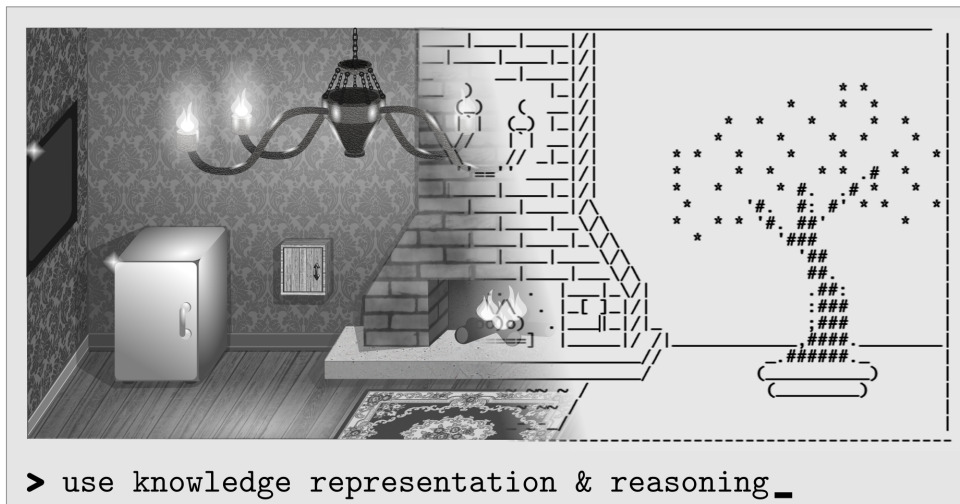


Miloš Stanojević

Interactive Fiction



Computer Science Tripos – Part II

Trinity College

May 18, 2017

Interactive Fiction is a genre of computer games with an imaginary world, which engages the player's wits while they solve puzzles, as they make their way towards the conclusion of the story.

This project aims to improve the way Interactive Fiction is developed and perceived, by introducing Knowledge Representation and Reasoning at the core of a historical engine's contemporary implementation.

***The cover image** depicts a creative player's interpretation of the rich imaginary world existing within an Interactive Fiction game, as seen through the window of a terminal.*

Proforma

Name: **Miloš Stanojević**
College: **Trinity College**
Project Title: **Interactive Fiction**
Examination: **Computer Science Tripos, Part II, June 2017**
Word Count: **11976¹**
Project Originator: **Dr Sean Holden**
Supervisor: **Dr Sean Holden**

Original Aims of the Project

Improving on the standard Interactive Fiction engine, which allows only for pre-defined responses to user input. This would be done by enhancing it with Knowledge Representation and Reasoning, which would allow for the addition of more informative, automated responses. The development of a Story Editor which would support the creation of game files, runnable by the engines (standard and enhanced). Designing an appropriate story file which would then be used in a comparative user study, to test the success of the improvement made to the standard engine.

Work Completed

The project has been highly successful. All the work outlined in the proposal was carried out and all extensions implemented, in addition to some new ones. I built a standard Interactive Fiction engine and extended it with Knowledge Representation and Reasoning to support four physical parameters (volume, temperature, state and mass). I implemented a custom story language in ANTLR and an appropriate story compiler for it. Finally, I created a custom Interactive Fiction story in the language and used it in a user study on 22 participants, which gave convincing results in favour of the improved engine.

Special Difficulties

None.

¹The word count was computed using T_EXcount, <http://app.uio.no/ifi/texcount/>.

Declaration

I, Miloš Stanojević of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed *Miloš Stanojević*

Date 18 May 2017

Contents

1	Introduction	9
1.1	Background	9
1.2	Motivation	10
1.3	Related Work	11
2	Preparation	13
2.1	Standard IF Engine Analysis	13
2.1.1	User Command Support	13
2.1.2	Story Development	14
2.2	Knowledge Representation and Reasoning	14
2.2.1	Basic Concepts	15
2.2.2	Object-oriented Representation of Knowledge	15
2.2.2.1	Frame Formalism	15
2.2.2.2	Reasoning with Frames	16
2.2.2.3	IF Use Case	17
2.3	NLP Frontend and Story Compiler	17
2.3.1	User Side - NLP Frontend	17
2.3.2	Developer Side - Story Compiler	17
2.4	Planning the User Study	18
2.5	Choice of Tools	18
2.5.1	Libraries	18
2.5.2	Language Choice – Java	19
2.5.3	Development Environments & Tools	19
2.6	Requirements & Software Engineering	20
2.7	Summary	22
3	Implementation	23
3.1	Overview	23
3.2	Engine	25
3.2.1	Standard Engine	26
3.2.2	Enhanced Engine	27
3.2.2.1	Physics Model	27
3.2.2.2	Knowledge Base	29
3.2.2.3	Automated Response Generation	30
3.2.2.4	Inference Loop	31
3.3	NLP Frontend	32

3.3.1	Sentence Analysis	32
3.4	Compiler Backend	34
3.4.1	Story Language	35
3.5	Summary	37
4	Evaluation	39
4.1	Overview	39
4.2	NLP Frontend Testing	39
4.3	Compiler Backend Testing	40
4.4	Engine Evaluation	40
4.4.1	Experiment Set-up	41
4.4.1.1	Questionnaires	41
4.4.1.2	In-game Evaluation – KeySEF	42
4.4.1.3	IF Game File	43
4.4.1.4	Additional Engine Features	43
4.4.2	Results & Analysis	44
4.4.2.1	Hypothesis Testing	44
4.4.2.2	Data Analysis	46
4.5	Summary	50
5	Conclusion	51
5.1	Achievements	51
5.2	Lessons Learnt	52
5.3	Further Work	52
	Bibliography	52
A	Complete Story Grammar	55
B	NLPtest – Example Batch File	61
B.1	Test Input	61
B.2	Expected Output	61
C	Enhanced IF Game Session	63
D	Project Proposal	71

List of Figures and Tables

Figure 1.1	Excerpt from the Hitchhiker’s Guide to the Galaxy IF game.	9
Figure 1.2	The pressure plate example for the primary problem.	11
Figure 2.1	The generic frame definition and example.	16
Table 2.1	Summary of the tools used during development.	19
Table 2.2	Summary of the project deliverables.	20
Figure 2.2	Dependency analysis of the Interactive Fiction project.	21
Figure 3.1	The structure of the Interactive Fiction expert system.	23
Figure 3.2	The sequence diagram showing a complete game execution.	24
Figure 3.3	The class diagram of the complete engine.	25
Figure 3.4	Example of a simple story state machine.	26
Figure 3.5	Simplified view of interactions between the parameters of the model.	28
Figure 3.6	The three classes of the KB, expressed in the frames formalism.	30
Figure 3.7	The procedures of the key-in-lock example, given in execution order.	31
Figure 3.8	The major dependencies of the NLP frontend.	32
Figure 3.9	Two similar example sentences, analysed using the NLP frontend.	33
Figure 3.10	The class diagram of the Compiler backend.	35
Figure 3.11	A small, self-contained story-file example.	36
Figure 3.12	A side-by-side comparison of the two engines.	38
Figure 4.1	A single run of the NLPtest framework.	40
Figure 4.2	A play-test of the final game-file.	40
Figure 4.3	The ordered set of response levels used for the Likert scales.	42
Table 4.1	A list of all the Likert items, with their corresponding metric.	42
Figure 4.4	The configuration of KeySEF for the IF user study.	43
Figure 4.5	The plot of the t-distribution, for 21 degrees of freedom.	45
Figure 4.6	Results obtained from the user study, when a t-test is applied.	46
Figure 4.7	A comparison of the results for all questionnaire metrics.	47
Figure 4.8	A comparison of the results for the three steps, from KeySEF.	48
Figure 4.9	A pie chart showing the demographic components of the cohort.	48
Figure 4.10	The average marks of the engines, depending on player experience.	49
Figure 4.11	The average marks of the engines, depending on session order.	49

Acknowledgements

The work presented in this dissertation, as well as the write-up itself, has been greatly facilitated by the contributions and suggestions of the following people, to whom I owe particular thanks:

- **Dr Sean Holden**, for giving valuable advice and being an outstanding mentor for the project as a whole, which lead to the successful completion of the work;
- **Andrej Ivašković** and **Gábor Szarka**, for their efforts and dedication as participants in the many iterations of User study refinement;
- **Petar Veličković**, for providing helpful advice, hints and tips in each stage of the production of this write-up;
- **My family and friends**, for all their support, understanding and enduring the many hours of technical discussion, on a daily basis.

Chapter 1

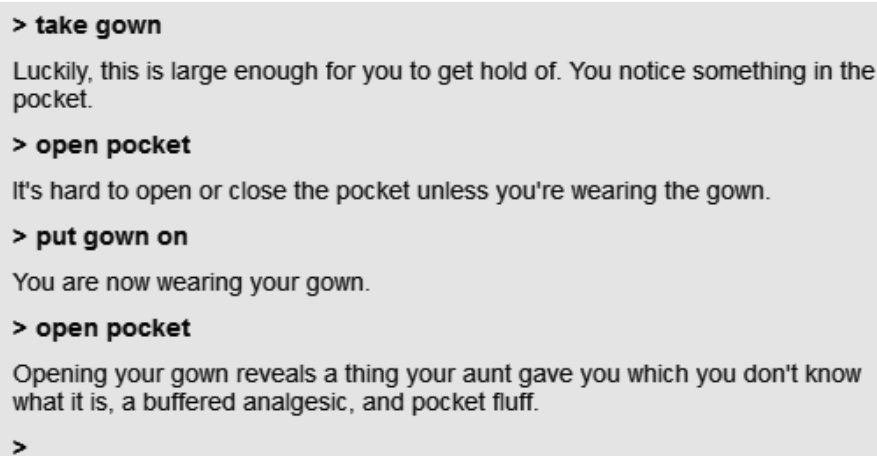
Introduction

The main focus of my CST Part II project has been to investigate the potential for improving the user experience of Interactive Fiction (IF) games by implementing Knowledge Representation and Reasoning (KR&R) at the core of the engine. In order to achieve this, a feature-rich Interactive Fiction engine has been fully implemented and evaluated using a formal user study.

In this chapter, I will discuss the primary motivation behind the need for such an improvement, present a case in favour of KR&R as the potential solution, and give a survey of related work.

1.1 Background

At the time when textual input was the only interface available to game developers, text-based adventure games were at the peak of their popularity. A particularly interesting category of such games was Interactive Fiction – a type of interactive story where players are presented with interesting environments which they could then interact with and influence in fun ways to progress through the story-line.



```
> take gown
Luckily, this is large enough for you to get hold of. You notice something in the
pocket.
> open pocket
It's hard to open or close the pocket unless you're wearing the gown.
> put gown on
You are now wearing your gown.
> open pocket
Opening your gown reveals a thing your aunt gave you which you don't know
what it is, a buffered analgesic, and pocket fluff.
>
```

Figure 1.1: Excerpt from the Hitchhiker’s Guide to the Galaxy Interactive Fiction game published by Infocom in 1984.

This sort of interaction usually includes manipulating items present in the environment and utilising or combining them in creative ways. An example of this can be seen in Figure 1.1, which was taken from the Hitchhiker’s Guide to the Galaxy [10].

Other examples of this genre include *Dunnet* (Ron Schnell 1982), *Planetfall* (Infocom 1983), *Trinity* (Infocom 1986) and so on. Due to their level of immersion and enticing puzzles, Interactive Fiction games have a notable following even today.

1.2 Motivation

In an ideal world, Interactive Fiction games would not have to be programmed, but would rather need simple guidance, as an artificial intelligence back-end would be able to take any human input and advance the story in an intelligent way. As the progress towards general AI is still in its very early stages [18, 7], the current state-of-the-art in IF games is far from this ideal.

Historically, there have been many attempts to design a universal Interactive Fiction story language, increasing in quality over time. Older examples include the Text Adventure Development System (TADS) [17] originally released in 1988. This early language was strongly typed and heavily based on Pascal and C. On the other hand, most of the modern day IF games are written in Inform 7 [14], which is a declarative programming language supporting rule-based relations.

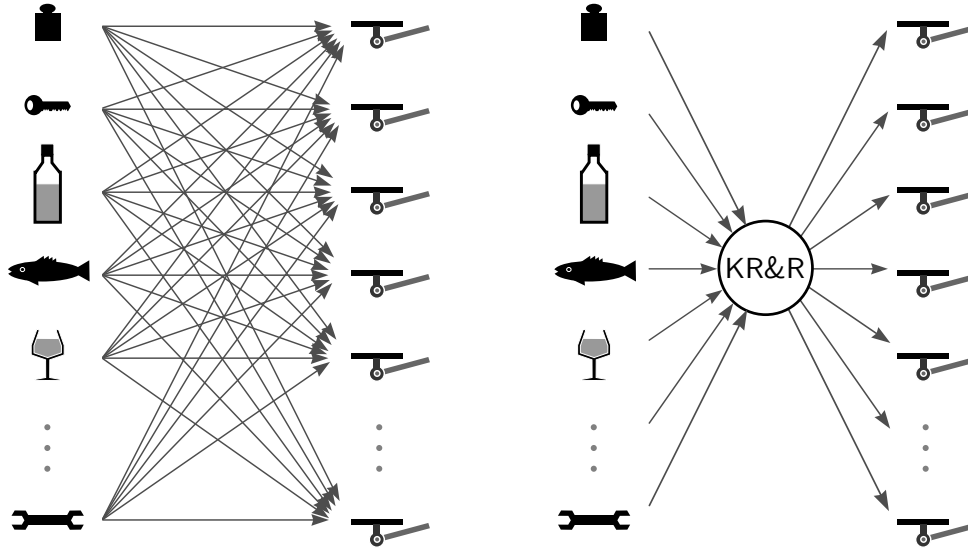
While this setting appears promising, for one to make a high-quality IF game in either of these languages, one needs to write out not only the entire story but also all of the many trivial interactions between various items. This is especially problematic for all the cases when an action cannot be performed.

An example would be if we had five or six items which all need to be placed on correct pressure-plates for a door to open (see Figure 1.2). In this case, there is only one correct positioning of the items, and all the other ones are incorrect. It would be simple enough to just provide a generic “I am unable to do that” or “I do not understand” response, but this makes for a frustrating user experience. On the other hand, if we wanted to give informative responses for every valid positioning as in Figure 1.2a, we would have to invest a disproportionate amount of effort. This approach is not only error prone but tedious for the game-developer and may lead to unclear game-play.

