Universiteit Leiden
The Netherlands

Department of Computer Science

Open source systems for dialogue systems

Niek Buwalda

Supervisors:
First: Wojtek Kowalczyk
Second: Koen van der Blom
External: Rens de Wolf

BACHELOR THESIS

Tracks Inspector
Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl`

12–09–2017

Abstract

The Sweetie 2.0 system built by Tracks Inspector, is a system that tries to gather contact information from men who are looking for webcam-sex with children in public chat rooms, and identify them. It uses a commercial chat bot that lacks some desired functionality. As the system will be used by other parties in the future, a new chat engine with more flexibility and support for natural conversations is required. During a 4 month internship my tasks were to choose this new chat engine, assist with the implementation and to support the operators of the Sweetie 2.0 system use the engine. To complement and eventually replace the current chat engine of the Sweetie 2.0 system, the SuperScript engine has been chosen. I selected SuperScript from a collection of some of the most used open-source chat engines based on criteria determined by all parties involved. Furthermore, to support the writers of the conversation scripts, a web-based edit and test environment was created. Finally, the SuperScript engine and the web-based editor have been incorporated into the Sweetie 2.0 system.

Acknowledgements

First, I would like to thank the board of Tracks Inspector for the opportunity of this internship.
Furthermore, thanks to my supervisors from the University for their comments and remarks.
Also, I would like to thank the developers of SuperScript for answering my questions and helping me find solutions to the problems I faced when I met unexpected behaviour of the chat bot engine.
And last but definitely not least I want to give my thanks to every one that works at Tracks Inspector for my incredible educational and also fun time there.

Contents

Terminology

AIML

Artificial Intelligence Markup Language. A popular script language for dialogue systems.

AJAX

Asynchronous JavaScript And XML.

API

Application Programming Interface.

Gambit

A line in the script of a chat bot, that represents the user input and output options.

No match

An unmatched input sequence.

POI

Person of interest.

Wildcard

Wildcards are not exact string matches, but are based on character pattern matching. A wildcard can represent zero or more tokens or characters.

WordNet

WordNet is a large lexical database of English, maintained by Princeton University. [1]

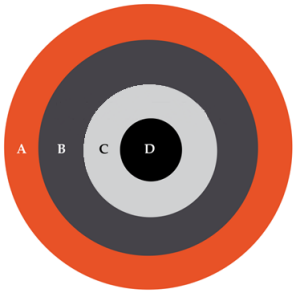
Chapter 1

Introduction

1.1 Fighting abuse of children online

Terre des Hommes, a non-profit organisation that protects the rights of children across the world, estimates that at any moment 750.000 men are looking for webcam-sex with children in public chat rooms [2]. They consider themselves untouchable, hiding behind fake identities. To battle this abuse Terre des Hommes tries to gather information about these men. The information is used to confront the men and possibly scare them off. This approach is based on a theory of professor Bogaearts of the University of Tilburg [3]. According to this theory, there are four stages someone who is looking for child pornography can be in, illustrated in Figure 1.1: First they are just curious (A), then they become regular viewers (B), stage three is collecting child pornography (C) and the last stage is actually abusing children themselves (D). About 25 percent of viewers of child pornography, become abusers of children. When people are confronted with their actions and showed that they are unethical in early stages, the chances they move on to the next stage will be strongly reduced. Because of the size of the problem, Terre des Hommes has commissioned Tracks Inspector to build a system to automate the gathering of the information and sending intervention messages, hopefully scaring offenders off. This system is called Sweetie 2.0.

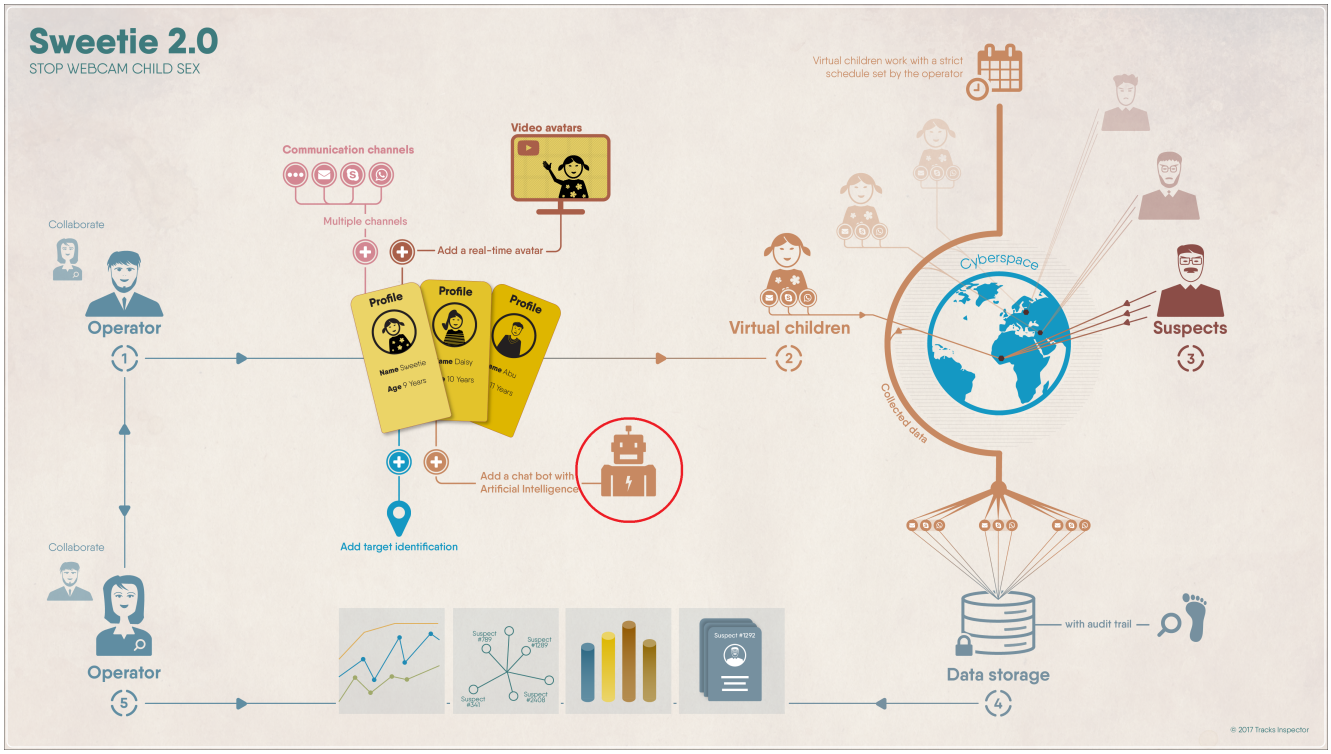
Figure 1.1: Stages of child pornography watchers according to [3]:
(A) Curiosity, (B) regular, (C) collector, (D) abuser



