroolsFactMethod
   public String myFactMethod(){ … }
}
```

Example 5.2: Fact annotation

```
@DroolsAction
public class MyComponent{
   @DroolsActionMethod
   public String myActionMethod(){ … }
}
```

Example 5.3: Acton annotation

5.2.2 XML file to specify facts and actions

Most components are not written to be specifically used with Drools. You wouldn't simplify anything if you would have to go into the code of existing components to add annotations just

to be able to use them with drools. This is why there is a second way of specifying which components are compatible with Drools components. For system components or components written by other people, you can write the specifications into one of two xml files. For fact components you may use *fact_components.xml* and for action components use *action_components.xml* (see example Example 5.4) to give information on which components are to be set as Drools compatible.

```
<?xml version=”1.0”?>
<Drools>
  <Class name=”MyComponent”>
    <Method name=”myFactMethod” />
  </Class>
</Drools>
```

Example 5.4: action_components.xml

5.3 User services in general

The *!!Drools* does not by itself provide any functionality to the Argos user, but rather gives component programmers a new tool. This tool may be used by component programmers to easily set up a program flow involving getting input from a component, using the *!!Drools* component to set up the rules and facts, and fire some operation in a component that gives output. The whole program flow, including getting the facts from sensors, doing operations against !!Drools and returning output can be done in one component. But the more elegant solution is using other components for input and output as explained above.

5.4 Rule Editor user service

The Rule Editor component uses this new tool to provide the user with a generic tool to create Drools services for Argos. It lets the users select facts to be used in a Boolean statement. When the statement is true an action is triggered. As described in the previous chapter both facts and actions are chosen from methods in the components in Argos. The chosen statements and actions are then turned into a rule that can be understood by Drools and sent into the *!!Drools* system component. See Figure 5.1 for an overview of the system as a whole.

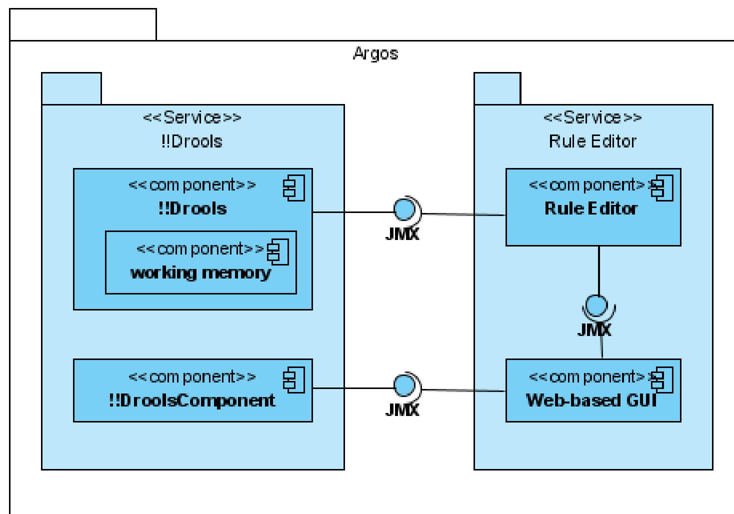


Figure 5.1: Component diagram of !!Drools and Rule Editor

5.4.1 The graphical user interface (GUI)

The Rule Editor component was written to be as generic as possible to be able to be used in many different contexts. The same thing goes for the GUI of the Rule Editor. The GUI is created so that a programmer can sit down and create a service in Argos using input from components and triggering action to happen. See Table 5.2 for areas of interest in the GUI.

Area	*Functionality*
Canvas	The canvas is the middle area of the screen where the rules are listed. The rules have two parts. One statement part consisting of three boxes and one action part that has one box.
Facts	The facts are listed on the left side of the screen. The fact list is populated by the facts specified by annotations and the specifications from the *fact_components.xml* file. The fact-methods can be pulled to the two boxes on each side of the operator box in the statement part of the rule.
Actions	The actions that can be triggered by the rule are listed on the right side. The list is populated by annotations and the specifications from the *acton_components.xml* file. The actions-methods can be pulled to the action part of the rule in

	the canvas.
Operators	The operators are listed on the top of the screen. They decide the relationship between two facts if a rule should be triggered. They can be pulled into the operator box between the two fact boxes on the canvas.