All of this stems from the simple nature of both initial and modern day IF engines, as they essentially implement a state machine of story steps¹. Of course, movement between rooms is provided as a separate mechanic, but apart from that, everything is up to the developer – tasks such as setting the contents of the rooms, how they are connected, what actions are triggered by each story step and so on.

This project concerns the application of KR&R, to allow the generation of more informative automated responses, in an attempt to improve user experience in IF. This is done with minimal effort on the part of the game developer, and the example seen in Figure 1.2b furthers this claim. More precisely, I use knowledge representation for the state of the game-world at every time-step, and reasoning is then used whenever the story cannot

¹Described in detail in §2.1.2.



(a) In the simple case, responses for all possible item-plate combinations need to be implemented, resulting in a necessary time effort complexity of $O(N^2)$.

(b) With KR&R at the core of the engine, the effort needed is $O(N)$, corresponding to the time it takes to initialise the relevant parameters for all N items.

Figure 1.2: The pressure plate example for the primary problem. There are N items that need to be positioned on N pressure plates, and there is only one correct ordering of the items on the plates.

be advanced in pre-programmed ways, thus producing more creative automated responses than a simple “I do not understand”. An example of this would be if we had a freezer container and a large elephant in a room and the player inputs “Put the elephant in the freezer.”. Here the KR&R extended engine would be aware of the capacity property of the freezer and the volume of the elephant, so after applying reasoning, it could respond with “The elephant is too big to fit in the freezer.”. This would be much more insightful for the player than a dull “I do not understand”.

1.3 Related Work

A notable paper, which formalises IF, is “Toward a Theory of Interactive Fiction” [13] first published by Nick Montfort in 2002 and most recently revised in 2011. According to this paper, IF is defined as:

- a text-accepting, text-generating computer program;
- a potential narrative, that is, a system which produces narrative during interaction;
- a simulation of an environment or world; and
- a structure of rules within which an outcome is sought, also known as a game.

These elements are important in understanding the design of a standard IF engine because their physical manifestations can be treated as distinct parts which together form the engine. This is discussed further in §3.

The paper also discusses some of the core concepts of an Interactive Fiction game, such as the idea of a *session* being a single play-through of a single player on an IF engine. This is particularly interesting for the evaluation of an IF engine (§4), as it gives a baseline for comparing player experience in different sessions.

Chapter 2

Preparation

In this chapter, I will discuss the necessary preparatory work which I performed before work on the project could commence. First, I will discuss the specification of an Interactive Fiction engine considered standard. I will then give an overview of the theory of Knowledge Representation and Reasoning necessary for the scope of the project, and provide brief consideration of the remaining components of the system. I will also consider the elements needed for a successful evaluation via a controlled experiment, as well as the choice of tools used in the development. Finally, I will present the full requirements analysis of the Interactive Fiction project as well as some of the software engineering techniques employed during development.

2.1 Standard IF Engine Analysis

The first step in determining the requirements was analysing an existing Interactive Fiction engine and deciding how to replicate its behaviour. I have chosen the 30th anniversary edition of the Hitchhiker's Guide to the Galaxy for this purpose. The main reason behind this choice is that its engine was generally considered by far the best available at the time, and because it was also recently refurbished by the BBC [3]. Unfortunately, due to this being a licensed title, I could not obtain its source code directly. So in an attempt to model the behaviour of the engine, a significant amount of time was invested into re-playing the game and analysing its responses to carefully designed inputs. The features I deduced in this analysis are described below.

2.1.1 User Command Support

The engine was capable of distinguishing three categories of commands, these being commands with:

- **no item arguments** – these commands have no object of interaction and are divided into two categories:
 - **regular** – commands consisting of a single verb or noun. Examples: look, inventory, wait and so on.

- **special** – hard-coded fixed commands unique to the story, usually consisting of a custom syntagm and sensitive to the word order. Example: “Ford, WHAT ABOUT MY HOME!?”.
- **single argument** – in this case, there is a single object or direction the command is enacted upon. Examples: use [item], take [item], examine [item], move [direction] and so on.
- **two item arguments** – these commands have two objects interacting in some way, and they can further be divided into:
 - **commutative** – two argument commands whose arguments can be exchanged with the command retaining meaning. Examples: combine [item1] with [item2].
 - **non-commutative** – two argument commands whose meaning changes if the arguments exchange places. Example: place [item1] on [item2].

For all of these commands, a small set of hard-coded synonyms was also recognised.

2.1.2 Story Development

The story of the Hitchhiker’s Guide to the Galaxy consists of linear sub-stories combined in a semi-non-linear way. By this, I mean that there are parts of the story where branching into several sub-stories occurs, and the player must choose one path to follow. These branches later lead to the same final sub-story, which is not influenced by the order or decisions made earlier. Taking all of this into account, the conclusion made about the underlying engine is that it keeps the following state:

- **a state machine of story steps**¹ – having a starting step with no parents, preconditions or effects, intermediate steps which advance the story, and termination either by an action that kills the player or one that grants him victory.
- **a room map** – containing all the information about how rooms are connected and which items each room contains.
- **the player’s inventory** – being the set of items held by the player.

2.2 Knowledge Representation and Reasoning

After considering the basic features of a standard IF engine a good deal of research needed to be conducted on the extension. Having gone through the Part IB Artificial Intelligence I course, the core concepts of KR&R were familiar to me. However, a decision needed to be made on which style of inference should be employed. This section is based on the theory presented in “Knowledge Representation and Reasoning” [2], which was an invaluable source of information during this stage of research.

¹A story step is a set of parent steps, preconditions (e.g. player in room X, item Y in inventory, etc.) and effects (e.g. kill player, remove item Y from game, etc.). It is triggered when either all or any one of its parent steps have triggered (depending on the setting) and once all its preconditions are satisfied, which is when its effects are applied to the environment.

2.2.1 Basic Concepts

When thinking about KR&R, in the context of IF, the first thing that needs clarification are its three defining words:

- **Knowledge** – can be regarded as a set of known facts – propositions which are either universally true or false within the current game-world. For example, propositions such as “the fridge is cold”, or the “ice is solid”.
- **Representation** – describes the way in which factual propositions of knowledge are mapped to sets of *formal symbols* – strings representing elements of the game-world. For example, the formal symbol “elephant” would represent the notion of an elephant in this context.
- **Reasoning** – defines the formal manipulation of the symbols, representing the collection of factual propositions, to produce representations of new ones.

Having considered the core concepts of KR&R, I will outline the ways of representing knowledge and reasoning with it.

2.2.2 Object-oriented Representation of Knowledge

The standard way of doing KR&R, taught in AI I as well as discussed in detail in the KR&R book, is first-order logic using Horn clauses (also the underpinnings of Prolog). However, one property of particular interest with this representation method is that it is flat: each piece of representation is self-contained and can be understood independently of any other. So, with this approach, knowledge about a given object or object type could be scattered around the knowledge base requiring us to do general inference on the entire knowledge base. This can be costly in terms of time, and in the context of an interactive computer game, can be bothersome for the player. For these reasons a more representationally motivated approach is necessary. Thus, instead of first-order logic, I considered an object-oriented representation of knowledge.

In one of the more seminal papers [11] in the history of KR&R, Marvin Minsky suggested the idea of using object-oriented groups of procedures to recognise and deal with new situations. Minsky defines the term *frame* for the primary data structure used to represent such groups. Much work has been done since then in creating a formal representation language of frames, and I present one [2] here.

2.2.2.1 Frame Formalism

From an implementation standpoint, a frame is a named list of *slots*, each of which can be assigned a *filler*. Slots can be visualised as “buckets” and fillers as items which occupy them. There are two main frame types: *individual frames*, used to represent single objects, and *generic frames*, used to represent object classes. For the purposes of this dissertation, I will consider only **generic frames**, which are semantically defined in Figure 2.1a.

Notationally, the names of generic frames are taken as capitalised, while the names of individual ones are uncapitalised mixed case. Here, slot fillers cannot be atomic, and

<i>(Frame-name</i>	<i>(Item</i>
<i><:IS-A filler0></i>	<i><:IS-A Object></i>
<i><:Slot-name1 filler1></i>	<i><:Location ItemLocation></i>
<i><:Slot-name2 filler2></i>	<i><:State [IF-NEEDED ComputeState]></i>
<i>...)</i>	<i>...)</i>

(a) The definition of a generic frame.

(b) An example of a generic frame.

Figure 2.1: The formal definition of a generic frame, and an example of its use in the context of IF.

either refer to other generic or individual frames. Generic frames also have a special *IS-A* slot, whose filler is a more general generic frame.

Slots of generic frames can also have **attached procedures**². These procedures can be one of two main types:

- *IF-NEEDED* – the procedure is called only if the value is requested. This is similar to lazy evaluation seen in functional programming.
- *IF-ADDED* – this procedure is called immediately if a value is added to the relevant slot.

Taking this into account, an example of a generic frame is given in Figure 2.1b.

2.2.2.2 Reasoning with Frames

The aforementioned procedures allow the frames framework to be flexible and organised for computation. Highlighting these benefits more specifically, a skeleton of a reasoning loop in such a system would consist of the following steps:

1. a user instantiating some generic frame, declaring to the knowledge system that a situation exists;
2. any slot fillers which can be inherited by the new frame, and which are not provided explicitly, are inherited;
3. for each slot filler, any existing *IF-ADDED* procedures which can be inherited are run, potentially causing new slots to fill, or new frames to be instantiated;
4. cycle from the start.

On the other hand, if a user or an attached procedure requires the filler of a slot, the following behaviour is observed:

1. if the slot has a filler attached to it, the filler is returned;
2. otherwise, any *IF-NEEDED* procedure that can be inherited is run, thus determining the filler for the slot, and potentially also causing other slots to be filled, or new frames instantiated.

²A single slot can have both a filler and a procedure attached to it.

2.2.2.3 IF Use Case

Overall, we can think of a frame knowledge base as a symbolic “spreadsheet” where the constraints between relevant objects are propagated via invocation of attached procedures. Due to their object-oriented nature, such knowledge bases perfectly lend themselves for implementation in a language such as Java or C++. Finally, the data stored within them can be connected in elaborate ways so that conclusions can be made and queries executed in a swift fashion. This is what makes the object-oriented way of representing knowledge adequate for the IF use case, where the emphasis is on the interactivity of the system and where any unnecessary delays may directly impact user experience. A detailed account of how KR&R is applied in this context is given in §3.

2.3 NLP Frontend and Story Compiler

The final piece of the puzzle was to determine the way users and the story developer interact with the engine. Taking into account all that I have found in §2.1 the following conclusions could be made:

2.3.1 User Side - NLP Frontend

Due to technical limitations at the time, original IF games were only able to recognise very simple textual input. Fortunately, there are many powerful natural language processing (NLP) tools currently available, tools which could be employed to facilitate considerable improvement at the frontend of an IF game. This is why I have chosen to employ one such tool, CoreNLP (§2.5.1), so that the user interface could be as flexible as possible, and accept a wide range of inputs. This primarily includes support for:

- **natural language input** – the sentences input by the user are to be correct sentences of the English language.
- **synonyms** – each command should allow for as many synonyms as it is reasonably possible.
- **context sensitive item references** – the engine should equivalently accept “the green car” as well as “car” if the word “car” is non-ambiguous in the context of the current room.

2.3.2 Developer Side - Story Compiler

Due to Inform 7 being self-contained and impossible to integrate with a custom engine, I had to move away from using it for the implementation of the story development language. Additionally, since the most recent iteration of this language has become close to “reading as English”, it was too complex to create a custom parser for it. For these reasons, I designed a custom language which has all the necessary constructs to model an IF game in the standard sense. The syntax and semantics of this language are presented in §3.4.1.

2.4 Planning the User Study

To be able to formally evaluate the success of the KR&R enhancements planned for the standard IF engine, I chose to design a controlled experiment for technology evaluation – the user study. Knowing that all of the main development needed to be completed first, the actual planning and creation of the user study were left for the final stages of the project. However, it is important to note that the two core steps, outlined in the project proposal, were maintained in each development iteration of the study. These steps are:

1. Participants are given two games with the same script to play, one executing on the Standard IF engine, and one executing on the Standard engine with the KR&R enhancements incorporated (Enhanced engine).
2. They would then be asked to complete a questionnaire comparing their experiences through a series of Likert Items [8].

Finally, a detailed account of the user study and its components is given in §4.

2.5 Choice of Tools

With all the requirements in mind, the next step was to determine all the tools necessary for their successful implementation.

2.5.1 Libraries

The following libraries³ were used during development:

- **CoreNLP** – Since the frontend needs to support natural language input, Stanford’s CoreNLP suite [9] was chosen to facilitate this, due to its power and reliability, and due to it being extensively covered in the Part II NLP course.
- **WordNet** – To support the second requirement of the engine’s frontend – synonyms, the Princeton WordNet [5] library was used. This was done by extracting the relevant library entries in a convenient format and storing them within the engine itself.
- **ANTLR** – On the backend side, the custom story language was intended to be written as a grammar. ANOther Tool for Language Recognition (ANTLR) [16], a powerful parser generator, was selected to ease the implementation of the compiler for this language.
- **JLine3** – Because of Java’s innate lack of raw console editing support, JLine [15] was used to obtain an interface to the Unix and Windows based consoles.
- **Paper&Pencil** – To reduce the amount of effort needed to develop formatted questionnaires in L^AT_EX the Paper&Pencil [4] package was used.

³Only CoreNLP and WordNet were included in the original proposal.

2.5.2 Language Choice – Java

For the implementation language, Java, Prolog and C++ were considered. All of these languages have their benefits and drawbacks, and after considerable thought, I believed Java to be the best tool for the job. The reasoning was as follows:

- Both ANTLR and CoreNLP are primarily Java based, making Java more favourable over C++, for which custom interfaces would be necessary.
- The object-oriented model of reasoning fits perfectly in an OOP context, which was a crucial reason for choosing Java over Prolog.

The only downside of this choice though is the fact that Java does not support direct, raw console character manipulation. Fortunately, this was easy to mitigate with the use of JLine.

2.5.3 Development Environments & Tools

All the relevant software development and dissertation write-up were performed on my personal machine (Asus, 2.40GHz Intel(R) Core(TM) i7-4700HQ CPU, 16.0GB RAM and 500GB Disk, Microsoft Windows 10). Additionally, ten MCS machines were used for the execution of the User Study – guest accounts for which were kindly provided by the University Computing Service. Finally, the complete listing of the remaining tools and their respective applications is given in Table 2.1.