1.2 Sweetie 2.0

The Sweetie 2.0 system, uses a commercial chat bot, circled in red, to talk to people in public chat rooms with the intention to gather contact information. The chat bot is designed to represent a young girl from the Philippines. The output of the bot is generated by a rule-based database system with expected output and corresponding output options. The Sweetie 2.0 system handles all the user input and in and output of the bot concurrently for multiple people. It can also use instant messaging or email for communication and to confirm that retrieved contact information is valid. The bot can be turned off and an operator can manually continue each conversation individually. It is important to mention that the bot does not start a conversation, but waits in the chat rooms to be approached. Each person initiating a conversation with the bot is then treated as a person of interest (POI). The system tries to link relevant POI's to each other. For instance, if the same email address is used by two apparent different users, those two user names get merged into a single POI. This gives an indication if someone is a repeating offender. Operators can view and annotate the chat history of every POI and mark the POI as relevant or not and manually merge with other POI's. The infographic in Figure 1.2 illustrates all the functionalities of the system mentioned previously.
Unfortunately the commercial chat bot, does not always behave properly and lacks certain important functionalities. Due to the fact it was primarily developed to provide e.g. customer service support on a website and not necessary support for natural language, it sometimes is too repetitive and it is not able to determine which state of the conversation it is in. This may expose the bot and leads to bans in chat rooms preventing more potential predators from talking to the bot.

Figure 1.2: Infographic of the Sweetie 2.0 system



Hence a new chat engine is desired. Not only should the new chat engine be able to have more human-like behaviour but it should also be able to switch between 'identities'. It should be able to represent boys and girls from different parts of the world, and it has to fit into the rest of the system. Furthermore, because of the specific goal of this project, the chat bot has to able to reach certain states in the conversation, recognize the state it is in and change its behaviour based on the current state. The goals of the bot are making the user aware of the fact that he is talking to an underage child, determine if the user is looking for sexual contact with the option of payment and retrieving contact information. After reaching one of these goals, the conversation should be send into the directions of the other goals, to prevent repetition.
To find the best suitable open-source chat engine for the Sweetie 2.0 system, I have preformed research during a 4-month internship at Tracks Inspector. This allowed me to familiarize myself with the current Sweetie 2.0 system, and investigate the available chat bot options. By analysing several chat bot engines and testing them for the criteria for the Sweetie 2.0 chat bot, a new chat engine was to be selected. Additionally, a web-based edit and test environment for SuperScript chat bots was created.
Finally, SuperScript was incorporated into the Sweetie 2.0 system, where the conversation scripts can be developed further. This will be done by a researcher from University of Tilburg and an operator from Terre des Hommes, who should be able to do develop the conversation scripts, without interference of Tracks Inspector. This thesis describes the research and the progress of the selection and implementation of this new chat bot engine for the Sweetie 2.0 system.
This bachelor project has been supervised by Rens de Wolf from Tracks Inspector and Wojtek Kowalczyk of Leiden Institute of Advanced Computer Science (LIACS).

1.3 Thesis Overview

This thesis is structured as follows. Chapter 2 describes the domain the chat bot will operate in and how it should behave to function properly. In Chapter 3 the chat engines that have been taken in consideration are evaluated based on established requirements. The thesis continues with Chapter 4, which discusses the process of building of the conversation. After that Chapter 5 handles the issues that were encountered during the use of the new chat engine. This thesis ends with Chapter 6, where some of the made decisions are evaluated and Chapter 7.

Chapter 2

A chat system for online chat rooms

2.1 Chat bots in general

When people think of the term chat bot, they often think about automated chats for online customer service. In that case the bot can be dumb, repeat answers and let the user know it is a bot. Also, the domain of the conversation is closed. Which means the bot only has to be able to talk about several predefined topics.
In this project, this is not the case. The chat bot should always generate a human-like response, because chat sites may have public rooms or fora where people can warn other users that a chat bot is active on the site. The chat bot must generate a response for every input. Another challenge is to steer the conversation in a certain direction. The fact that the bot will operate in a chat room has upsides as well as downsides.

2.2 Chat room specific language

Communication in chat rooms differs much from a real-life conversation. The word usage in chats is different than the real world and can be different per country, room type and age. There is also typical chat language or slang. This is usually an abbreviation of a few often-used words. Two examples:

- asl: age, sex, location
- hru: how are you

2.3 Impersonating a child

The main language in targeted chat rooms is English. Because the chat bot impersonates a child of roughly age ten, the English of the chat bot does not have to be perfect. In fact, it must not be perfect. Spelling mistakes, grammatical errors and a limited vocabulary are expected from a non-native English speaking child. A child

usually does not have a broad general knowledge, interest in politics or philosophy. Children do not know explicit terms, for example Latin terms for body parts or, more related to this project, the name of sex positions. Therefore the domain the responses have to cover is a lot smaller.