Table 5.2: Areas of interest in the GUI

6 IMPLEMENTATION

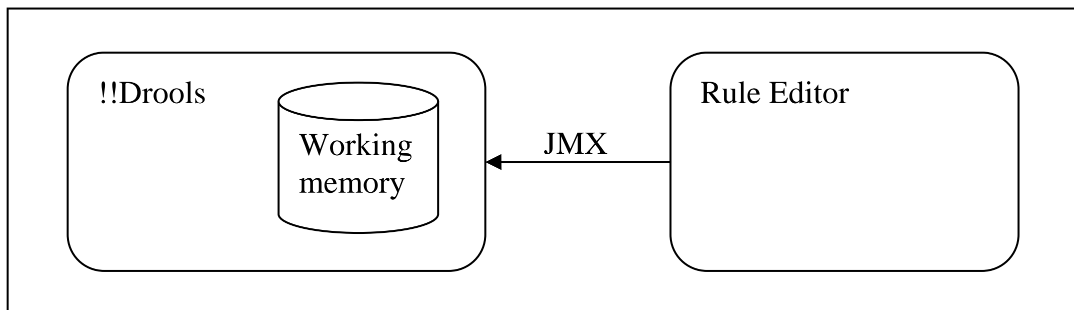
In this chapter the implementation will be outlined. First there will be a closer description of implementation of the *!!Drools* system service. Then we will look closer at the implementation of the *Rule Editor* user component that uses the system service.

6.1 Drools system component

6.1.1 !!Drools

The !!Drools component starts one instance of a Drools *working memory* in Argos. The user may fill it with rule sets and facts and this will again trigger other components in Argos to do their work as described in previous chapters. When creating this component there was a question if there should one *working memory* <u>for each</u> component using *!!Drools* or if there should just be one *working memory* where everything is put into. The conclusion fell on using a singe *working memory* because it gives us a simpler model. Like in the Argos container it gives us one "box" to put everything into and the contents will live their own life in there. The question is: can using only a singe *working memory* lead to problems with naming or slow down the system? Naming can only be a problem as long as the user is using the class as base for facts, because the components wouldn't be able to tell a difference between who the facts are meant for. On the other hand, if two components are using the same class for the rules coming in it's more likely that these components are waiting from input from the same source, and it's a good thing that the input is sent to only one *working memory*. When it comes to slowing the system down by having very many rules and facts in one *working memory* it should scale well enough for Argos' purposes.

The operations provided by the *!!Drools* component are basically a wrapping over the basic functions for a *working memory* in Drools. The operations called are passed on to the instantiated *working memory* running in the component. See Example 4.1 for a simplified model of the system.



Example 6.1: Simplified architecture

6.1.2 !!DroolsComponent

As described in chapter 5.2 there are two different ways to register a component to be compatible with Drools. When writing a new component that is suitable to be used as a *fact-* or *action-component* in another component using *!!Drools* the easiest way is to use the annotations framework provided by the *!!DroolsComponent* class. When a component is launched by Argos a notification will be sent out to the components running, and it will be picked up by *!!DroolsComponent*s notification handler class (*DroolsComponent.handler(Notification n)*). This class will then check if the newly launched classes contain any of the Drools notifications: *@DroolsFact*, *@DroolsFactMethod*, *@DroolsAction*, *@DroolsActionMethod*. If any of these annotations are present the components name and the name of the methods that are suitable for facts or actions are stored in a list for later use. When another component at some later point wonders which components are available as facts then can easily just query *DroolsComponent.facts()* or *DroolsComponents.actions()* to get lists over the facts-components and action-components in the system.

The second way to specify that a component is suitable to be used as a fact or action is to use one of the two configuration files: *action_components.xml* (see Example 5.4) or *fact_components.xml*. These files are only parsed when the !!Drools component is first loaded, so if the files are updated the component has to be reloaded for the changes to take effect. The *argos.bangbang.drools.Config* class is responsible for reading in and parsing these

XML-files. The files are parsed using the Simple API for XML (SAX[13]) which is a standard interface for event-based XML parsing.