Tool	Purpose
IntelliJ IDEA Community Edition 2016.3.3	IDE
Texmaker 4.5	L ^A T _E X editor
Notepad 6.9.2	Text editor
Microsoft Visio 2013	Vector graphics editor
Xara Designer Pro 11	Image editor
Photoshop CS4	Image editor
Microsoft Excel 2013	Spreadsheet editor
javac 1.8.0_71	Java compiler
git 2.6.2	Revision control
Acronis True Image 2016	Backup tool

Table 2.1: Summary of the tools used during development.

2.6 Requirements & Software Engineering

Motivated by the arguments given earlier, and looking ahead to the evaluation phase of the project, I defined the **primary problem statement** as follows:

Given an Interactive Fiction engine with a set of representative defining features, is it possible to improve its user experience on the scales of helpfulness, guidance, clarity and overall ease of use, by utilising Knowledge Representation and Reasoning at its core?

With this in mind, I was able to perform a complete requirements analysis, and distinguish between the necessary project deliverables, which are presented in Table 2.2. Additionally, each deliverable was assigned an appropriate priority level from the following set:

- **L0** – level zero, signifying the core development of the project;
- **L1** – level one, representing highly desirable extensions;
- **L2** – level two, signifying desirable extensions.

While most of these deliverables are mentioned in the project proposal, there are some (mostly regarding the controlled experiment) which are featured for the first time. It is also important to note the **starting point** of the project – only the libraries listed in §2.5.1 were available before the start of development.

Deliverable	Difficulty	Priority
Standard IF engine implementation	High	P0 L0
KR&R extended engine property (Volume)	High	P0 L0
NLP frontend implementation	Medium	P0 L0
Story Compiler backend implementation	High	P0 L0
NLP frontend test suite	Low	P0 L0
Story Compiler backend test suite	Medium	P0 L0
KR&R extended engine property (Mass)	Medium	P0 L1
Story Compiler extra features	High	P0 L2
KR&R extended engine properties (Temperature and State)	High	P0 L2
User Study questionnaire	High	P1 L0
Experiment story file	High	P1 L0
Study harness	Low	P1 L0
User Study additional documents	Medium	P1 L1
IF engine user experience features	Medium	P1 L2

Table 2.2: Summary of the project deliverables.

In addition to this, a full dependency analysis (given in Figure 2.2) was performed on the deliverables. This allowed me to divide the project into two phases:

- **P0** – phase zero, where the core implementation would happen;
- **P1** – phase one, where the development of the user study would be executed.

Such a division was possible primarily due to the user study being heavily dependent on the implementation of the four main elements of the engine – the NLP frontend, Standard IF engine, KR&R extensions and Compiler backend. Moreover, the dependencies within each phase were minimal, so most of the deliverables could be developed independently and in any order.

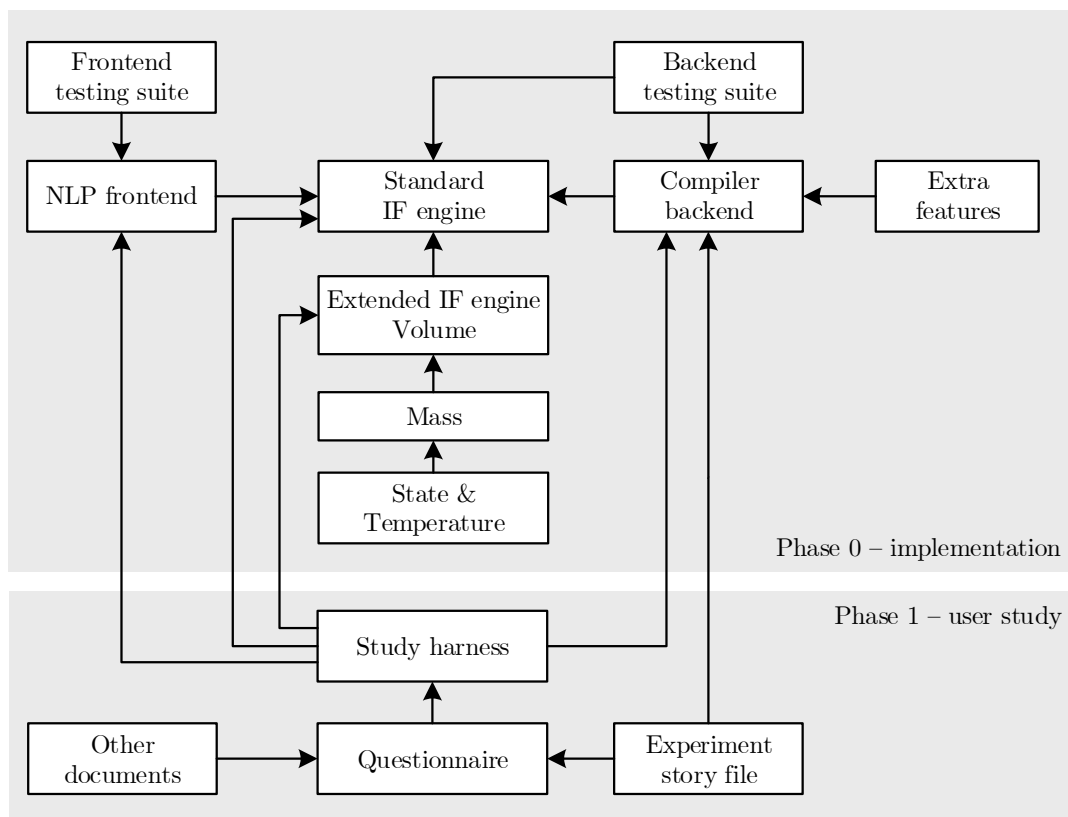


Figure 2.2: Dependency analysis of the Interactive Fiction project. The arrow indicates the direction of dependence ($A \rightarrow B$ means that A depends on B).

Finally, after reviewing a set of applicable software engineering approaches [12], I decided to implement the project according to the **iterative development model**. This model is the predecessor of the spiral model, and the main difference between them is that the iterative development model is less risk driven than its successor. Additionally, this model choice allowed me to have functioning deliverables at multiple stages of development, exploiting the modular nature of the design proposed above.

2.7 Summary

In this chapter, I have summarised the necessary preparatory work that was performed before the implementation phase of the project could begin. I have provided a specification of a Standard engine, given an overview of the necessary background theory of KR&R and a detailed consideration of the remaining components of the system and its evaluation. Finally, I have discussed the starting point of the project, its deliverables and their dependencies, as well as the software engineering techniques employed.

Chapter 3

Implementation

In this chapter, I will present the implementation details of the Interactive Fiction expert system. This includes detailed discussions of the four major components of the system: the NLP frontend, the Standard and Enhanced engines and the Compiler backend. Finally, I will conclude with a summary including a side-by-side comparison of the two engines, executed on the same example story file.

3.1 Overview

The Interactive Fiction expert system, implemented for the purposes of my CST Part II dissertation, consists of four major components:

- **Standard engine** – Designed as a representative reproduction of an existing Interactive Fiction engine. It supports all the features outlined in §2.1, in their basic form.
- **Enhanced engine** – Builds upon the Standard engine, enriching it with automated responses.
- **NLP frontend** – Created to facilitate easier communication of commands to the engine, by allowing natural language to be used as input. Supports commands given in the form of a single, valid sentence of the English language.

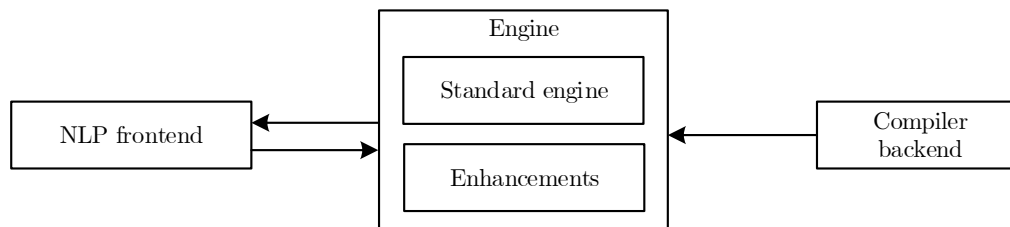


Figure 3.1: The structure of the Interactive Fiction expert system.

- **Compiler backend** – Designed as an additional level of abstraction, allowing story files written in a custom language, to be compiled and executed independently of the underlying engine.

The complete block diagram, showing the structure of the expert system can be seen in Figure 3.1. Looking ahead to the evaluation phase, the Standard and Enhanced engines were implemented as a single logical unit, so that the enhancements can be easily turned off to expose only the Standard elements of the engine.

Regarding execution, the top-level view of the component interactions can be seen in Figure 3.2. The important thing to note here is that the Compiler backend is only used at the start of the game – to initialise the engine and NLP frontend. Following this, the game-play exchange occurs between the user (via the console) and the engine. Here the user provides input to the engine, which is then interpreted as a command in the NLP frontend. The command is then executed on the engine, and an appropriate response returned to the user.

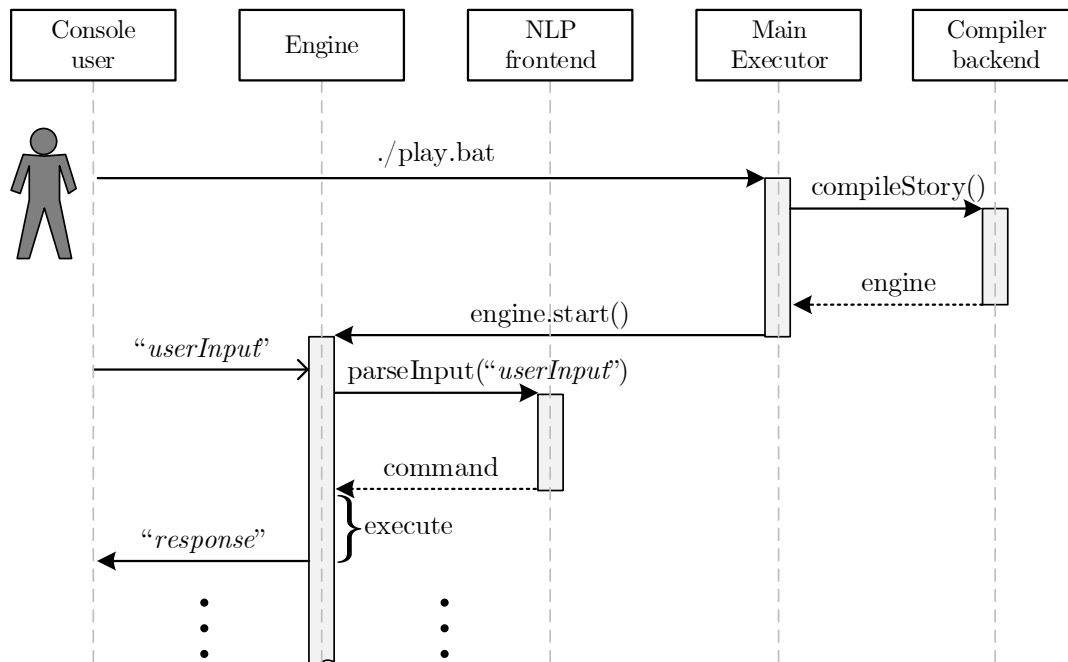


Figure 3.2: The sequence diagram showing a complete game execution (no-error case). Here, the Main Executor is a simple class used only to coordinate the other components on start-up.

The sequence shown in Figure 3.2 assumes no errors came up during compilation. However, if an error did occur, the Main Executor would wait for recompilation or until “quit” is input. On another note, I consider the raw console I/O as part of the standard engine, and not the NLP component. This was done to allow testing of the engine with a simplified input module, prior to the development of the NLP frontend.

I will now give a detailed account of the design, features and implementation of the four main components of the Interactive Fiction expert system.

3.2 Engine

The physical implementation of the engines is a single **Engine** class, which allows for either of the underlying engine's functionality to be easily exposed. Additionally, to reduce the complexity of adding Knowledge Representation and Reasoning to the Standard engine, the relevant standard component of the engine implements a knowledge base at its core. In this context, the knowledge base is used only as means of holding the game-world state, since this was deemed necessary in the specification determined in §2.1. Reasoning is not used here because that feature is reserved exclusively for the Enhanced component. Finally, the complete class diagram of the engine is given in Figure 3.3, showcasing the structure of the relevant classes necessary for both versions of the engine to function.