2.4 Human-like behaviour

As said before, it is crucial that the chat bot responds like a real human being would. If the bot is uncovered, the username or the IP address it uses could be banned from the chat server or someone can warn other users that there is a chat bot active. First of all it should be able to respond on almost anything, even if it is just quibbling. Second, the chat bot generates a response nearly immediately. For humans it typically takes more time to produce a response, because the message must be typed in physically. This means the responses have to be delayed. But imagine the user sends more messages before a response of the chat bot is send. A person would only respond to one or two questions or statements, but the chat bot would respond to all user input, line by line. This means sometimes one or more messages of the chat bot have to be discarded. Furthermore, the chat bot should not repeat itself too often. The conversation will be steered into the direction of exchanging contact information by the chat bot. When asking for an email-address for example, the bot should not ask for an address or give its own address too many times, to prevent raising suspicion. When a person of interest (POI) shares some contact information, the chat bot should not ask again for the same information. Furthermore, humans also sometimes give wrong answers. The chat bot will give answers based on rules. The rules should sometimes give 'wrong' answers. Another human aspect of chatting is emotions in messages. Think of emoticons and excessive use of capital letters. Response should take these emotions into account to seem more human.

2.5 Behaviour of chatters

Several typical chatting behaviours are observed, and will be discussed:
POIs tend to have multiple chat sessions at the same time. This gives the advantage that, they have a less clear idea of what has been said in the conversation, because it is hard to keep multiple conversations at the same time and keep them apart. Second, they follow routine, meaning they usually follow the same conversation line. Because of this, it is easier to predict the input the chat bot will receive.
Furthermore, people with the intention to get the child in front of a webcam, usually operate with a tunnel vision on their goal. This is a benefit because they do not quickly suspect they are talking to a chat bot. And if they get suspicious, they sometimes even think they are talking to a person pretending to be a child.
Also, there are two general flows of conversations depending on the POIs attitude, which is aggressive or passive. Aggressive POIs do not make much conversation, they rush towards webcam contact. Which makes it harder to retrieve information from them than from the passive POIs, as the window is smaller. The passive POIs take more time for a conversation, with exchanging names for example, this makes the window to

obtain their information larger, but the chat bot must be able to respond to a broader selection of topics. The classification of POIs into profiles is an ongoing research as part of the Sweetie 2.0 project.
Last, POIs sometimes instruct their conversation partners to go to a different room. Usually a private and password protected room they consider safer or a room that is under their control. This complicates the task of the chat bot.

Chapter 3

Selection criteria and chat bot engines

3.1 Criteria

In order to make a good selection for the engine system behind the chat bot, selection criteria were determined by elicitation from the stakeholders. The requirements of Tracks Inspector, the users of the Sweetie 2.0 system and myself have been taken into consideration. The MoSCoW-method [4] was used to rate the requirements. Requirements can have the following ratings:

M - must have This requirement has to be in the engine. It is not useful otherwise.

S - should have This requirement is highly desirable, but the engine would still be useful without it.

C - could have This requirement has a low priority. For example, it will only be considered if there is enough time to realize it.

W - won't have This requirement will not be in the used in this project, but maybe used in a follow-up project

The criteria agreed upon are listed in table 3.1 and discussed in detail in the next subsection.

Index	Criterion	Rating
1	Standalone	M
2	Script language	M
3	Communication	M
4	Separation	M
5	Documentation	S
6	Up-to-date	S
7	History	S
8	Distribution license	S
9	Programming language	C

Table 3.1: Criteria for the new technique

3.2 Explanation of the criteria

3.2.1 Standalone

The chat bot is not cloud-based, it can be deployed as a standalone service without the need to connect to the internet. The chat bot engine should be able to run on its own and no data needs to be exchanged with a third party. In order to keep the system modular and free from any third-party influence, the system should be able to be installed in a closed environment managed by Tracks Inspector or other third party.

3.2.2 Script language

The scripting language is rich enough so that the chat bot has all desired functionalities. Which means it should have the ability to substitute variables, match wildcards, grouping of topics and to select an answer from all possible answers. However, the conversation script languages the chat bot can interpret should be convenient and easy to learn and use. The chat bot will most likely be modelled by non-technical users of the Sweetie 2.0 system. They will know everything about the conversation, but have little to no experience with scripting languages.

3.2.3 Communication

Communication with the chat bot engine is supported through a common application protocol (like HTTP) or can be connected to the Sweetie 2.0 system via an API. The chat bot will have to communicate with the Sweetie 2.0 system. Again, for the implementation this would be the easiest through a common protocol. However, other alternatives like using an API that allows the Sweetie 2.0 system to integrate and control the chat bot engine is acceptable also. Of course the API should be available in a common programming language supported by the Tracks Inspector development team.

3.2.4 Separation

The modelling of the conversation is separate from the chat bot engine configuration. The script wherein the conversation is defined should not include configuration parameters for the chat bot engine itself. Some chat bot engines do not have a clear distinction between the conversation model, the operational parameters and the application configuration. For example, this is all provided in a single configuration file. Given the fact that the people maintaining the chat bots will have no technical background, they should only be bothered with the modelling of the conversations and not the other technical configurations and parameters of the chat bot engine.

3.2.5 Documentation

Documentation, manuals and examples of the installation, configuration, modelling, language and usability are publicly available. Without documentation of the chat bot engine regarding installation, configuration, use etc. the software is either too premature or does not have a serious user-base. Documentation, including examples, regarding the way conversations must be modelled is important for the users of the Sweetie 2.0 system, because it is the most common form of support.

3.2.6 Up-to-date

The chat bot engine is actively maintained. This means bugs or other problems found are typically fixed in the near future. In addition, the software maintainers of the chat bot engine might be available to answer questions and discuss potential issues or mistakes found.

3.2.7 History

The conversation history of the chat bot can be saved and accessed for analyses during the conversation and/or after it has taken place. Analysing conversations that have previously been made can help detect flaws in the conversation flow and understand decisions made by the chat bot. For example, to improve the quality of the chat bot, it is essential to determine what user input triggered so-called no matches.