6.2 RuleEditor

6.2.1 User component with web-based interface

The Rule Editor is where the user can set up relations between facts from *fact-components* to trigger the actions in *action-components*. There are two parts of the rule editor, the component that basically just sends requests on to the *!!Drools* component, and the web-based interface. The web-based interface is coded with JSP and uses Scriptaculous (see chapter 4.5) to get the drag-and-drop functionality. The JSP files are automatically deployed from the components web folder to the Jetty web-server in Argos when the component is deployed. The web files consists of a index page that displays menus for facts, actions and operands for facts, as well as a canvas in the middle where operations from the menus can be dropped into (see Table 6.1 for more info on the web-files).

Filename	*Function*
index.jsp	The framework page that again calls the menus and dropable page.
dropable.jsp	This page is the heart of the application. It is where the rules are listed and where the facts and actions can be “dropped” into by using drag-and-drop.
factmenu.jsp	This is the menu where the *fact-components* are displayed. The methods can be dragged and dropped on the appropriate field in the statements.
actionmenu.jsp	This is the menu where the *action-components* are displayed. The methods can be dragged and dropped on the fields for actions.
operatormenu.jsp	This is where the user can pick the operator to use to compare the facts with.
drools.css	The style sheet for the layout of the pages.

Table 6.1: Files in the web folder

[13] http://www.saxproject.org

In Figure 6.1 we can see the finished web-based GUI with red labels corresponding with the files described in Table 6.1. To open this page you just have to run Argos with the !!Drools.jar fine deployed and open your browser (Mozilla Firefox[14] is recommended) to http://localhost:8080/RuleEditor/ . To start setting up rules the user must first write in the wanted name of the rule in the text field and press "Add new rule". If you press the button without writing a name or with an already existing name you will get a error message. When you have added a new rule you will get four orange blocks in the middle of the screen. When you have dropped something into each of the four blocks you have created a rule. Pick a method from the fact-menu or set a value to drop into the first and third box. Drag an operator from the operator-menu and drop it into the second orange box. This operator will show the test we are making between the facts. If that test turns out to be true then an action will be triggered. Drag a method from the action-menu and drop it in the forth box to show what will happen when a rule triggers. You may also use the orange links under the rules to create more statements (facts and operands) or more actions in a single rule.

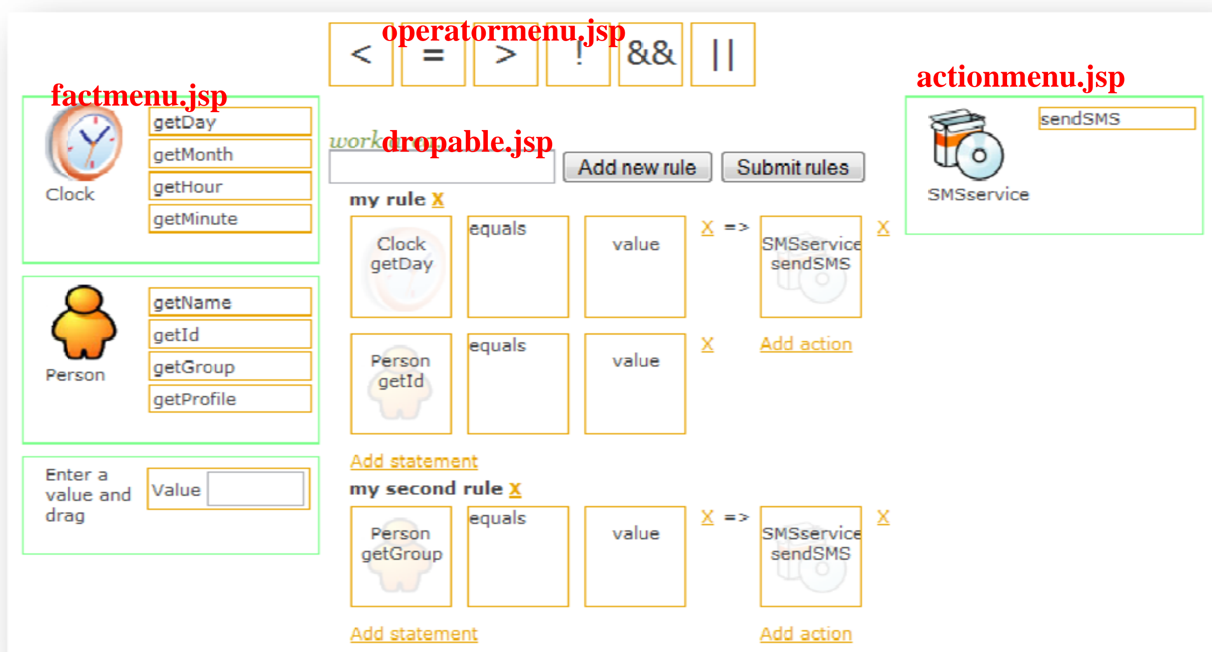


Figure 6.1: Web based GUI

[14] http://www.firefox.com

The components shown in the action-menu and fact-menu are those that have been specified to be suitable facts or actions by the *!!DroolsComponent* (see 0). The menu pages uses JMX in the JSP page to open a connection to Argos. It then invokes *!!DroolsComponent.fact()* or *!!DroolsComponent.action()* to get the appropriate lists over suitable components their suitable methods. The components are listed in small green boxes where the names of the components are shown as id and the names of the components methods are put in orange boxes and made draggable. It is these orange boxes that can be pulled into the rule boxes in the middle. When an item is dropped into one of the boxes in the middle the *droppable.jsp* page is actually reloaded. This is hidden from the user by the use of AJAX (see chapter 4.5). The user will only see the new rule pop into the list. When the page is reloaded parameters are also sent into the new page. The content of these parameters will trigger what happens in the JSP page when it's loaded. Here is a list of the parameters that are used:

f	*r*	*e*	*s*	*d*	*Description*
submitRules					Sends rules to the Rule Editor Argos component
addRule	x				Adds another rule
addStatement	x				Adds another statement a rule
addAction	x				Adds another action to a rule
removeRule	x				Removes a rule
removeStatement	x	x			Removes a statement
removeAction	x	x			Removes an action
setOption	x	x	x	x	Sets a given field to be the value of what was dropped into it.

Table 6.2: Parameters in dropable.jsp

In Table 6.2 you can see the functions that can be triggered, the parameters that is used in the different functions, and a small description of the function. Five of the columns the table only represented with a letter. These letters represent parameters used by the page. See Table 6.3 for a description of these parameters.

Symbol	*Name*	*Description*
f	function	The *f* parameter has the name of the function that is to be used when the page is reloaded.
r	rule	The *r* parameter contains the name (used as id) of the rule. In *addRule* the *r* parameter is used to set the name of the rule. In all the other cases it is used to look up that same rule.
e	element	The *e* parameter points out a statement or action in a rule that should either be removed or be set to a specific value.
s	slot	The *s* parameter says which slot in a rule should be modified. The slots are numbered 0 to 3 where 0 is the first operand, 1 is the operator, 2 is the second operand and 3 is the action.
d	drop	The *d* parameter contains the name of the fact/operator/action that was dropped into this slot.

Table 6.3: Description of the parameters

6.2.2 Component Icon

The component model in Argos was expanded with an icon to be a graphical representation of the components in GUIs. To add this feature to Argos with the least bit of hassle, the icon would simply be a file called *icon.png* in the web folder. Components that need a graphical representation of other components can now easily find it there. Since the *RuleEditor* component has a graphical user interface that uses the icons of other components we get the real advantage of putting it in the web folder. The icon.png file of each component will then naturally be made accessible trough the built in Jetty web-server in Argos. So to get the icon of one component you need only use the URL of that component + “icon.png”. An example can be http://localhost:8080/RuleEditor/icon.png. An advantage with extending the component model in this way is that there are no new rules that are forced on component programmers; rather we are establishing a new convention. It remains to be seen if this convention will be picked up by other Argos developers, but a suggestion has been made on the Argos mailing list.

7 EVALUATION

In this chapter we will go over the finished project and evaluate it against the requirements described in chapter 3. We will look at both the functional and non-functional requirements. We will also evaluate how the scrum worked out as a method, and look at why the technology used was chosen over some of its competitors.

7.1 Functional requirement evaluation

Table 7.1, Table 7.2 and Table 7.3 provides listings of the functional requirements. We will first look at the functional requirements for the system component (*!!Drools*). Then we will look at the requirements for the user component (*Rule Editor*) and last at the requirements for the application that can be created in the *Rule Editor*. The functional requirements were specified in chapters 3.1, 3.2 and 3.3.