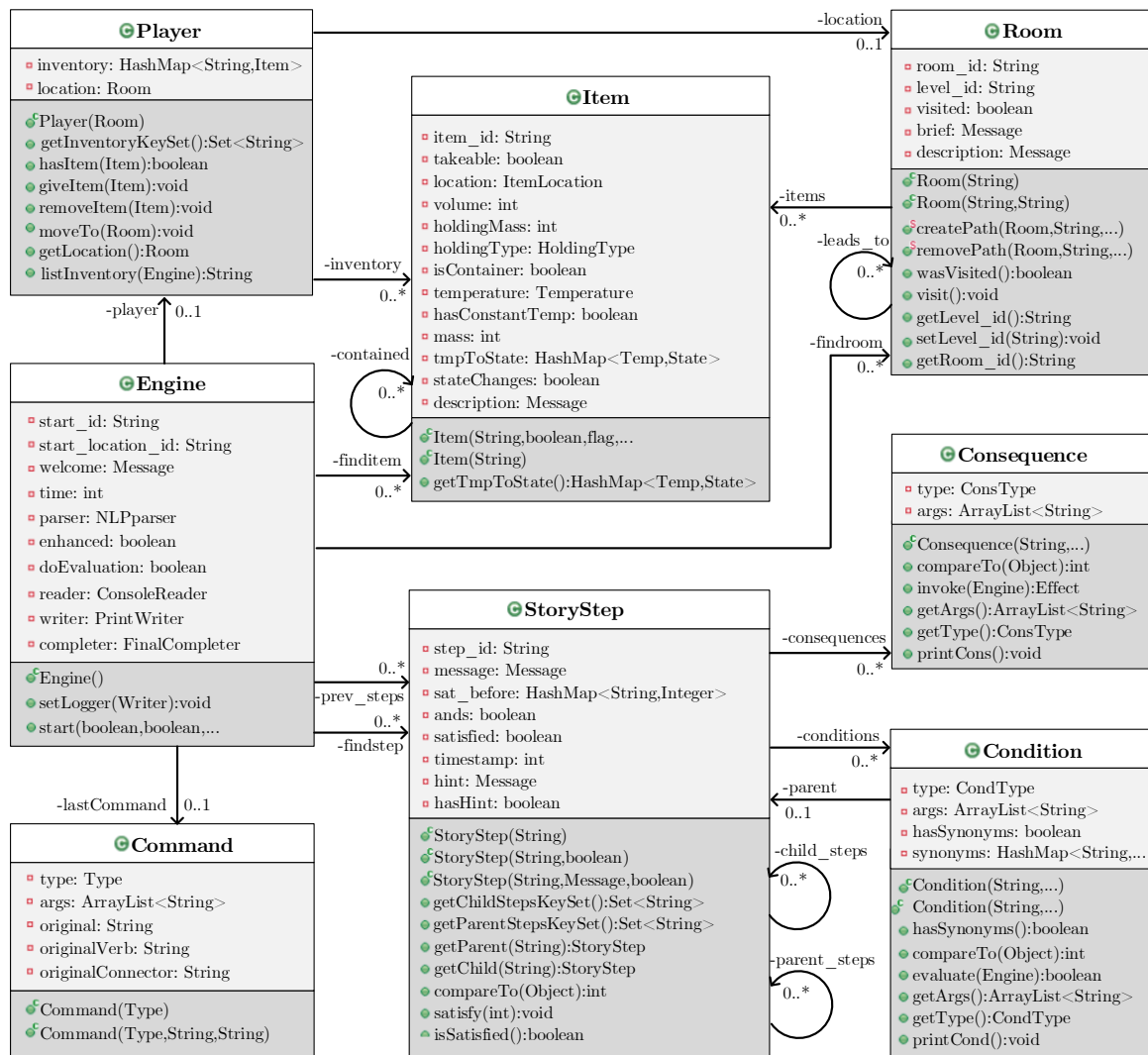


Figure 3.3: The class diagram of the complete engine. The less relevant member variables and function prototypes are excluded for brevity.

3.2.1 Standard Engine

The primary purpose of the standard engine is to hold, modify and evaluate the *Story State Machine*. The Story State Machine is a directed graph of *StoryStep* objects, which are connected according to (*parent* \rightarrow *child*) relations specified at initialisation. These relations are primarily important for advancing the game because they convey the paths which the story can take. The notion of step *triggering* is important here – to trigger a *StoryStep* means to satisfy a set of necessary conditions in-game, which results in the story moving forward, and the descriptive text associated with the triggered step being printed to the user. To make the process of triggering deterministic, and easier for the game developer to utilise, each *StoryStep* can only trigger once.

A step cannot trigger until either *all* or *any*¹ of its direct parents have triggered at a previous time-step. In addition to this, each *StoryStep* object contains a set of Condition and Consequence objects. Conditions can relate to any aspect of the game-world, such as the player or an item being at a specific location. They can also depend on what the previous user command was, these being: **combine**, **use**, **remove**, etc. Additionally, the consequences can execute physical actions on the objects of the game-world. By this I mean that they can “teleport” a player to another location, add, move or remove items from the game, open or close connections between locations and so on.

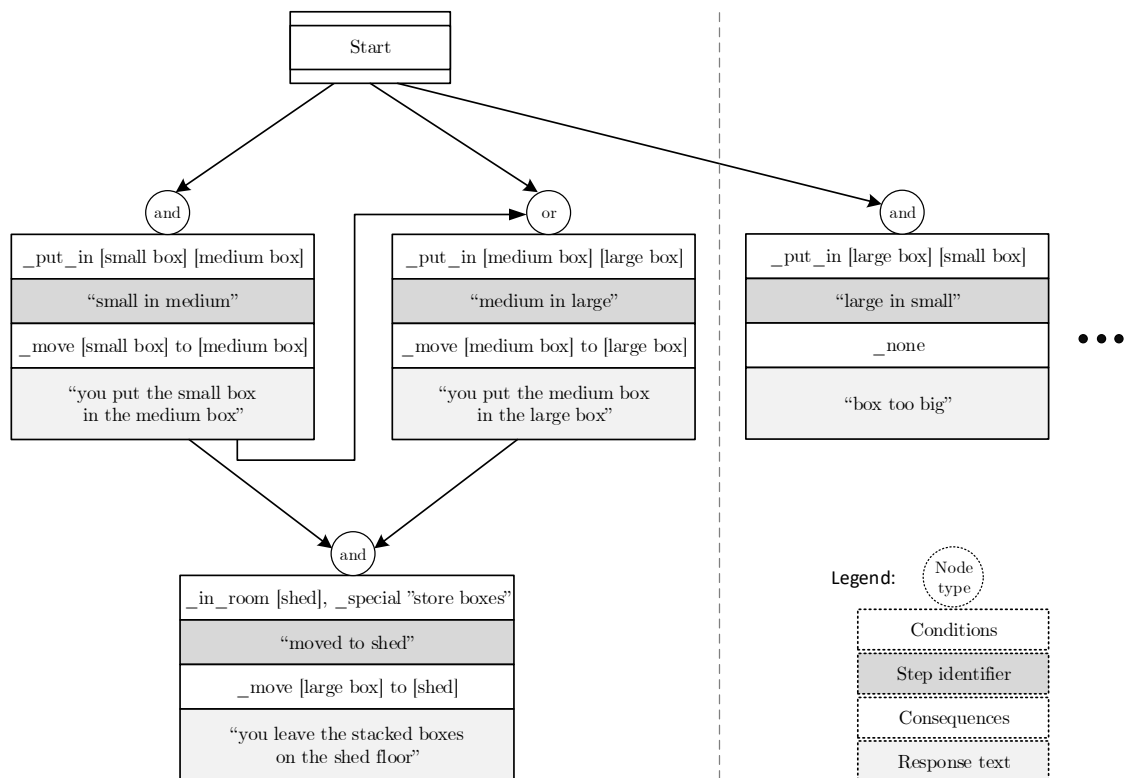


Figure 3.4: Example of a simple story state machine. Note that all steps to the right of the dashed line are **not** necessary when using the Enhanced engine.

¹Specified as a parameter at initialisation of the *StoryStep* – is it an *anding* or an *oring* node.

Before a step can trigger, all Condition objects associated with it must be satisfied, and this must happen after or at the same time as the satisfaction of its parent's criterion. Only one StoryStep can trigger in a single time-step, and once it does, all of its consequences, held in the appropriate Consequence objects, are incorporated in the state of the world. This is done by modifying the current situation appropriately (thus creating a new one), and then advancing to the next time-step.

The Story State Machine also has a specific starting step with no consequences and conditions, which is triggered automatically by the engine as soon as the game commences. This step serves as the entry point to the state machine and must be the parent of at least one StoryStep. Finally, the Story State Machine is terminated by any step which kills the player or grants him victory as its consequence.

An example of such a state machine is given in Figure 3.4. In this example, the story is based around storing away boxes of different sizes (small, medium and large). The goal is to have the small box in the medium one, and then both of them in the large one so that they could be stored away in a shed. This means that the volume constraints we wish to impose are $V[\textit{small box}] < V[\textit{medium box}] < V[\textit{large box}]$. Of course, in the Standard engine, we cannot set the volumes of items or any other parameter, so the resulting story state machine must be much more complex to achieve the same number of informative responses as the Enhanced engine.

3.2.2 Enhanced Engine

As the core of the project, the Enhanced engine consists of three major elements:

- **Standard engine** – on which it builds upon, and whose story state machine it uses to keep track of story advancement;
- **Physics model** – the chosen model, consisting of mutually interacting physical parameters, physically implemented in the knowledge base;
- **Knowledge Base** – implemented according to the object-oriented view of knowledge and consisting of the Item, Room and Player classes. It also contains the key attached procedures which enforce the constraints on the parameters dictated by the physics model.

3.2.2.1 Physics Model

To mitigate the limitations discussed in §1 and §2.1, an example physics model was developed. This model serves as the core of the knowledge about each individual item, and thus directly influences the design of the knowledge base itself. The attached procedures are also affected by it because they must implement the relevant imposed constraints. The final model consists of the following physical parameters (in order of development):

- **Volume** – represents the physical dimensions of an item (in *litres*);
- **Mass** – similarly to volume, stands for the physical mass of an item (in *grammes*);

- **Temperature** – represents the temperature *level* of an item on a descriptive scale of [*frozen* < *cold* < *normal* < *warm* < *hot*]²;
- **State** – like temperature, represents the physical state of matter of an item on the standard scale of [*solid* < *liquid* < *gaseous*]

The original model, given in the Project Proposal (Appendix D), suggested that only a single physical parameter was to be taken as the core of the model, and allowed for another one to be added as an extension. Fortunately, due to the project being ahead of schedule during Phase 0, I was able to not only implement the core and proposed extension of the model, but to also raise its complexity further, with two additional physical parameters. The choice to stop at four total parameters was made having evaluation in mind, knowing that the participants would be time-constrained when playing each individual engine, and thus would have a hard time experiencing even these four parameters fully. Of course, more parameters could be integrated in the future.

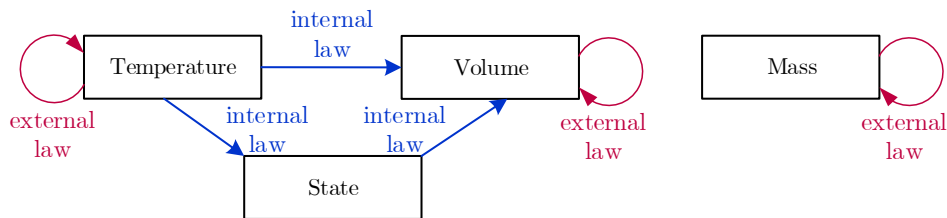


Figure 3.5: Simplified view of interactions between the parameters of the model. Here, $param1 \rightarrow param2$ indicates that $param2$ depends on $param1$, and the arrow colour indicates the type of law which governs the dependency.

The modelled physical laws that regulate the behaviour and mutual dependence of the parameters discussed above can be separated into two categories:

1. **Internal** – laws which govern the relations between the parameters of a single item. These laws are:
 - **Mass** and **temperature** are internally independent of all other parameters.
 - *Entities* (items and rooms) are allowed to have *fixed* parameters. Only parameters which do not depend on other parameters can be fixed though.
 - All rooms are fixed at the *normal* temperature level.
 - The **state** of an item is directly dictated by its temperature level. The exact mapping of this interaction is fully customisable (e.g. {[*frozen*, *cold*], [*normal*, *warm*], [*hot*]} maps to {*solid*, *liquid*, *gaseous*} directly).
 - The **volume** of the item is influenced by both **temperature** and **state**. This is done with special multipliers, linearly influenced by these parameters.

²The choice not to implement a numeric representation was made to facilitate clearer story development. Informal pilot testing early on has shown that the complexity of parameter interaction increases vastly when numeric values are taken instead, causing confusion amongst participants.

2. **External** – laws which specify the constraints between the parameters of two interacting *entities*. These laws are:

- When two entities *touch*³, they **exchange heat** amongst themselves and retain the average value of their initial temperatures. Of course, if the temperature is fixed for either of the entities, the one with variable temperature heats up/cool down to the temperature of the other one. Otherwise, if both entities have the same temperature level, nothing happens.
- For an item to be inserted into a container, the relevant developer-specified relation regarding the **holding volume** and **holding mass** must be satisfied. These relations specify either the *maximum*, *exact* or *minimum* cumulative quantity of volume or mass that can occupy the container.

A simplified view of the interactions between the four parameters of the model described above can be seen in Figure 3.5.

3.2.2.2 Knowledge Base

Having in mind how the physical model is designed, the knowledge base consists of three main classes:

- **Player** – the class which contains a list of items representing the player inventory as well as the current location of the player;
- **Item** – the class containing all four physical parameters of the item and its other relevant characteristics;
- **Room** – the class which contains a list of all items present in the room in the current time-step, as well as a list of exits to other rooms and other relevant characteristics.

In the context of the object-oriented representation of knowledge and the formalism presented in §2.2, the three main classes of the knowledge base can be expressed as generic frames, as illustrated in Figure 3.6.

The notion of a *situation* is very important here – a situation represents the state of all frames of the system at a given time-step. With this in mind, the main purpose of the knowledge base is to hold all the necessary information about the current situation. Of course, it serves another important purpose – to keep all the constraints of the physical model satisfied between its individual frames, and to act accordingly if they are not (refuse modification).

Of course, all of these classes have appropriate attached procedures which allow for the knowledge base to remain consistent with the laws of the model detailed in §3.2.2.1. The most important attached procedures are:

- *ChkConst* and *modTmp* – which are called when an item is added to a container item or a room. These procedures first check if all the relevant laws are obeyed, and if so, call the temperature modification procedures which cascade into modifying the appropriate remaining parameters if-needed;

³When an item is placed in a container item, or when an item is dropped in a room.

<pre> (Room <:IS-A Entity> <:Contains List<Item> [IF-ADDED modTmp]> <:Exits List<Room>> ...) </pre>	<pre> (Item <:IS-A Entity> <:Location Entity> <:Volume [IF-NEEDED ComputeVolume]> <:Mass Mass> <:Temp Temperature> <:State [IF-NEEDED ComputeState]> <:IsContainer Boolean> <:Contains List<Item> [IF-ADDED ChkConst]> <:HoldingMass Mass> <:HoldingVolume Volume> ...) </pre>
<pre> (Player <:IS-A Object> <:Inventory List<Item>> <:Location Room> ...) </pre>	

Figure 3.6: The three main classes of the knowledge base expressed in the frames formalism. The “...” represents the remaining properties of the generic frames, omitted for brevity.

- *ComputeVolume* and *ComputeState* – which are called by *chkConst* and other such procedures, when the volume or state are needed for constraint evaluation.

This is just a small subset of the necessary procedures, others were omitted for brevity.

3.2.2.3 Automated Response Generation

Another important thing to note about the attached procedures is that they return a String object, in addition to their appropriate return value. This String represents a fragment of the automated response relating to the action which was performed by the procedure. Due to the recursive nature of inference in an object-oriented knowledge base, these fragments can be assembled and integrated within caller procedures, and then hierarchically returned in the form of the complete sentence back to the user. For example, an exchange between a user and the engine could be:

> Please insert the key into the lock.