3.2.8 Distribution license

The software license of the chat bot engine allows the engine to be distributed with a commercial application, without any additional subscription or license cost. Furthermore, the software can be used for a commercial application without the commercial application needing to be open-source.

3.2.9 Programming language

The system is based on a common programming language. Preferably already used for the Sweetie 2.0 system. Using a common programming language supported by the Tracks Inspector development team will save time when implementing the engine into the Sweetie 2.0 system. From a maintenance perspective, the development team may even contribute to the open source project and debug the chat bot engine if required to resolve serious bugs in the software.

3.3 Test conversation

In addition to the criteria above, all considered chat bot engines have been installed and tested if possible with a short test conversation based on the outline below. With this short conversation, the ability to use variables, both pre-defined and created during the conversation, the ability to use user input and choosing different options for a single response is tested. Also, the logging of conversations will be observed. Additionally, this test will give an idea of the complexity of the script language.

User	**Bot**
-Greeting	Greeting (several options)-
-Give name	Acknowledge name and save it-
-Ask name bot	Return predefined bot name-
-"Do you like *?"	"I love *!" (where * is user input)-

Table 3.2: Outline test conversation

3.4 Chat bot engines

The current Sweetie 2.0 system works with a rule-based knowledge management system (KMS) from a third party. As described in the requirements, the chat bot engine could be managed by Tracks Inspector and the operators of the Sweetie 2.0 system, without additional subscription or license cost. Therefore, the decision was made to consider open-source techniques only. The following chat bot engines have been selected and analysed.

1. RiveScript [5]
2. ChatterBot [6]
3. ChatScript [7]
4. Program-O [8]
5. SuperScript [9]

3.4.1 RiveScript

"*RiveScript is a text-based scripting language meant to aid in the development of interactive chatbots. A chatbot is a software application that can communicate with humans using natural languages such as English in order to provide entertainment, services or just have a conversation.*" [5]
RiveScript needs an interpreter, as it is a scripting language, not a complete chat engine. Interpreters are available in Go, Perl, Java, JavaScript and Python, satisfying the programming language requirement. This also means that the script can be written independently from the interpreter, separating the modelling of

the conversation from the chat engine. RiveScript supports wildcard matching. Variables can be predefined in the script of the bot. Variables can be created or changed during a conversation. Possible responses can have different weights. The test conversation was successfully made in RiveScript. RiveScript also supports Topics. Therefore, the script language is rich enough to fulfil the requirement. Furthermore, the communication requirement is satisfied, because the JavaScript interpreter can be used as a module or with JQuery and Asynchronous JavaScript and XML (AJAX) requests and the Python interpreter can communicate using JSON. None of the interpreters save conversations, which means RiveScript does not satisfy the history requirement. Documentation is provided for installation and execution. There is a community for users of RiveScript on Slack and Facebook. RiveScript is licensed under MIT license [10], which is a very permissive license. It allows reuse of proprietary software, as long as every copy of the original software contains MIT License terms and the copyright notice.

Requirement	Score
Standalone	✓
Script language	✓
Communication	✓
Separation	✓
Documentation	✓
Up-to-date	✓
History	×
License	✓
Programming language	✓

3.4.2 ChatterBot

ChatterBot is a Python library for learning bots, which means it has to be trained. The program selects the closest matching response by searching the known statements that matches the input. It then chooses a response from the selection of known responses to that statement. Some input and output adapters are provided, but changes have to be made in order to communicate with the bot. Either in the adapter or the connector. Conversations are saved in a JSON-file, meeting the history requirement. Even to make the test conversation with ChatterBot, the bot has to be trained. For the sake of time and the lack of usefulness of a self-learning technique for this particular project, the bot was not trained. The fact that this chat bot is self-learning, also means that there is no predefined script. Therefore, there is no scripting language, which means for example, user input can not be saved as variable. Furthermore, the conversation can not be modelled. Documentation is provided for installation and execution. There is a ChatterBot GitHub page where issues can be reported. Chatterbot is licensed under BSD 3-clause [11]. This is only different from the MIT license is that the name of the product and contributors may not be used to promote products derived from it.

Requirement	Score
Standalone	✓
Script language	×
Communication	×
Separation	×
Documentation	✓
Up-to-date	✓
History	✓
License	✓
Programming language	✓

3.4.3 ChatScript

ChatScript is an chat bot engine, in C/C++, with its own scripting language. Scripts can be written separately from the chat engine configurations, and support the required functionalities. User and bot variables are available and topics and wildcards are supported. ChatScript can convert statements to JSON and vice versa. ChatScript maintains an independent history with each user-bot combination in a single text file, satisfying the history requirement. Broad documentation is provided. Issues can reported on GitHub and the creator of the program can be contacted via email. ChatScript is licensed under the MIT license.

Requirement	Score
Standalone	✓
Script language	✓
Communication	✓
Separation	✓
Documentation	✓
Up-to-date	✓
History	✓
License	✓
Programming language	✓

3.4.4 Program-O

Program-O is an interpreter for AIML in PHP. Modelling of the conversation is separate from the configuration of the chat bot, because AIML script can be loaded into the bot. AIML supports topics and wildcards. Also variables can be set for the bot. Communication with the bot can be done through a static HTML page, AJAX or the Program-O API. Some documentation is provided for installation and implementation. In order to produce a response Program-O uses a MSQL database. The database was created and linked to the program, but AIML files failed to load into the database. Because of this the test conversation could not be tested

with this technique, and the amount of logging of conversations is unknown. Any issues can be reported on GitHub or the Program-O forum. Program-O is licensed under GNU GPL-3.0 license. This license allow use for commercial applications, but restricts the derived product to be open-source as well.