Requirement	*Status*	*Evaluation*
DS-1	Success	Creating a component that runs Drools under Argos was a success. Note that there was a bug in Argos for the loading of the library that was worked around for now, and will be fixed in a later release of Argos.
DS-2	Success	The *!!Drools* component provides support for adding rule bases and facts into the rule engine.
DS-3	Success	The *!!Drools* component provides support for removing rules from the rule engine. To remove a rule a String containing the name of that rule must be provided.
DS-4	Success	There has been provided support for annotating files to specify that they can be used as facts or actions by another component in Argos. The *!!Drools* service provides a operation to get a list of *fact-components* and *action-components*.

DS-5	*Success*	*There is support for reading a configuration file that lists the already existing components as facts or actions.*
DS-6	Failure	Persistence has not been implemented yet (see chapter 7.5).

Table 7.1: Evaluation of functional requirements for the Drools system service

Requirement	*Status*	*Evaluation*
DU-1	Success	A JMX connecting can be opened between the user component and the system component.
DU-2	Success	The Rule Editor has support for creating new rules with unique names.
DU-3	Success	More statements and actions can be added to a rule by use of the Rule Editor. There are links in the web-interface under the list of statements and actions in the rule for adding more statements and rules.
DU-4	Success	Facts, statements and actions can be removed again in the Rule Editor. There is an "x" in the upper corner of each of these elements in the web-interface for removing them again.
DU-5	Success	The components suitable to use as facts or actions are listed in each side of the web-interface. They can be dragged-and-dropped into the rule setup.
DU-6	Success	There is support for the basic Boolean operations in the Rule Editor. It supports *equals*, *greater then*, *smaller then* and *not*.
DU-7	Success	A Drag-and-drop GUI has been implemented in the web-interface.

Table 7.2:Evaluation of functional requirements for the Drools user service

Requirement	*Status*	*Evaluation*
RL-1	Failure	This requirement has only been partially implemented. In the end the focus had to be at finishing the Rule Editor while the end service had to suffer. There is a Person component that will become a *fact* in Drools, but all the input methods have not been finished.

RL-2	Failure	There has not been created any component for getting the sensor data from the user.
RL-3	Failure	This requirement was partially implemented, since there is already a component for Argos for sending SMS. It could be used directly as a fact, but the Rule Editor doesn't yet support parameters.

Table 7.3:Evaluation of the Real life usage of the User service

7.2 Non-Functional requirement evaluation

We will in this chapter take a look at the non functional requirements listed in chapter 3.4.

Requirement	*Status*	*Evaluation*
NF-1	Success	This requirement stated that the both the system- and user-service should be as generic as possible. This requirement has been met. The system service was designed and implemented so that any component can use the *!!Drools* component to get basic rule engine support, while the *Rule Editor* component lets the user create any type of rule based service.
NF-2	Success	When having a choice between several solutions the simplest should be chosen. Of the well suited technologies looked at for this project it is our opinion that the technology chosen is the simplest of the pack.
NF-3	Success	It was a requirement to use the already existing technology in Argos when it existed. This is shown in the general design of the Rule Editor when it requires the user to use the components in Argos as facts and actions. It also shows in the choice of GUI when it was decided to use the built in web-server functionality instead of creating a new free standing GUI.

Table 7.4:Evaluation of non-functional requirements

7.3 Method - Scrum

As described in chapter 0 the method used in this thesis was Scrum. The book Agile Software development with Scrum (4) was used as a reference both before starting and during the work. Software from Scrumworks was used to manage the scrum project and setup the backlogs. The project started with identifying as many tasks as possible for the product backlog. Items were then picked from the product backlog, and put into the first sprint backlog and work could start on the tasks in the first sprint. A rule in Scrum is that no new backlog items should be added to the sprint backlog after the sprint had started. This was practically impossible in the first couple of sprints since new important tasks appeared, that was critical just to get the project on the way. After the initial few sprints experience had been gained in using Scrum, and the backlogs had been filled a bit more out with the important tasks to do. It was now easier to stay true to the sprint backlog. We didn't have daily scrums, but rather a set weekly meeting on Fridays and discussions whenever it was needed with the Scrum master. This is also not completely in accordance with the scrum way, but because of the situation around writing a thesis, Scrum had to be adapted to the realities at hand. A sprint lasted two weeks and then sprint review was held to discuss the previous sprint and plan the next one. It worked very well to have sprints as short as two weeks. And it was motivating to have a short deadline to work against.