The warm solid golden key is too big to fit in the hexagonal lock.

The fragments of the response, obtained from their appropriate attached procedures, are highlighted and underlined. The relevant procedures which were activated in this example during inference, are given in detail in Figure 3.7. Here the inference failed, and the *ChkConst* procedure assembled the response reflecting that the volume of the key in relation to the holding volume of the lock, does not satisfy the physical laws of the model.

```

| → ChkConst(put_in [Golden Key] [Hexagonal Lock])
    | → CompareVolume([Golden Key] [Hexagonal Lock])
        | → ComputeVolume([Golden Key])
        | → ComputeVolume([Hexagonal Lock])
    | → AddIsOrAre([Golden Key])
        | → AddThe([Golden Key])
            | → GetFullID([Golden Key])
                | → GetTemperature([Golden Key])
                | → ComputeState([Golden Key])
    | → AddThe([Hexagonal Lock])
        | → GetFullID([Hexagonal Lock])

```

Figure 3.7: The procedures of the key-in-lock example, given in execution order. The *proc1* | *→ proc2* indicates that *proc2* was called from *proc1*.

3.2.2.4 Inference Loop

With all this in mind, the complete inference loop of the enhanced engine operates as:

1. Receive processed user command in the form of a Command Object;
2. Execute the appropriate command on the knowledge base, by performing the described action;
3. This activates all the necessary *IF-ADDED* and *IF-NEEDED* procedures which test the satisfaction of the laws of the physical model;
4. If these tests are not satisfied, the modification of the knowledge base is rejected, and proceed to step 8;
5. Otherwise, appropriate modifications are performed on the knowledge base – potentially triggering more procedures (e.g. temperature exchange);
6. The story state machine (from the standard engine) is then consulted to test for story advancement;
7. If there was a story step that triggered, its response is added to the one obtained previously, and the knowledge base is modified to reflect its consequences⁴;
8. Finally, the complete response, gathered in the previous steps, is printed to the user and the engine returns to step 1.

Note that the enhanced engine retains the standard engine's state automata for the purpose of determining key story actions. However, all the other manipulations of the items, rooms and the player are left to the knowledge base to maintain and execute, which is why automated responses reflecting this behaviour were possible in the first place.

⁴This does not activate procedures, it is the same modification as in the standard engine.

3.3 NLP Frontend

The NLP frontend consists of a single class, namely **NLPparser**, which utilises the natural language processing pipeline provided by CoreNLP. A diagram showcasing the exact dependencies is given in Figure 3.8, where the main components used from the CoreNLP pipeline, and their respective purposes, are the following:

1. **Tokenizer** – separates the sentence into *tokens* (useful semantic units of processing – usually words), throwing away punctuation;
2. **Part-of-speech (POS) tagger** [20] – assigns *parts of speech* (e.g. noun, verb, adjective, etc.) to each token;
3. **Lemmatizer** – reduces inflectional forms and derivationally related forms of the token to a common base form (e.g. am, is, are → be);
4. **Stanford Dependency graph** – provides a representation of grammatical relations between the tokens.

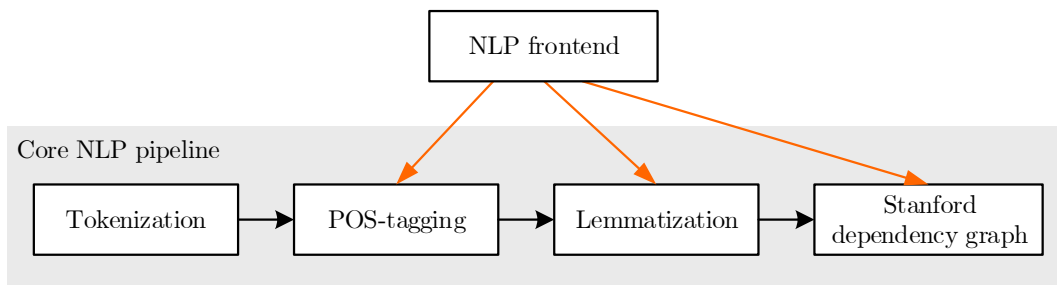
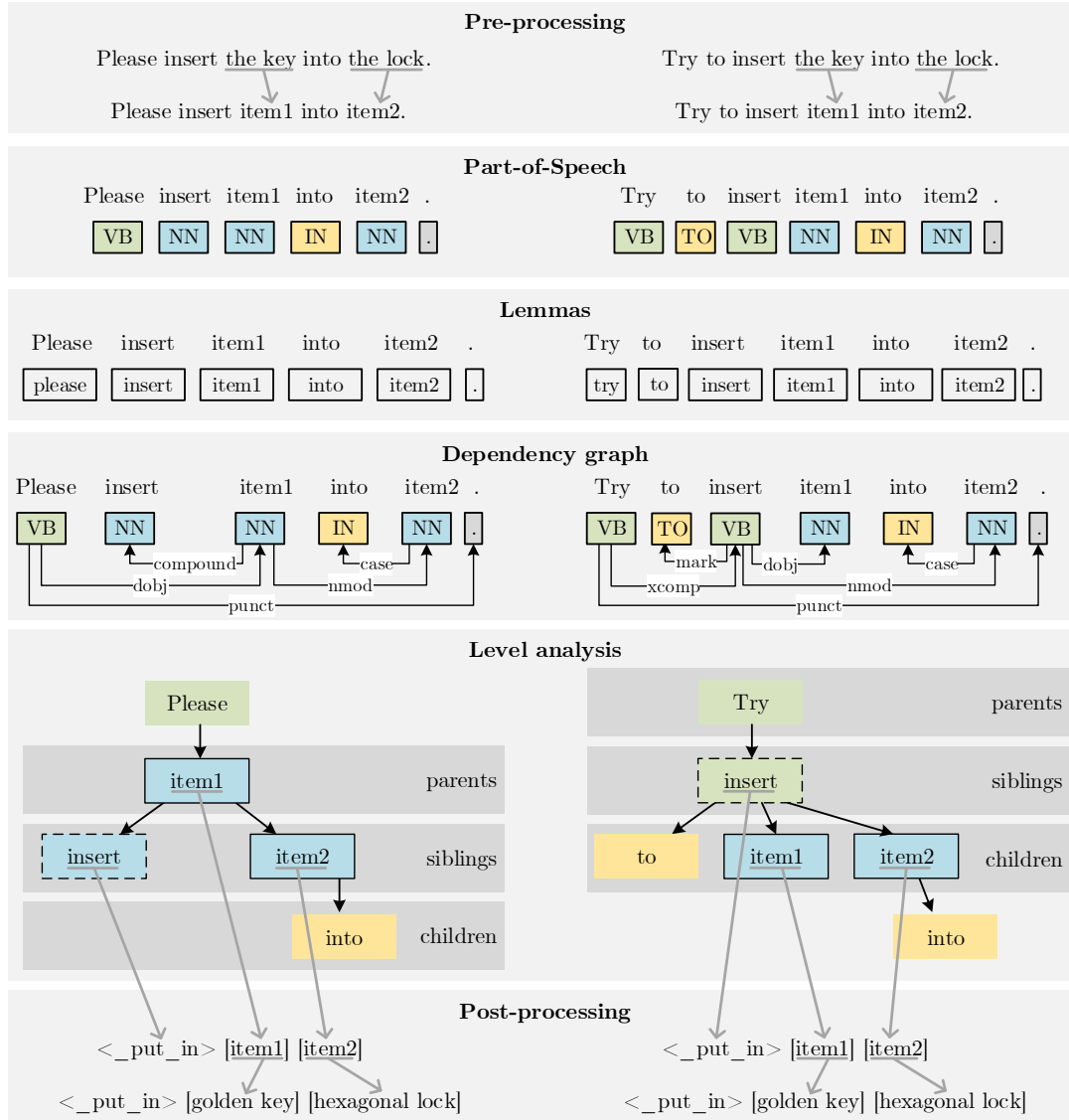


Figure 3.8: The major dependencies of the NLP frontend, on the CoreNLP pipeline. Dependencies and the internal sequence, are indicated by orange and black arrows respectively.

The frontend itself does most of its processing on the *Stanford dependency graph*. This graph is particularly important because the commands implemented by the engines all have a very simple grammatical structure (i.e. a verb predicate and up to two noun objects). This means that the key elements of the command can easily be extracted once the grammatical structure is apparent through the use of the dependency graph.

3.3.1 Sentence Analysis

The NLP frontend’s main task is to analyse user input, provided by the engine in the form of a String object, and return a special **Command** object containing the processed command. Two examples, showcasing this analysis, are given in Figure 3.9.



(a) Here, the verb **insert** is wrongly recognised as a noun.

(b) The sentence grammar is correctly recognised in the dependency graph.

Figure 3.9: Two similar example sentences, analysed using the NLP frontend. Since they convey the same command, the end result is identical in both cases.

Before the user sentence can be parsed with the NLP pipeline, pre-processing must occur. In this step, *item identifiers* (e.g. **the key**, **the lock**) are recognised⁵ and replaced with place-holders (i.e. **item1**, **item2**).

Once the pre-processing is finalised, the modified sentence is analysed with the NLP pipeline. Unfortunately, the grammar structure recognised in the resulting Stanford dependency graph does not always capture the command's grammar correctly (e.g. **insert** is interpreted as a noun instead of a verb in Figure 3.9a). This is primarily due to the

⁵Performed using longest prefix match against a database of potential identifiers. This database is computed from the list of possible items that can be interacted with in the current time-step – all the items from the current room and player inventory.

effects of the POS-Tagging phase because the corpus on which the tagger was trained does not correctly represent Interactive Fiction. Since it was infeasible⁶ to train the tagger on a custom corpus, another approach had to be taken to avoid incorrect command interpretation.

The approach taken was to first isolate the *identifying verb* of the command (e.g. `insert`). This is done by iterating through the set of all⁷ possible command verbs in priority order, and attempting to find the verb in the sentence by comparing it against the lemmatization of each token, until a match is found. It was feasible to do this due to the small size of the complete command set. Having the identifying verb isolated also disambiguates the exact type (e.g. `use`, `combine`, etc.) of command, determining the number and type of arguments that need to be retrieved.

To retrieve the command’s arguments, the entire parent, sibling and child levels around the relevant identifying verb are extracted from the dependency graph. The complete list thus obtained is then searched for arguments, whose quantity, type and any potential connectors (e.g. `into`) were determined in the previous step. Once the relevant arguments are found, the place-holders created in the first step are disambiguated, and a Command object is prepared and returned.

Particular care was taken throughout the entirety of the procedure described above. Error checking, malicious command detection and a special Exception-handling wrapper are all in place to ensure that the NLPparser exhibits graceful degradation in the event of a problem. This allows the remainder of the system to continue operating uninterrupted, making it easy to instantiate a new NLPparser for the next user command – thus minimising the impact on the user experience.

3.4 Compiler Backend

As outlined in §2.5.1, the custom story language which is at the core of the Compiler backend was written as a context-free grammar. Due to the complexity of this grammar, a parser generator (such as ANTLR) was necessary to ensure successful implementation of the backend. For these reasons, the lexer and parser, generated by ANTLR, play a key role in the two initial steps of the compilation pipeline.

The Compiler backend itself consists of a main **StoryCompiler** class containing the complete compilation pipeline, as can be seen in Figure 3.10. The components of this pipeline, in the order of execution, are:

1. **StoryGrammarLexer** – the lexer, automatically generated by ANTLR, with the purpose of performing lexical analysis of the input story file, and producing an appropriate token stream;

⁶Both because it would be impossible for me to tag a data-set by hand on my own, and because this was out of the scope of the project.

⁷This includes the hard-coded synonyms extracted from WordNet, as well as the custom developer-defined synonyms. More details on the custom synonyms can be found in §3.4.

2. **StoryGrammarParser** – the parser, also automatically generated by ANTLR, whose purpose is to create a parse tree over the stream of tokens provided by the lexer;
3. **StoryTreeVisitor** – the interpreter class, whose goal is to recursively analyse the parse tree, and assemble the appropriate **engine**;
4. **StoryLinker** – the linker class, designed to test if all the references within the newly created engine exist, are valid and create new ones if necessary.

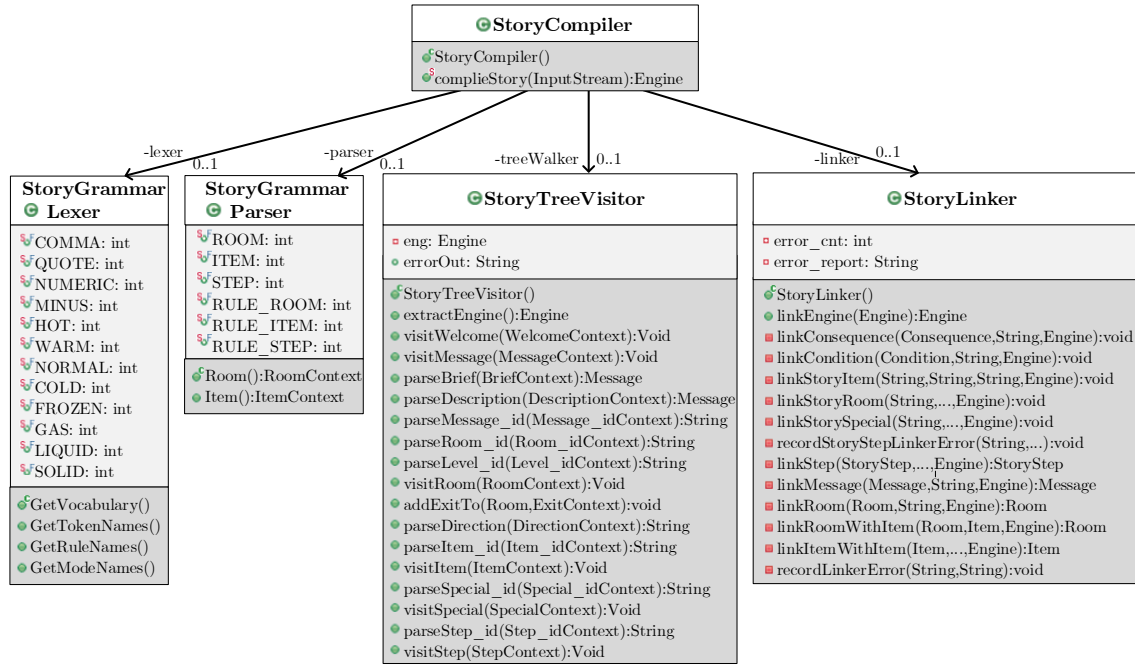


Figure 3.10: The class diagram of the Compiler backend. The less relevant member variables and function prototypes were excluded for brevity.