Requirement	Score
Standalone	✓
Script language	✓
Communication	✓
Separation	✓
Documentation	×
Up-to-date	✓
History	×
License	×
Programming language	✓

3.4.5 SuperScript

"*SuperScript is a dialog system and bot engine for creating human-like conversation chat bots. It exposes an expressive script for crafting dialogue and features text-expansion using WordNet and information retrieval using a fact system built on a Level interface.*" [9]

The bot engine SuperScript is in JavaScript and has its own scripting language, however it is inspired by RiveScript and ChatScript. The scripts can be created separately from the configurations. SuperScript works on the web over WebSocket, HTTP or TCP. User input can be saved and used. With the initialization of a bot, bot facts can be loaded into the bot. Tables of relations between words can be defined, for example opposites. Self made functions are supported. This can be useful to keep track of states of the dialogue for example. Documentation is provided for installation and implementation. The conversation can be separated into topics, but topics can not be nested. Conversations are stored per bot in a JSON-file. The test conversation was created in SuperScript with success. SuperScript is licensed under MIT license.

Requirement	Score
Standalone	✓
Script language	✓
Communication	✓
Separation	✓
Documentation	✓
Up-to-date	✓
History	✓
License	✓
Programming language	✓

3.5 Results

As can be seen in Table 3.3, there a two chat bot engines which satisfy all requirements, ChatScript and SuperScript. The decisive factor for the choice between ChatScript and SuperScript is the support of self-created functions in SuperScript, which could prove useful for the Sweetie 2.0 system. For example, the chat bot asks the user for an email address, but the user ignores the question. As the purpose is retrieving contact information, it should not give up yet. But if the user is stubborn and keeps ignoring the requests, the chat bot should stop asking at some point. However, it is usual in a script that a question can be asked once or infinitely. For this problem, a self-created function which keeps a simple counter could be the answer.

Requirement	Rating	RiveScript	ChatterBot	ChatScript	Program-O	SuperScript
Standalone	M	✓	✓	✓	✓	✓
Script language	M	✓	×	✓	✓	✓
Communication	M	✓	×	✓	✓	✓
Separation	M	✓	×	✓	✓	✓
Documentation	S	✓	✓	✓	×	✓
Up-to-date	S	✓	✓	✓	✓	✓
History	S	×	✓	✓	×	✓
License	S	✓	✓	✓	×	✓
Programming language	C	✓	✓	✓	✓	✓

Table 3.3: Overview of score of the engines

Chapter 4

Building a conversation

After choosing SuperScript as chat bot engine, the conversation-lines had to be made for the chat bot. This task has been taken on by a researcher from Tilburg University and an operator from Terre des Hommes. Because they have little technical knowledge and no experience with computer programming, helping them understand the scripting language and creating extra functions was essential. Hence, a test environment was created to help them focus on the scripting language and not on the technical details of SuperScript.

4.1 Bot editor

This environment is an online editor for the scripts. Also, the plug-ins, (see section 4.2), which contain self-created functions, and concepts (see section 4.3) can be edited. New script file, plug-ins and concepts can be added and deleted. It is important that the chat bot engine is abstracted as much as possible for the users of the editor. Users do not have to worry about file extensions, including concepts in the parse options, connecting to the chat bot and the different directories to store and retrieve the different types of files.
The editor has been slightly altered several times based on suggestions by the users of the environment. For example, the original version opened a new window when a chat was initiated. In order to check changes made to the script, all users agreed that the script and the chat should be displayed next to each other on the same screen, see Figure 4.1 and Figure 4.2. Also, because of the addition of concepts, an extra tab had to be added in the editor interface.
Eventually the creation of new chat bots, building their conversations and testing them would be integrated in the Sweetie 2.0 system.