The biggest challenge with using Scrum was that the team member continuously needs to estimate the time left on a task or a sprint. When a day's work has been done the team member uses the Scrumworks program to modify the number of hours left on a task. The difficulty here is to be accurate enough. If the task is finished with time to spare, then good and well, but as soon as there is much more work in a task then anticipated, then it can even mean that the sprints goals aren't reached. One reason for wrongly estimating the time needed for a programming task can be that the time for learning new technology was not taken into consideration when estimating the time usage for a task. This happened during the implementation on one sprint in this thesis, and all the tasks were not finished at the end of the sprint. The only logical thing to do was to move the remaining tasks to make them part of the next sprint. Not a very big deal really, but it is easy to become unmotivated when you don't reach a set goal for a sprint. A small motivation talk with the scrum master was luckily all that was needed in my case to get the motivation up again. There is no reason to dwell on not

reaching a goal at one sprint, just continue the work and get on. There is enough room in Scrum to adapt if something doesn't go exactly as planned. There may of course be a deeper underlying problem if goals are not reached sprint after sprint.

In conclusion, Scrum is a very straight forward method, and it feels very intuitive to use. To get the full benefits of Scrum you will need to acquire a certain amount of experience in estimating the number of hours a specific task will take. It is obvious that dividing a task up into as many subtasks as possible, in the planning stage, simplifies this. Even though this project was a "solo scrum" with only one team member, one can see the benefits it would bring when working in a larger team too.

7.4 Technical solutions

Because of the objective of the thesis was to create rule engine support for Argos, the Argos middleware framework was a given technology for the development, but some of the other technologies used wasn't as given in the beginning. We will here take a look at some of the technologies that was chosen to be used, and some of the technologies that was chosen to be used.

7.4.1 Rule Engines

Since Argos components are implemented in Java can simplify the implementation to use a rule engine that in Java too. Luckily there is a vide range of rule engines in Java available. To name a few there is Drools, Jess, SweetRules, Mandarax and JRuleEngine (23). There have been done some tests with both Drools and Jess in Argos before, so these two were the most likely rule engines to use as a starting point. These two rule engines are some of the most widely used and it is a good for a programmer to know that the libraries used in an application is well tested and used in production before. There have already been said a lot about Drools in this thesis, so we will first look a bit closer on the Jess rule engine.

7.4.1.1 Jess is a popular rule engine and scripting language developed at Sandia National Laboratories in Livermore, California in the late 1995. The CLIPS[15] system shell, an open-source rule engine written in C, was the original inspiration for Jess. Jess and CLIPS have very different implementations, and Jess also have support for Java's powerful APIs for networking, graphics and database access, but the rule languages in these two systems are still very similar. A downside with Jess is that unlike CLIPS it's not licensed as Open Source. Jess do provide educational licences free so it could have been used as part of this thesis, but if the application were to live on, Open Source is clearly preferred. An advantage with using Jess is that it has support for enterprise environments like J2EE (15). This may have been a great advantage when pairing it with JMX in Argos, but since Jess was not the rule engine chosen for this thesis it has not been looked closer into.

7.4.1.2 Drools was the rule engine chosen to be used in Argos (see chapter 0 for an introduction rule engines and Drools). Drools have for several years been a leading Java Open Source Rule Engine and it also has a strong online community. (17) A key factor in picking Drools as the chosen rule engine was that there has already been some work done with combining Drools with Argos by Eystein Måløy Stenberg. Unfortunately it turned out that this work was too interleaved with the application it was made for to be of much use for implementing the kind of generic service wanted in this thesis.

7.4.2 Graphical User Interface

7.4.2.1 Java Swing was one of the original candidates when it came to choosing a GUI. Swing is part of JFC (Java Foundation Classes) and which encompass a group of features for building graphical user interfaces and adding rich graphics functionality and interactivity to Java applications. (24) An advantage with Swing is that NetBeans IDE has support for creating interfaces trough a graphical WYSIWYG[16] editor. Unfortunately drag-and-drop (requirement DU-7) does not have very good support when creating Swing under NetBeans. The advantage of Swing under NetBeans then disappears. Creating Swing-interfaces with

[15] http://www.ghg.net/clips/CLIPS.html
[16] What you see is what you get

traditional Java programming can be very complex for systems as advanced as this and this would be a disadvantage under requirement NF-2.