I implemented the visitor and linker classes paying careful attention to error reporting. All the errors which occur at any stage of the pipeline are reported, and the rest of the pipeline is halted, producing no engine. Additionally, I exposed all of the errors reported by the automatically generated classes and incorporated them into the error report. These informative reports allow for issues in the story-file to be identified more accurately, alleviating a degree of complexity from game development.

3.4.1 Story Language

The most important element of the Compiler backend to discuss is the custom Story Language, which was used to generate the relevant classes using ANTLR, and heavily influenced the development of the **StoryTreeVisitor** and **StoryLinker** classes. An example story file, designed to illustrate most of the important features of the Story Language, is given in Figure 3.11.

The Story Language itself has a structured format motivated by XML, as well as line endings and bracketing similar to those of the C language family. To reduce the complexity

<pre> 1 _welcome 2 { 3 [start]; 4 [example room]; 5 "This is a short example 6 story:"; 7 } 8 9 _room 10 { 11 [example room]; 12 [first level]; 13 _N - "You cannot go there."; 14 "Welcome to an example room! 15 There are an elephant and 16 a fridge here."; 17 "You are in an example 18 room."; 19 } 20 21 _item 22 { 23 [elephant]; 24 _takeable; 25 _inroom [example room]; 26 _isitem; 27 _vol 3000; 28 _temp _warm, _constant; 29 "An adult elephant."; 30 } </pre>	<pre> 31 _item 32 { 33 [fridge]; 34 _fixed; 35 _inroom [example room]; 36 _iscont, _max, 2000; 37 _vol 200; 38 _mass _surpress 1000; 39 _temp _cold, _constant; 40 "Your average fridge."; 41 } 42 43 _step 44 { 45 [throw fridge at elephant]; 46 [and]; 47 [start](0); 48 _useon {(<throw>, <at>), 49 (<toss>, <at>)} 50 [fridge] [elephant]; 51 _none; 52 [why do that]; 53 } 54 55 _message 56 { 57 [why do that]; 58 "Why on Earth would you do 59 that!?"; 60 } </pre>
---	--

Figure 3.11: A small, self-contained story-file example, showcasing the expressive power of the Story Language.

of the grammar, underscores (“_”) are used to prefix key-words, and all item and room identifiers are required to be enclosed in square brackets (“[” and “]”). This is done to allow arbitrary alpha-numeric strings to be used for identifiers, making the language more readable. There are six main block types, used to describe various Interactive Fiction elements, these being:

- **_welcome** – used as the *entry point* of the story file, and consisting of the starting step identifier, initial player’s location, as well as the *welcome* message (which is only printed once upon starting the game);
- **_room** – the block defining a **room** – an ambient element of the story, and the setting where items and the player exist. Multiple rooms can be present, and traversing

them is possible by defining *connectors* in all four cardinal directions⁸ of a given room;

- `_item` – the block used to create an **item** – the key element of the story, ambience and logical puzzle creation. This is where the relevant physical parameters can be defined. If a parameter is omitted, it will take on a predefined default value;
- `_step` – the block which defines a *story step*, defined in §3.2.1, representing an important *moment* in the plot of the game;
- `_message` – designed to allow for long formatted messages to be separated from their respective blocks;
- `_special` – used for defining fixed “special” commands, as discussed in §2.1.1.

These blocks can be ordered arbitrarily in the story file, save for the `_welcome` block, which must precede all other blocks in the code. This is necessary because it is the entry point of the story.

Another interesting feature of the language to explore is the custom synonyms support. For each story step that depends on a user-input action as its precondition, a set of custom synonyms can be defined. These synonyms will apply only to that step, and retain the same functionality as the command they are substituting. This allows for the creation of more immersive stories because the developer can define custom commands which better reflect the desired action(s). For example, the step defined in Figure 3.11 has two custom synonyms defined for the `use on` action (keyword `_useon`), these being `throw at` and `toss at`. Of course, the implementation of this feature is appropriately reflected in the NLP frontend.

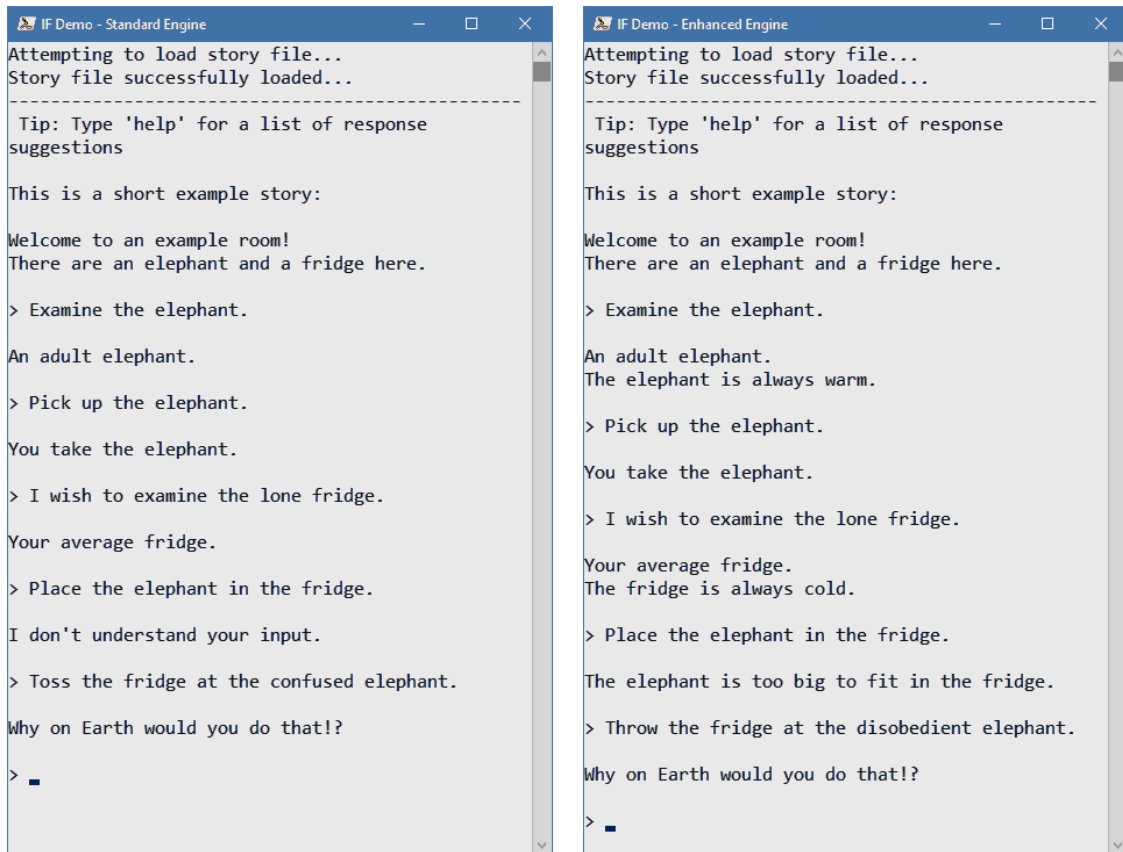
Finally, for all the remaining details which I have not covered explicitly in this chapter, please consult the full grammar provided in Appendix A.

3.5 Summary

In this chapter, I have summarised the implementation work that was performed in phase zero of development, before the design and development of the user study (phase one), could commence. I first provided a detailed overview of the entire system and the way the four key components interact when communicating with a user. I then discussed the main elements of the engines: the story state machine for the Standard engine, and the physics model, knowledge base, automated responses and inference loop in the case of the Enhanced engine. Following this, I explored the implementation details of the NLP frontend and the Compiler backend, discussing their features.

Finally, to further clarify the differences between the Standard and Enhanced engines of the Interactive Fiction expert system, Figure 3.12 showcases two short excerpts from the execution of these engines. In this example, both engines were compiled with the same story file (given in Figure 3.11), and then input with an ordered, identical set of

⁸i.e. where `_N` stands for “*towards North*”, `_E` “*towards East*”, etc.



(a) Execution trace on the **Standard** engine. (b) Execution trace on the **Enhanced** engine.

Figure 3.12: A side-by-side comparison of the two engines, executing the story file from Figure 3.11.

commands⁹. Here, the Enhanced engine provided more verbose and informative answers, compared to its standard counterpart.

⁹Save for the final one, which is there to demonstrate the custom synonyms discussed in §3.4.1.

Chapter 4

Evaluation

In this chapter, I will present the tools and procedures used to evaluate the Interactive Fiction Expert System. This includes a detailed overview of the User Study, used to evaluate the engine, and its results, as well as the unit testing and play-testing utilised for the evaluation of the remaining components.

4.1 Overview

The evaluation of the Interactive Fiction expert system consisted of completing several success criteria, outlined in the project proposal, and further explored and extended in §2.6. The main satisfaction criteria thus defined were:

1. Evaluation of the **NLP frontend**, with detailed unit testing;
2. Evaluation of the **Compiler backend**, by being able to interpret and store a simple story file;
3. Quantitative comparison of the **Standard** and **Enhanced engines**, by means of an exhaustive user study, where the primary problem statement (as specified in §2.6) would be resolved.

I will now discuss how each of these evaluation criteria was satisfied.

4.2 NLP Frontend Testing

As part of the satisfaction criteria requiring unit testing of the frontend, I developed a custom testing framework based on the class **NLPtest**. This class executes all of the test user-input sentences using the NLPparser, compares the resulting command to the expected interpretations, and reports any mismatches detected as errors. Both of these lists (user-input sentences and expected interpretations) are given to the framework in the form of plain-text files, to allow for a simple, on-demand addition of new tests. The execution of a single unit-test batch on the framework is given in Figure 4.1, and the relevant unit-test files can be found in Appendix B.

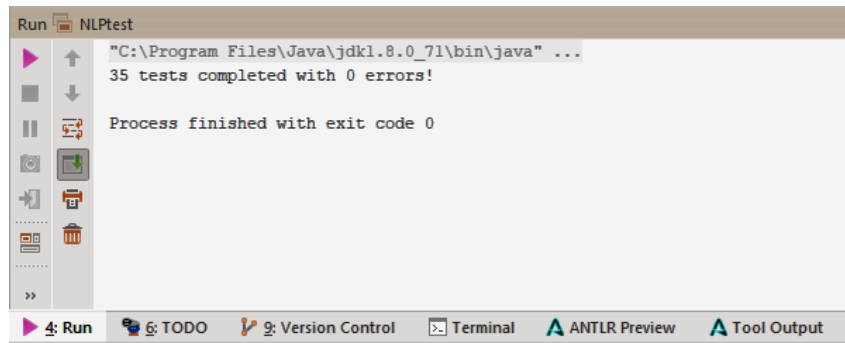


Figure 4.1: A single run of the NLPtest framework.

4.3 Compiler Backend Testing

Due to the complexity, and frequent upgrades to the structure of the language itself, it was unreasonable to develop a full unit testing suite for the Compiler backend. The approach taken instead, was frequent play-testing of the representative final game-file while scanning for inconsistencies in the story, bugs in the **engine** and issues with the **Compiler**. Of course, for a play-test to be run in the first place, the Compiler must not have produced errors at any stage of compilation. An excerpt from one such play-test is given in Figure 4.2.

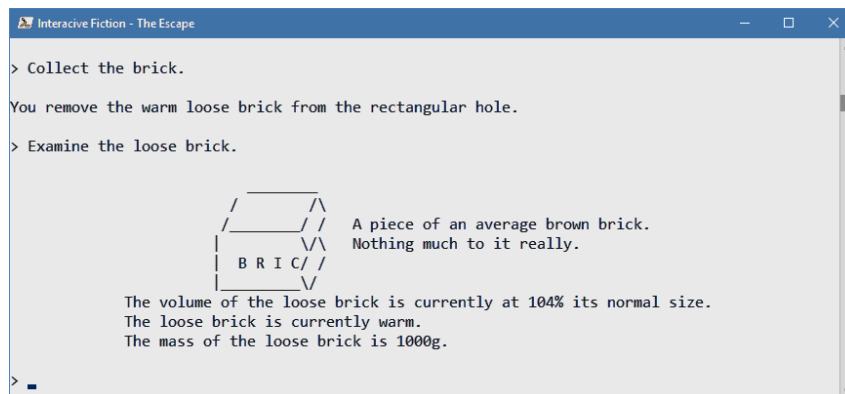


Figure 4.2: A play-test of the final game-file.

4.4 Engine Evaluation

Out of all the satisfaction criteria, by far the most important one is the quantitative comparison of the two engines. Even though the outcome of the experiment would not have a direct effect on the success of the project as a whole, it was crucial in determining the resolution to the main problem statement, presented in §2.6.

The remainder of this section describes the work performed in Phase one of the project. During this phase, the user study was designed with careful attention to detail and performed on 22 participants. The results obtained, and presented below, conclusively show that the answer to the problem statement is a positive one – the Enhanced engine was superior to the Standard one, on all of the measured metrics.

4.4.1 Experiment Set-up

To be able to correctly evaluate the problem statement, a three-stage controlled experiment was designed. These stages were:

1. **Briefing** – a short group session, where the participants were familiarised with the interface of an IF game, and given details about the experiment. They were also asked to fill out a short *demographic questionnaire* and *consent form* at this stage.
2. **Sessions & Feedback** – where the participants were asked to play two 20-minute long *sessions*¹ of the same IF game, each time on a different engine. While in-game they were asked to evaluate some *individual responses*, and to then fill out an appropriate *questionnaire* upon completing each session.
3. **Debriefing** – designed primarily to collect *qualitative data*, this stage consisted of 2-on-1 group interviews with about ten minutes of guided discussion.

In such an experiment set-up, participants would be playing *the same game* twice. This means that their experience from the first session could have an effect on their perception of the second. This is partly because they would have become familiar with the story itself. To mitigate any potential impact this would have on the experiment, the participants were divided into two equal groups:

1. *Standard-first group* – which had their first session executed on the Standard, and the second one on the Enhanced engine;
2. *Enhanced-first group* – similarly, participants in this group played their first session on the Enhanced and the second one on the Standard engine.