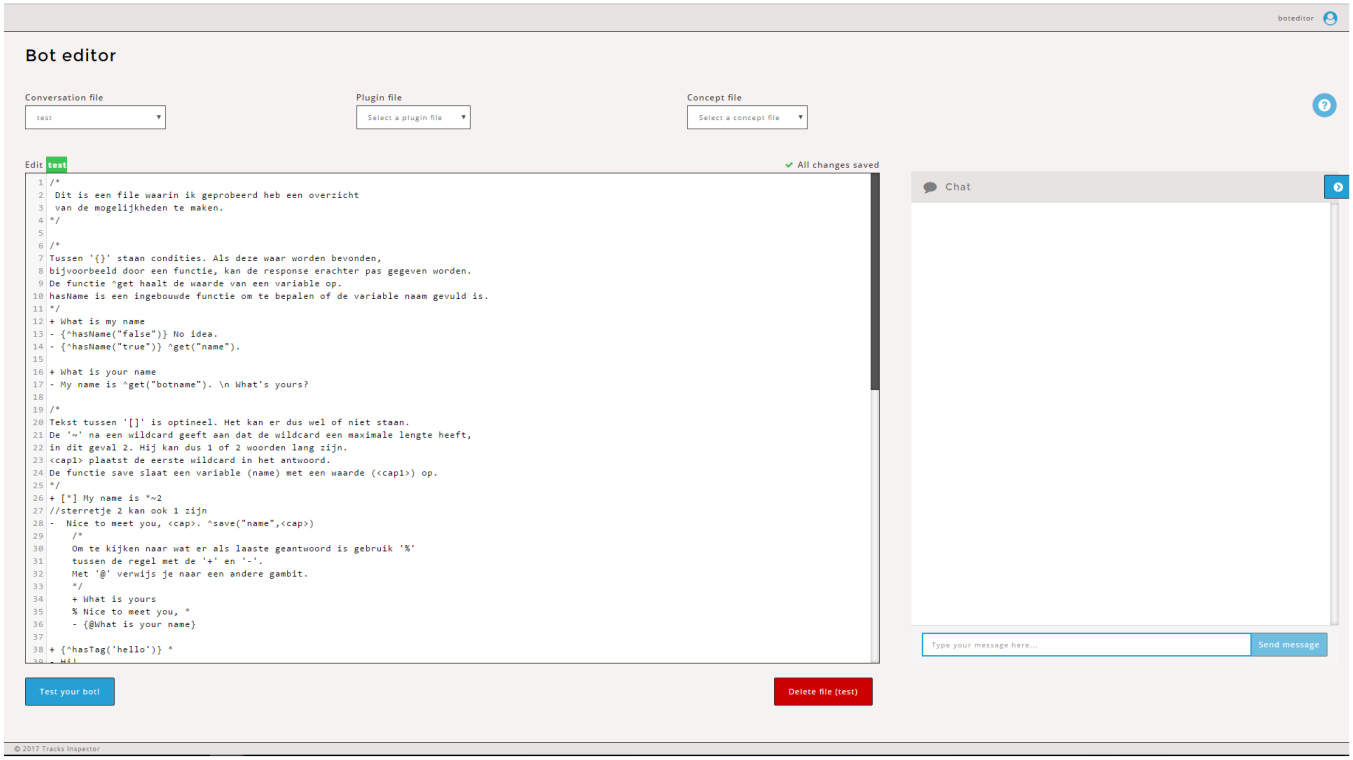


Figure 4.1: Screen shot of script file in the test environment

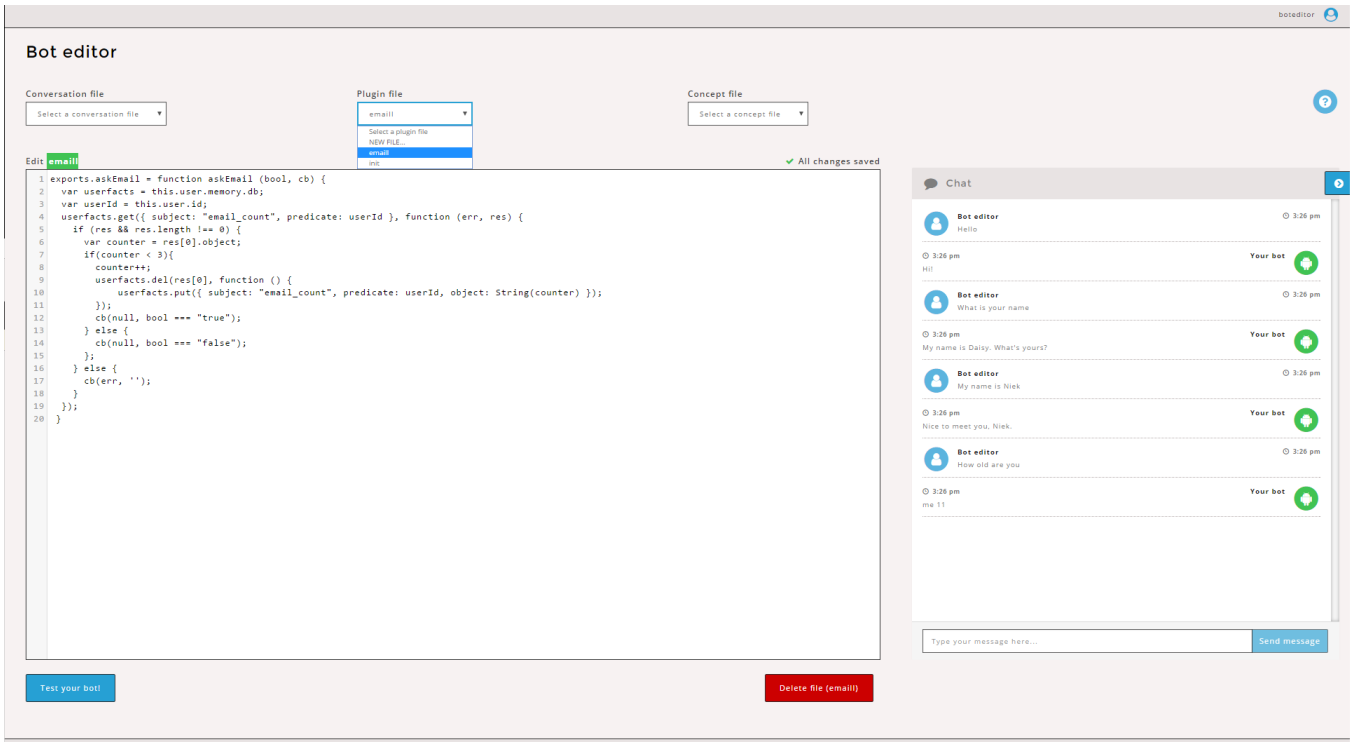


Figure 4.2: Screen shot of a plug-in and conversation in the test environment

4.2 Plug-ins

In order to extend the basic functionality of SuperScript, it is possible to create custom functions in JavaScript and use them in the SuperScript chat bot. In this section, a few of the plug-ins are discussed.

4.2.1 checkEmail

In a regular conversation SuperScript responses are either discards or always kept. Because of the purpose of the chat bot, asking for contact information, e.g. an email address, should be done once or a couple of times. This means the output that asks for the email address, should not be discarded. But to make the chat bot more human-like, it should not repeat itself endlessly, which means the output cannot be kept forever.
To deal with this problem, the plug-in askEmail was created. Every time the chat bot ask for an email-address, this plug-in should be called. It increments a variable created at initialization of the chat bot that represents how many times the email-address of the user has been asked. If this variable exceeds a certain value, the chat bot will no longer use the output that ask for an email-address. This plug-in can easily be transformed into a check if another output should still be used or not. Email is just an example. The function is included in Appendix A.

4.2.2 ageCheck

When the chat bot asks a user for their ASL (age, sex, location), a number is expected in the reply. If the age of the user is to be saved, it needs to be an actual number, within certain boundaries.
To check if this is the case, the plug-in ageCheck was created. It uses another plug-in to check for numbers in the users input. The last message of the user is checked for numbers. If it contains a number between 0 and 100, the age is saved in the variable ”age”. Then the maturity of the user is determined by evaluating if the number is lower than 18. The response of the chat bot is determined by this evaluation. If no number was detected, a different response is required than for the other options. The function is included in Appendix B.