7.4.2.2 JSP and a web-based approach were chosen as the GUI in the end. The non-functional requirement NF-2 states that: "when it comes to technology choices the simplest solution of several suiting choices should be picked". Web-pages give a very strong tool to simply create very complex user interfaces. The use of Cascading Style Sheets (CSS) lets the programmer separate the functionality completely from the layout, yielding code that is both easy to read and to extend with more functionality. JSP also have very good built in support to work with JavaBeans. This is very useful when creating more complex classes, and lets the programmer test out the JavaBeans in the developer environment of his choice before putting it into the web interface. It can be challenging to debug web-interfaces, but having the advantages described above, it is not a hinder in the development. Non-functional requirement NF-3 states that already built in technology in Argos should be chosen instead of writing new technology that does the same. In Argos there is no real convention for creating user interfaces. But since Argos already have support for JSP, NF-3 points us in that direction. In retrospect I don't think much time was saved or wasted by choosing JSP over Swing, but the solution of using JSP is more elegant, since it builds on what is already in Argos. A web-based interface can also be accessed from any computer on the Internet, which enables the user to work remotely against the service.

7.4.2.3 Scriptaculous was the library chosen to get drag-and-drop functionality, according to requirement DU-7, in the web-interface. It was obvious that some kind of JavaScript should be used to get drag-and-drop in JSP, and Scriptaculous gives exactly that in a few lines of code. There are a few libraries like Scriptaculous available for free online (e.g. DragLib, WebToolkit, Django). The advantage with Scriptaculous is that it only takes a couple of lines of code to get all the functionality needed, and that they had several examples on their web-page that was close to what was needed in this project. Non-functional requirement NF-2 says that the simplest solution should be picked and in this case Scriptaculous appeared to be the simplest.

7.5 Remaining work

There was some functionality that there wasn't time to finish implementing.

Remaining work	*Description*
Hibernate	There was originally a plan to use Hibernate to give the rule engine persistence according to requirement DS-6. In the end it was decided that it wasn't necessary for demonstrating the core functionality of the system, even if it's necessary to have in real practical use.
Input for actions/facts	As the system is now, only fact-methods an action-methods without parameters can be used. The system needs a popup window to input the parameter in the user interface, like the functionality found in Argus.

8 CONCLUSION

8.1 Achievements

The problem definition in chapter 1.2 states:

> *The main goal of this work is to develop a system service for Argos that combines the rule engine with the existing functionality to get data from sensors to an integrated system. The goal is that it should be simpler to create services that combine sensors and rules in a natural program flow. There will also be developed user service in Argos that offers access to the to the Rule engine trough a simplified GUI.*

During this thesis we have developed a prototype of the system service described in the problem definition above. The *!!Drools system service* offers an API to other components so that they can add and remove rules and add facts to a running rule engine. To get the data from a sensor, the facts are input from other components in the rule engine. This covers both components getting input from sensors and any other kind of components that returns some value.

The *Rule Editor user service* offers a simple tool for inputting rules into the rule engine running in *!!Drools*. The user can pick from lists of components methods to use as facts in the rule expressions and from a separate list of component methods to specify the action that is triggered when a rule fires. We now get a very natural flow of data trough the system. We use a *fact-component* already in the system to get the input. The rule engine in the *!!Drools* component can use the action to trigger the *action-component* that gives the user some output.

8.2 Future work

In this section we will present some areas in which there could be interesting to do some future work on the system. See Table 8.1 for a list over suggestions for future work.

Future work	*Description*
Set up the Lifestyle service with the Rule editor	The Lifestyle service that has been described in this thesis has only been partially implemented. It needs more components to give it facts and the SMS component for Argos has to be supported by the Rule Editor in full.
Support more drools functionality	The system component should be extended to support a bigger part of the functionality that Drools offers. The Rule Editor can also be extended to support the same functions that the system component has been extended with.
Simplify input further in system component	The functionality in the rule editor that compiles the rule could probably be moved to the system component. This method would take two facts one operator and one action as input and compile this to a String that is put into the rule base as a new rule.