Neither of the groups was made aware of which engine the game was running on, in either of their sessions. Such a step was necessary in order to remove any potential bias.

During the creation of this user study, and all of its respective components, several iterations of the **iterative development model** were performed (as part of Phase one outlined in §2.6). At the end of each iteration, a comprehensive informal pilot test was conducted, on two individual participants. This provided valuable feedback and aided the creation of new improvement goals for both the documentation, game file and engine features, which were to be executed in the iteration that followed. I will now discuss the final versions of the more important documents and features developed during Phase one.

4.4.1.1 Questionnaires

To be able to correctly evaluate the metrics defined in the problem statement, two questionnaires were designed – each of them corresponding to one of the two sessions played during the **Sessions & Feedback** phase of the experiment. Each of these questionnaires primarily contained an identical set of *Likert items* – individual statements, which participants are asked to evaluate on a *visual analogue scale*. The exact scale used in this experiment was the *five-level Likert scale*, specified in Figure 4.3.

¹A single play-through of an IF game, of a single player, as defined in §1.3.

1. Strongly disagree 2. Disagree 3. Neither agree nor disagree 4. Agree 5. Strongly agree

Figure 4.3: The ordered set of response levels used for the Likert scales.

Each Likert item was designed to test **exactly one** of the metrics outlined in the problem statement. With this in mind, a full list of all the Likert items which formed the questionnaires is given in Table 4.1.

Likert Item	Metric
The responses in this session were helpful.	helpfulness
I was satisfied with the way responses guided me in this session.	guidance
The text of the responses in this session was clear.	clarity
The game interface was easy to use.	comfort
I found this session frustrating.	frustration
I enjoyed this session.	satisfaction

Table 4.1: A list of all the Likert items posed in the two questionnaires, and their respective measured metric. Note that the sessions were appropriately labelled as *first* and *second* in each questionnaire, to further the anti-bias plan outlined earlier.

4.4.1.2 In-game Evaluation – KeySEF

To obtain more fine-grained information about the quality of user experience, I implemented a special feature in the mutual basis of the game engines. This feature is called **Key Step Evaluation Framework (KeySEF)**, and it allows for posing Likert items **in-game**, immediately following a response generated by an engine. The main features of KeySEF are the following:

- It is fully configurable, and allows for a set of *important plot moments* of an IF game to be defined, for which evaluation is necessarily desired;
- Whenever any of these moments are reached *for the first time* during a session, KeySEF initiates *evaluation mode* on the engine. In this mode, the usual engine response and its appropriate Likert item are printed, and the user response is recorded. The mode then terminates with the specified *closing sentence*;
- To make the evaluation requests appear as if they occur at random and to reduce user bias, KeySEF also initiates evaluation mode for all game moments with a probability of 20%.

For the purposes of the experiment, only three key moments were identified and defined, evenly spaced throughout the game’s plot. This was done primarily to reduce the frequency of evaluation requests since pilot testing has shown that the reliability of such

evaluations decreases with an increase in their number. Finally, the exact parameters used to configure KeySEF to measure helpfulness in the study, are given in Figure 4.4.

```
>-----<
> I feel this response helps me to make progress in the game. <
> 1 - Strongly disagree <
> 2 - Disagree <
> 3 - Neither agree nor disagree <
> 4 - Agree <
> 5 - Strongly agree <
```

(a) The choice of *evaluation question*.

```
> Entry recorded. Continuing... <
>-----<
```

(b) The choice of *closing sentence*.

Figure 4.4: The configuration of KeySEF for the IF user study.

4.4.1.3 IF Game File

In addition to the appropriate documents necessary for the three main stages, a special *IF game file* was developed, solely for the purposes of the study. It was written in the language described in §3.4.1 so that it can be run on both versions of the engine interchangeably. A custom linear story was developed here, in order to expose the differences in the engines, rather than ones originating from the exact story-related choices made during the sessions. Due to the file being quite long, instead of the code itself I provide an example session, showcasing the game-file’s execution on the Enhanced engine, in Appendix C.

4.4.1.4 Additional Engine Features

Finally, to facilitate the smooth execution of the user study, and collect **further data**, several additions were made to the mutual basis of the game engines. These features are as follows:

- **The study harness** – A wrapper for all four main components of the system, with the purpose of executing the complete experiment.
- **Timestamp enriched transcripts** – The *session transcript* containing a complete snapshot of a single session, and the *feedback transcript* containing the user responses obtained via KeySEF.
- **Hinting framework** – Allows the players to request context-sensitive, plot-related aid, by typing the special command `hint`. It was developed to help overcome the 20-minute time limit and is fully configurable through the story language.

- **Console features** – Facilitating a smoother console experience, and resembling the Unix command-line, these features are: *Item name auto-complete*, *String manipulation features* and *Command history*.

4.4.2 Results & Analysis

The user study, as specified above, was performed on **22 participants**, which more than doubly exceeded the initially anticipated turn-out of ten. Thanks to this, a great volume of useful data was obtained over the course of the experiment. With this in mind, the exact data obtained for each participant includes:

1. **Two questionnaires** – relating to their **first** and **second** session;
2. **Two session transcripts** – one for each appropriate session;
3. **Two KeySEF response sets** – evaluating the **three key steps**² in each session;
4. **Debriefing recording** – relating to their debriefing session.

The most valuable data here is the *quantitative data*, obtained through the questionnaires and KeySEF, and analysed in detail in the statistical analysis that follows. The remaining data is qualitative and serves only as circumstantial backing, and as such will not be discussed in detail further.

4.4.2.1 Hypothesis Testing

Having in mind the *main problem statement*, as given at the beginning of this chapter, I will now define two hypothesis:

1. **H₀** – the **null hypothesis** ($\mu_0 = 0$):

*The **Enhanced** and the **Standard** engines **cannot be distinguished**, on the scales of the control metrics, when executing the same Story file.*

2. **H_a** – the **alternative hypothesis** ($\mu_0 > 0$):

*The **Enhanced** engine receives **better scores** than the **Standard** one, on the scales of the control metrics, when executing the same Story file.*

Both of these hypotheses apply on a per-metric basis, where the control metrics are defined as *helpfulness*, *guidance*, *clarity*, *comfort*, *frustration* and *satisfaction*.

Additionally, all the obtained qualitative data is in the form of Likert scales. These come from Likert items which precisely define their appropriate metric (Figure 4.1), and most importantly – they **are paired**. For these reasons, the **paired difference test** is perfect for evaluating which of these hypotheses is valid.

²These steps were necessarily triggered in *both sessions*, and thus could be paired.

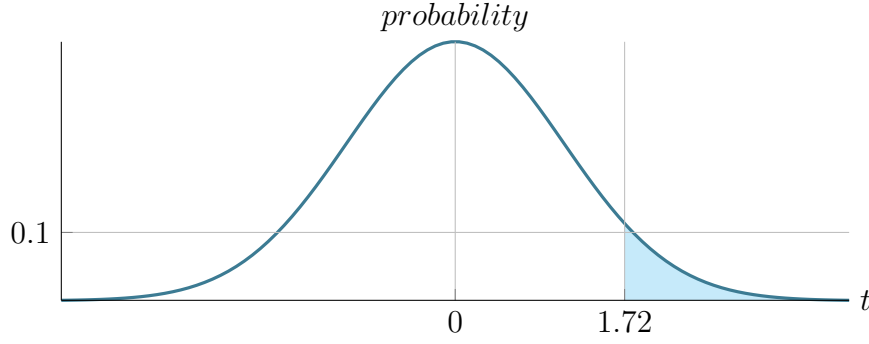


Figure 4.5: The plot of the **t-distribution** probability density function for 21 degrees of freedom. The shaded area represents the threshold p-value of $p = 5\%$, with its appropriate t-value of $t_{5\%} = 1.72$.

For the purposes of this analysis, the **single-tailed paired t-test** [19] was applied to the data, for each individual metric *independently*. This type of t-test aims to find evidence of significant difference between a population mean and a hypothesised value (zero in this case). The **t-value** measures the magnitude of this difference, relative to the variation in the sample data. Hence, the greater the value of t (either positive or negative, depending on the scenario), the *greater* the evidence *against* the null hypothesis. Here, the t-value is calculated as follows:

$$t = \frac{\mu_D - \mu_0}{\frac{s_D}{\sqrt{N}}} \quad (4.1)$$

where the parameters are:

- μ_D – the *mean* of all the paired differences;
- μ_0 – equals *zero*, and stands for the hypothesised value of the mean;
- s_D – the *sample variance* [1];
- N – the total number of paired data-points for the metric in question.

The t-value thus defined, is used to compute the **p-value** – the probability that a value of t , at least as extreme as the t-value, is observed under the null hypothesis. This is done by evaluating the *t-distribution* with $df = N - 1$ degrees of freedom – an example of which can be seen in Figure 4.5.

The common choice of *threshold p-value*, in determining statistical significance, is $p = 5\%$, which corresponds to a **confidence interval** of 95% when proving the alternative hypothesis. This, in turn, gives us $t_{5\%}$ – the **threshold t-value**, retrieved from the t-distribution with the appropriate df . This threshold t-value is particularly important because it allows for a simple way to test the validity of the alternative hypothesis:

If the t-value obtained by analysing the experimental data is greater than the threshold t-value, \mathbf{H}_0 is rejected, and \mathbf{H}_a is confirmed with statistical significance. Otherwise, \mathbf{H}_0 cannot be rejected with statistical significance.

Metric	Both groups			Standard-first			Enhanced-first		
	t(21)	t _{5%} (21)	sgnf?	t(10)	t _{5%} (10)	sgnf?	t(10)	t _{5%} (10)	sgnf?
Helpfulness	5.55	1.72	yes	3.07	1.81	yes	5.16	1.81	yes
Guidance	3.15	1.72	yes	0.32	1.81	no	5.16	1.81	yes
Clarity	2.08	1.72	yes	0.00	1.81	no	2.61	1.81	yes
Comfort	3.17	1.72	yes	2.89	1.81	yes	1.94	1.81	yes
Frustration	-7.51	-1.72	yes	-3.73	-1.81	yes	-9.81	-1.81	yes
Satisfaction	2.02	1.72	yes	-0.80	1.81	no	3.83	1.81	yes
KeySEF helpfulness #1	8.59	1.72	yes	5.04	1.81	yes	7.37	1.81	yes
KeySEF helpfulness #2	5.43	1.72	yes	2.19	1.81	yes	7.42	1.81	yes
KeySEF helpfulness #3	12.75	1.72	yes	7.60	1.81	yes	10.70	1.81	yes

Figure 4.6: The table containing the results obtained from the user study, when the statistical t-test is applied, including the fine-grained data obtained through KeySEF. Here $t(df)$ signifies the t-value obtained from the appropriate data, for df degrees of freedom, $t_{5\%}(df)$ signifies the t-value for the 5% margin for that df and **sgnf?** represents the question “*Was statistical significance achieved?*”. Note that frustration is a negative trait – a smaller value yields a better score.

With all this in mind, the single-tailed paired t-test was applied to the paired data obtained in the experiment, for each relevant metric. The obtained results can be seen in Figure 4.6.

It is important to note that H_a is confirmed, with **statistical significance across the board**, when both groups of participants are considered together (both Enhanced-first and Standard-first). However, when the data is divided by participant group, a clear distinction is noticed in the t-value for nearly all metrics. When just the questionnaire data of the Standard-first group is considered, H_a is confirmed for only half of the metrics. Fortunately, KeySEF was able to provide much more conclusive results in this case, confirming H_a for all three key questions on the helpfulness metric. On the other hand, when observing the Enhanced-first group, all results confirm the alternative hypothesis with consistently large t-values, suggesting a far greater confidence interval than 95%.

This confirms my suspicion that the experience from the first session inevitably affected the second one for all participants. A plausible explanation for this observation is that the Enhanced-first group became familiar with the feature-rich Enhanced engine. They thus had a clear metric to judge the Standard engine with, unlike the Standard-first group, which had features introduced in their second session.

4.4.2.2 Data Analysis

Having proven the alternative hypothesis, further conclusions about the superiority of the Enhanced engine can be made by observing the data directly. With this in mind, I will now give a brief overview of the more prominent features noticed in the data.

Firstly, Figure 4.7 shows the responses to all the Likert items, collected from the questionnaires for both groups of participants combined. What is interesting to note

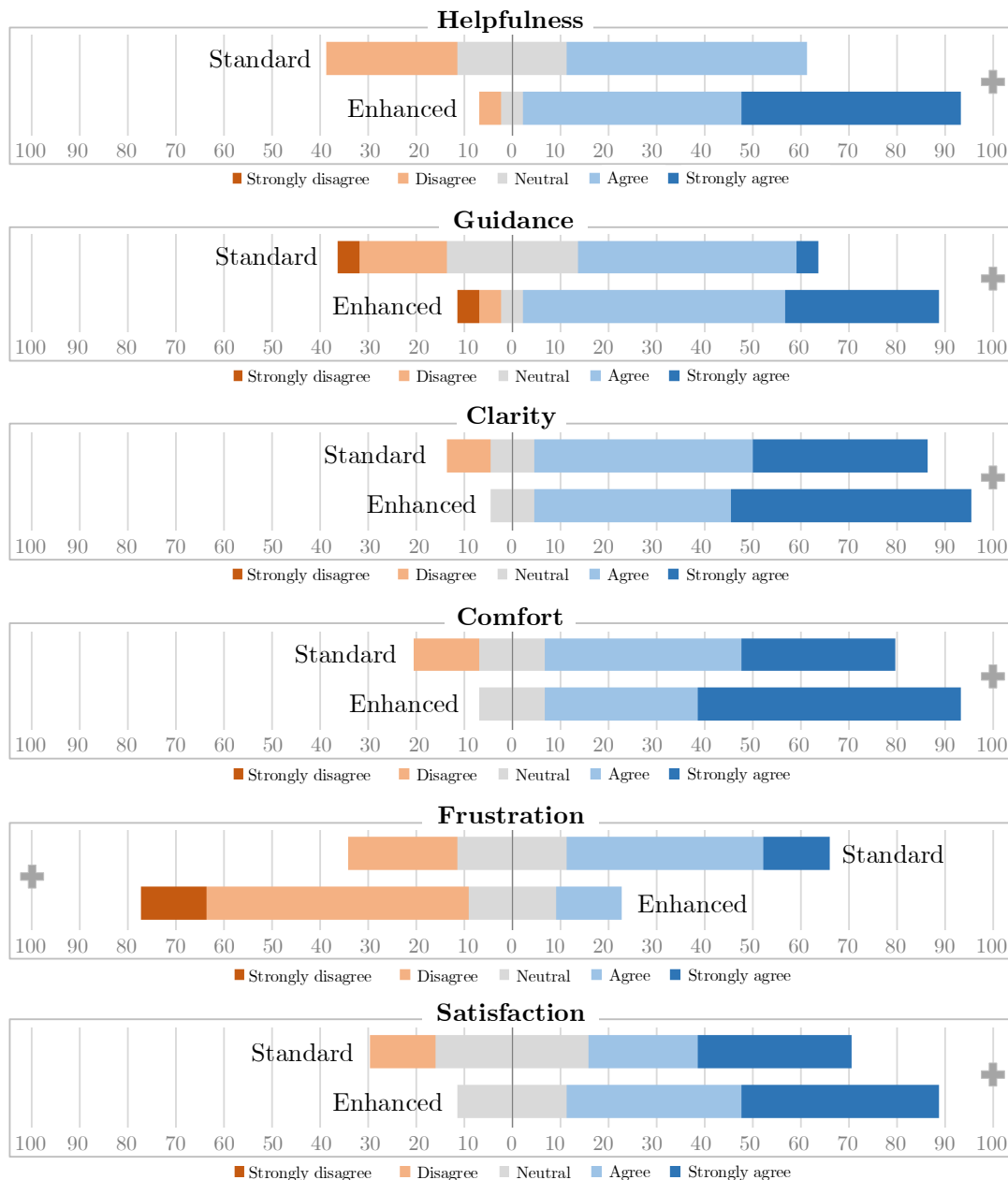


Figure 4.7: A comparison of the results, for the data obtained from the questionnaires, for all metrics, on a percentage scale. This and similar charts were produced according to suggestions from [6].