4.3 Concepts

SuperScript also supports custom concepts. These concept expand a word with synonyms or words that are interchangeable in a certain context. A good example is the custom confirm concept. In this concept, a lot of confirmatory answers are grouped under the tag ∼confirm. The bot replaces it with one of the choices in its output and matches all choices in its input. Here the example:

```
concept: ~confirm(yes yess yep yup sure jes jez certainly si cool ok oke okidoki
okay true ”yes yes” ofcourse ”no prob” yea ”no problem” ”of course” yeah)
```

Chapter 5

Issues around dialogue modelling

While the editor was being used, SuperScript occasionally displayed unusual or unexpected behaviour. These occurrences and their solutions, if there were any, will be discussed in this chapter.

5.1 Discarding of responses

SuperScript has the build-in functionality of discarding responses after they are used. This is done by looking at the chat history with the user. However, by default only the last ten responses are considered. This means that after ten messages the bot may repeat itself. Because this was a problem with the old bot engine, repetition has to be prevented. Fortunately, there is a build option to change the number of considered replies from the history. By chancing this to a higher number than ten, one hundred was chosen for now, this issue is resolved.

5.2 Group of words in a concept

Along with creating the confirm concept seen before, the issue arose that a group of words cannot be separated with a space, because that is how single words are separated.
After consulting the developers, it became clear that words can be grouped using double quotes around the group. Again, the confirm concept as example:

```
concept: ~confirm(yes yess yep yup sure jes jez certainly si cool ok oke okidoki
okay true "yes yes" ofcourse "no prob" yea "no problem" "of course" yeah)
```

5.3 Concepts in conflict with WordNetDB expansion

In gambits and responses, words can be expanded using an ∼. Without custom concepts, SuperScript tries to expand a word using the WordNet database [1]. Concepts with the same name however should be preferred over WordNet. For example, a concept ”sports” is defined, which only expands the word ”baseball”. Now see this short script example:

```
+ do you like ~sport
- No i do ~sport
```

This results in the following behaviour:

User: do you like baseball
Bot: No i do rock climbing
User: do you like rock climbing
Bot: I don’t understand.

This means that the custom concept is only used on user input. When expanding a word in a response, SuperScript will expand it through WordNet. This results in unpredictable responses, something that is not desired, because the behaviour of WordNet is unknown. This problem has been reported to the developers of SuperScript and they consider this a bug.
A workaround is simply not expanding word in output of the chat bot, but this feature is useful to make the chat bot seem more human. It helps with creating less generic responses, by expanding the word choice in the output.

5.4 Alphabetical order of topic files

In the extension of the criterion of saving the conversation history to check which input is a no match, a topic ’unmatched’ was added in the test environment. This topic matched on all input. But because of the way I presumed SuperScript chooses which gambit matches best, is never chosen, unless all other gambits do not match. Unfortunately, when a topic that would be after unmatched in alphabetical order, would never be chosen. That means that it is very dangerous to put a gambit in a general topic that matches everything, because better matches in topics afterwards will not be considered. To prevent this, but still see what input did not got matched, the gambit that matches everything was relocated to a special topic, the ’post’ topic. This topic will always be considered last.

5.5 Return from a layered conversation

SuperScript offers a way to layer a conversation in the way that a response is only given, when the last output matches a certain pattern. This is done with a line starting with an % between the expected input and the reply. An example from the SuperScript documentation illustrates this well.

```
+ conversation
- What is your name?

    + [my name is] *1
    % * what is your name
    - So your first name is <cap1>?

        + ~confirm
        % so your first name is *
        - Okay good.

        + *
        % so your first name is *
        - Oh, lets try this again... {@conversation}

    + *
    % * what is your name
    I don't get it. {@conversation}
```

When the chat bot asks the question "What is your name?" it expects the user to give their name. To be sure, the chat bot ask the user to confirm his or her name. The next two gambits only trigger when the last output of the chat bot was the question to confirm the user's name. The last gambit only triggers if the user does not enter something that looks like a name after the first question. When the output "Okay good." is generated, the end of the layered conversation is reached. But any input the user will now give will still be matched against the bottom layer of this conversation, unless redirected as in the other gambits.
To jump out of layered instruction without redirecting, {clear} should be added to the response. I had to ask the developers about this, as it was not in the documentation.

5.6 Splitting message into words

After the creation of the checkAge plug-in, there were occurrences of unpredicted behaviour with longer input without spaces. The plug-in works fine for smaller words with numbers in them, but not for longer ones. This

is because SuperScript splits words with numbers up in separate words. For example, ”male15” becomes ”male 15”. That is why the plug-in can check if the input contains a word that is a number. However, when a word exceeds seven characters, e.g. 12malemanila the system no longer divides the input into different words, preventing my plug-in from working properly.
This behaviour is caused by the parse library used in SuperScript. It prioritises splitting strings on IDs, which are words with numbers or letters or a combination of the two and have a length longer than eight characters, over a single occurrence of a number. However, the ’raw’ input, the exact input string, is also available in the plug-ins. Numbers are now isolated from the raw input, instead of the word attribute.

5.7 MongoDB

As a back-up storage SuperScript uses a Mongo database. MongoDB [12] is a NoSQL-database in which data is stored as binary JSON documents. The gambits, replies, topic and user data is stored in separate collections. To fully understand some behaviour of SuperScript, and to view the chat history, I had to learn to use MongoDB to retrieve stored data.

5.8 Commands

Besides input of the users, there are several commands the Sweetie system uses to communicate with chat bot. A distinction can be made between input and output of the chat bot. Examples of input commands are webcam request and an validation of an email address, and examples of output commands are ending the conversation and opening a webcam stream. Input is either a command or a conversation line. Output always contains an conversation line, however this can be an empty string. It can also contain an optional command, see Figure 5.1. SuperScript only expects input and also sends its output, in text format. It therefore cannot distinguish the commands from regular user input.
To solve this, commands to and from the bot are encoded as predefined strings that should never occur in a normal chat session. Command can now be used in triggers and gambits.