Combine Drools graphical input tool with system component	The newest version of Drools is bundled with a graphical input tool that is more complex and generic then the one created in this thesis. It would be interesting to design and set up a workflow where that tool is used to create the rules.

Table 8.1: Future work

8.3 Conclusion

Throughout this thesis a system service and a user service for Argos have emerged. We have showed how the Rule Editor user service lets a user graphically orchestrate the program flow of a new service by using already existing components and the !!Drools system component. We have also looked into creating a Lifestyle service using the rule editor.

We have learned about how Rule engines work, and how to write rules in the drl rule language. By designing and implementing components for Argos we are also left with a much deeper understanding of Argos and its component model. Taking advantage of the built in

features and components from 3rd party developers can cut down on the development time of projects.

To work trough the process of creating software from the conception phase until the result you see today have also been very rewarding. The Scrum method of developing software gives us a good tool to manage projects. And even if it was done solo this time I feel better prepared to go out into the industry and work in scrum teams after this experience.

9 BIBLIOGRAPHY

1. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. *http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html.* [Online]

2. Jboss Rules factsheet. *http://www.jboss.com/pdf/jboss_rules_fact_sheet_04_07.pdf.* [Online]

3. **Eriksen, Dan Peder.** *Argos Container, Core and Extension Framework.* s.l. : University of Tromsø, 2007.

4. **Schwaber, Ken and Beedle, Mike.** *Agile Software Development with Scrum.* 2002.

5. **Takeuchi, Hirotaka and Nonaka, Ikujiro.** The New New Product Development Game. *http://harvardbusinessonline.hbsp.harvard.edu/b02/en/common/item_detail.jhtml?id=86116.* [Online] Jan-Feb 1986.

6. Scrum (development) - Wikipedia. *http://en.wikipedia.org/wiki/Scrum_(development).* [Online] 2007.

7. **Svendsen, Gunvald Bendix.** Lifestyle change by divine intervention. 2007.

8. *FACTS - A Rule-based Middleware Architecture for Wireless Sensor Networks.* **Terfloth, Kirsten, Wittenburg, Georg and Schiller, Jochen.** s.l. : IEEE, 2006.

9. *Marked research about lifestyle tools.* **Almås-Sørensen, Live.** s.l. : Telenor R&I, 2007.

10. *Mobile Persuasion: 20 Perspectives on the Future of Behavior Change.* **Fogg, BJ and Eckles, Dean.** s.l. : Stanford Captology Media, 2007.

11. *Ubiquitous Computing Technology for Just-in-Time Motivation of Behavior Change.* **Intille, Stephen S.** 2003.

12. **Perry, J. Steven.** *Java Management Extensions.* 2002.

13. Hibernate Reference Documentation. *http://www.hibernate.org/hib_docs/v3/reference/en/html/.* [Online]

14. Rules-based Programming with JBoss Rules/Drools. *http://www.codeodor.com/index.cfm/2007/9/10/Rules-based-Programming-with-JBoss-RulesDrools/1600.* [Online] 2007.

15. **Friedman-Hill, Ernest.** *Jess in Action.* s.l. : Manning Publication Co., 2003.

16. **Proctor, Mark.** Drools Documentation. [Online]

17. Drools - Wikipedia. *http://en.wikipedia.org/wiki/JBoss_Rules.* [Online] Dec 2007.

18. Forward chaining - Wikipedia. *http://en.wikipedia.org/wiki/Forward_chaining.* [Online] Dec 2007.

19. JavaServer Pages Technology. *http://java.sun.com/products/jsp/.* [Online]

20. Asyncronous JavaScript Technology and XML (AJAX).
http://java.sun.com/developer/technicalArticles/J2EE/AJAX/. [Online]

21. Scriptaculous documentation. *http://wiki.script.aculo.us/scriptaculous/.* [Online]

22. Java Annotations.
http://java.sun.com/javase/6/docs/technotes/guides/language/annotations.html. [Online]

23. Open Source Rule Engines in Java. *http://java-source.net/open-source/rule-engines.* [Online]

24. Creating a GUI with JFC/Swing. *http://java.sun.com/docs/books/tutorial/uiswing/.* [Online] Sun Microsystems, Inc., 2007.