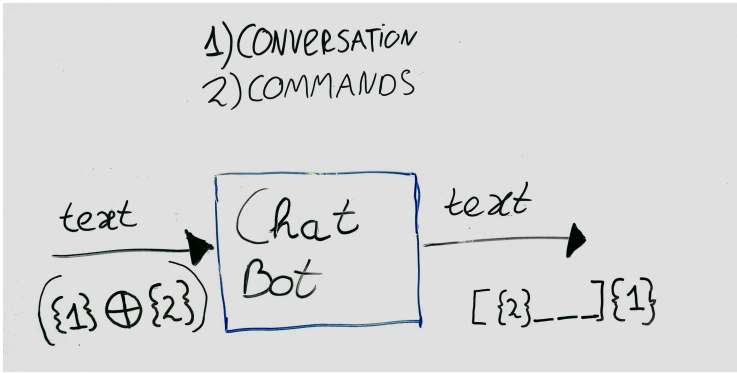


Figure 5.1: Possible input and output for the chat bot

Chapter 6

Discussion

When researching possible options for the replacement of the chat bot, only relative popular and big chat engines, have been considered. There may be more suitable options out there, that remain unconsidered. The up-to-date requirement could be a big part of the cause of this, as it requires an active community, which is more common with popular engines.

The most important reason to choose SuperScript over ChatScript, has been the possibility to enrich the conversation with the functionality of self-created functions. This however could possibly be in contradiction with the requirement that the modelling of the conversation could be done by non-technical operators. I believe the right decision was made, as these function can be created almost separately from the scripts, and are usually not very complex. But this does mean there should be someone who can create and maintain these functions and help the operators use them. This can simply be done by training one operator or for example hiring a programmer. I think custom functions are very valuable for this project, and might be the only way to reach all goals of the chat bot.

One of the considered chat bot engines, ChatterBot, is a self-learning chat bot. Due to the absence of sufficient training data, ChatterBot was not deemed suitable for the Sweetie 2.0 system. However, when the SuperScript will be deployed, it will save the conversations. These conversations could be used as training data, to create a self-learning chat bot for the Sweetie 2.0 system, by adding a dedicated module to it.

Chapter 7

Conclusions

It is too early to say that SuperScript completely covers all shortcomings of the commercial chat bot engine. Further implementation of conversation of the chat bot has to prove if SuperScript was the right choice, but it is clear that it has a lot of potential. It is a lot more flexible and the structure of conversation is clearer. Preliminary use showed that the script language is quite easy to understand.
However, the people creating the conversation find it difficult to work with the plug-ins as they have no programming experience. But the plug-ins are almost completely abstracted from the actual conversation, only the calls to the plug-ins are in the scripts. Because of this, it would suffice to appoint a single programmer for support.
The full potential of SuperScript can be used in the Sweetie 2.0 system. Operators can create new conversations script and change existing ones, in new or already created chat bot instances. The basis of the SuperScript chat bot has been finished, and now the conversation line can be implemented and further optimized.

Bibliography

[1] https://wordnet.princeton.edu
Official site of the WordNet database, 19-05-17.

[2] www.terredeshommes.nl/en/programmes/sweetie-20-stop-webcam-child-sex
Official site of the Sweetie Project, 15-05-17.

[3] From child pornography offending to child sexual abuse: A review of child pornography offender characteristics and risks for cross-over. Aggression and Violent Behavior, 19, 466-473.
Houtepen, J.A.B.M., Sijtsema, J.J. & Bogaerts, S. (2014),

[4] Software Engineering: Principles and Practice, third edition, p. 63
Hans van Vliet, Wiley, Chichester (UK), 2008

[5] https://www.rivescript.com/
Official site of RiveScript, 22-06-17

[6] https://chatterbot.readthedocs.io/en/stable/
Official site of ChatterBot, 22-06-17

[7] chatscript.sourceforge.net
Official site of ChatScript, 22-06-17

[8] http://www.program-o.com/
Official site of Program-o, 22-06-17

[9] http://superscriptjs.com
Official site of the SuperScript, 22-05-17

[10] https://opensource.org/licenses/MIT
Full MIT licence, 26-06-17

[11] https://opensource.org/licenses/BSD-3-Clause
Full BSD-3-Clause, 26-06-17

[12] https://www.mongodb.com
Offical site of MongoDB, 19-05-17

Appendices

Appendix A

Plug-in askEmail

```
exports.askEmail = function askEmail (bool, cb) {
  var userfacts = this.user.memory.db;
  var userId = this.user.id;
  userfacts.get({ subject: "email_count", predicate: userId }, function (err, res) {
    if (res && res.length !== 0) {
      var counter = res[0].object;
      if(counter < 2){
        counter++;
        userfacts.del(res[0], function () {
                userfacts.put({ subject: "email_count", predicate: userId, object: String(
        });
        cb(null, bool === "true");
      } else {
        cb(null, bool === "false");
      };
    } else {
      cb(err, '');
    }
  });
}
```

Appendix B

Plug-in checkAge

```
exports.ageCheck = function ageCheck (age, cb) {
  var userfacts = this.user.memory.db;
  var userId = this.user.id;
  var message = this.message;
  var num = hasNumber(message);
  if(num !== null && num < 100) {
    userfacts.del(num, function () {
        userfacts.put({ subject: "age", predicate: userId, object: String(num) });
    });
    if (num < 18) {
      userfacts.put({ subject: "mature", predicate: userId, object: false })
      cb(null, age === "young");
    }
    else {
      userfacts.put({ subject: "mature", predicate: userId, object: true })
      cb(null, age === "mature");
    }
  }
  else {
    cb(null, age === "nan");
  }
}
```