here, is that the Enhanced engine fares *consistently better* than its Standard counterpart, on each and every one of the designated metrics. The lack of frustration is most prominent here, closely followed by a distinct increase in helpfulness, guidance and satisfaction. The only minor improvements in clarity and overall comfort could be explained by the fact that most of the responses are fixed as part of the story, and as such clear by design.

Since I consider helpfulness an important aspect of an IF game, further evaluation of this metric was performed through KeySEF. The data obtained for both groups, per engine, is given in Figure 4.8. This exposed even more profound differences, in favour of the Enhanced engines, for all three key steps of the experimental IF game. I believe this

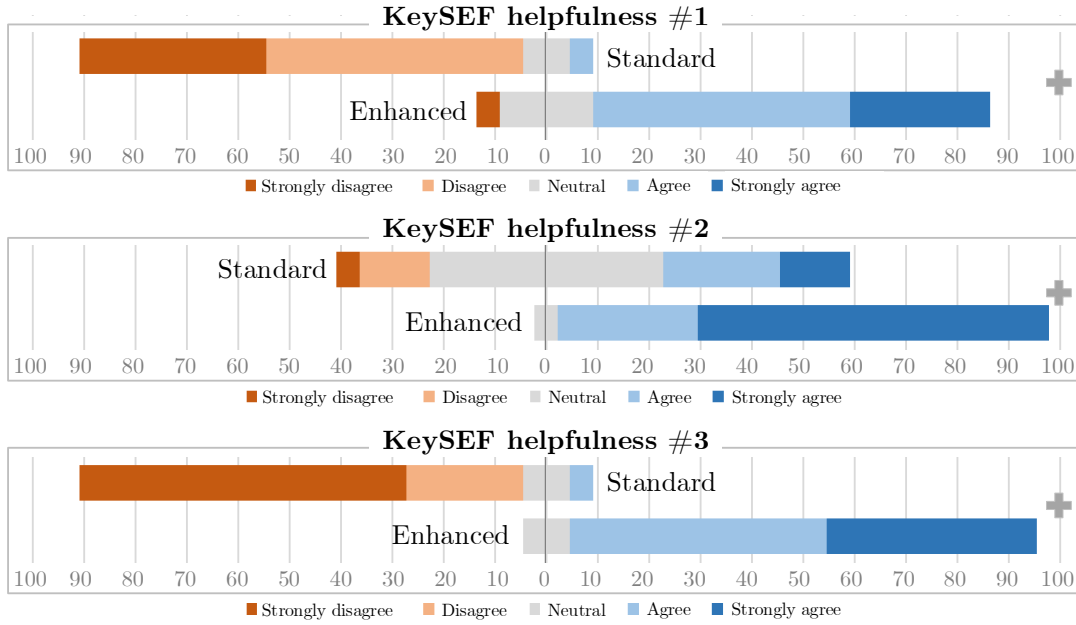


Figure 4.8: A comparison of the results, for the data obtained through KeySEF, for all metrics, on a percentage scale.

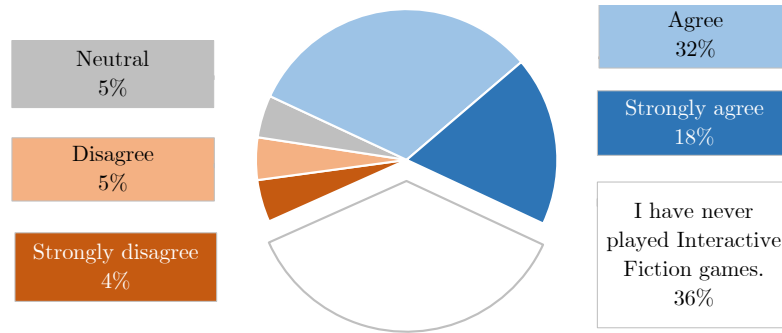


Figure 4.9: A pie chart showing the demographic components of the cohort, with respect to the statement: “*I enjoy playing IF games.*”.

disparity between the engines was successfully exposed due to the locality of evaluation – the Likert items posed through KeySEF are in the narrow context of the engine’s response immediately above them.

As part of the demographic questionnaire, data relating to the participants’ experience with IF games was collected. This data is presented in Figure 4.9 and is quite interesting to explore further. In Figure 4.10, the average marks for each engine were determined for the three generalised demographic categories of the participants. The data shows an interesting relation between the tree groups – they all have a different perception of the Standard engine which is consistent with their original preference. On the other hand, their views of the Enhanced engine are quite similar. This suggests that the enhancements made to the Standard engine using KR&R introduce a good balance to the game. With them included, all participants show similar levels of helpfulness, regardless of their previous experience with IF. This further supports H_a and the validity of the experiment.

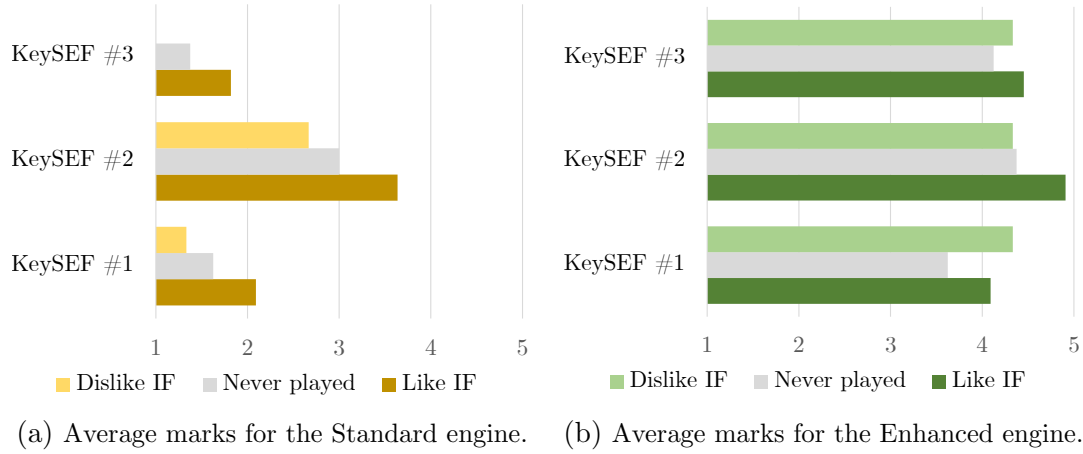


Figure 4.10: The average marks of each engine, depending on participants' previous experience with IF games. Category mapping (using Figure 4.9): neutral and lower \rightarrow Dislike IF, agree and higher \rightarrow Like IF, the rest Never played.

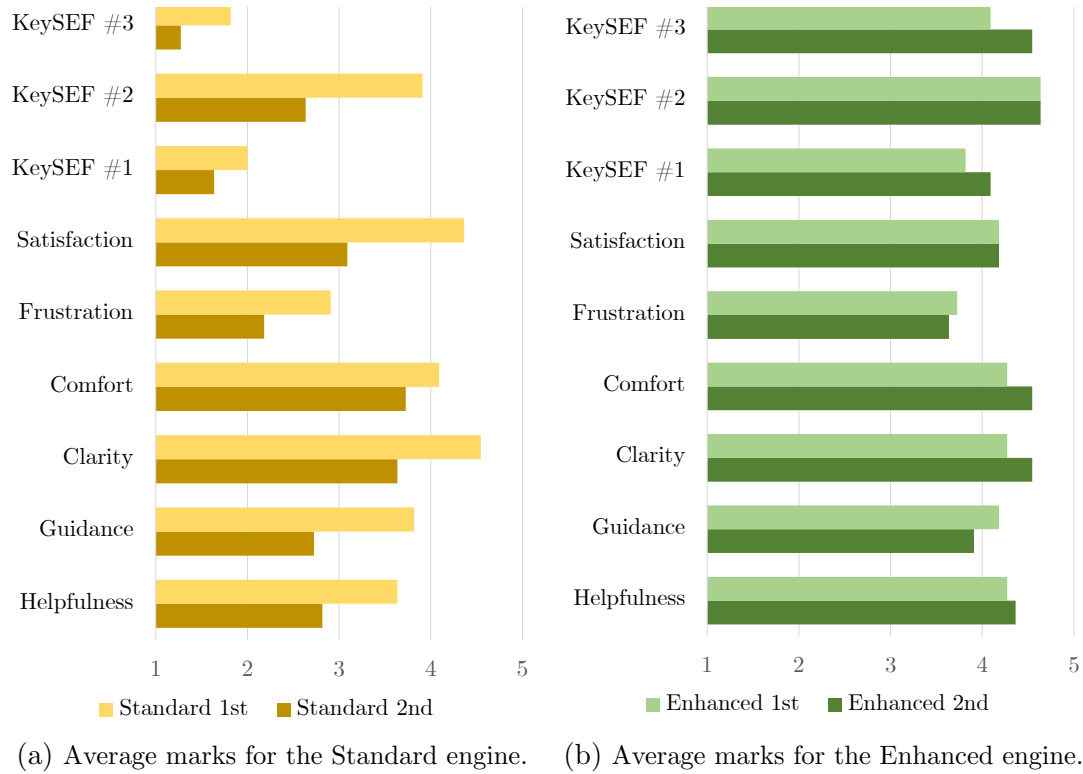


Figure 4.11: The average marks of each engine, depending on the session order. Mark mapping: 1 \rightarrow Strongly Disagree, 2 \rightarrow Disagree, etc.

Finally, in Figure 4.11, the average marks per engine were determined for both session orders. It is interesting to observe that the participants who played the standard engine first consistently rated it better compared to when they played it in their second session (4.11a). On the other hand, the order of play did not seem to have an impact on the rating of the Enhanced engine (4.11b). This further supports the claim that the experience in the Enhanced engine serves as a more stable metric than its Standard counterpart.

4.5 Summary

In this chapter, I have discussed how each of the components of the Interactive Fiction expert system was evaluated. I first presented the unit and play-testing techniques used to evaluate the NLP frontend and Compiler backend. Finally, I gave a comprehensive insight into the configuration and execution of the controlled experiment and provided conclusive evidence supporting the hypothesis that the Enhanced engine is indeed better than its Standard counterpart.

Chapter 5

Conclusion

Over the course of this dissertation, I have described the work undertaken to deliver a successful Interactive Fiction expert system. I started by introducing the motivation behind the project – making IF more enjoyable and easier to develop. I then presented the necessary preparatory work undertaken, with an emphasis on the understanding of Knowledge Representation and Reasoning – the core material needed for the proposed enhancements. Following this, I described in detail the implementation of the four major components of the system – the Standard and Enhanced engines, NLP frontend and Compiler backend. Finally, I presented a strong case for the success of the improvements made with the Enhanced engine, by means of a scientifically rigorous controlled experiment, which achieved compelling, statistically significant results across the board.

5.1 Achievements

The Interactive Fiction expert system was implemented to the full specification laid out in the project proposal, including the original extensions. Furthermore, the Enhanced engine was extended further than initially planned, with an even greater number of mutually-dependent physical parameters (§3.2.2.1). Similarly, other components of the system were enriched with extra features, such as the custom synonyms (§3.4.1) added to the grammar, the KeySEF (§4.4.1.2) and the hinting system included in the engine core (§4.4.1.4) and so on.

At the same time, the system created is coherent and incorporates several vastly different areas of computer science, including Knowledge Representation and Reasoning (§3.2.2), Finite Automata (§3.2.1), Natural Language Processing (§3.3), Context-free grammars and Compiler design (§3.4), and so on. I found this aspect of the project very valuable because through researching and developing the project itself, I found that my knowledge in these fields has substantially increased.

Finally, conclusive evaluation results show significant improvement, when the modified engine is compared to a model of a contemporary Interactive Fiction engine (§4.4.2). In particular, the enhancements have resulted in comparable levels of helpfulness for all players, regardless of past experience (§4.4.2.2).

5.2 Lessons Learnt

Even though the process of developing a system of this proportion had its challenges, the greatest ones for me were encountered during evaluation. Most notably, designing and executing a rigorous scientific experiment involving human participants, has proven itself to be far more challenging than originally anticipated. I believe this is primarily due to the fact that I have not had previous experience with such endeavours. Thankfully, the knowledge obtained in the Human-Computer Interaction course was very valuable in resolving this issue, as well as in designing the experiment itself. Similarly, I underestimated the time-cost of correct statistical analysis of the results obtained through the user study, for these same reasons.

In conclusion, the entirety of the dissertation experience, from start to finish, was incredibly exciting and rewarding.

5.3 Further Work

The Interactive Fiction expert system is complete, in the sense that it satisfies the specification set out in the project proposal and elaborated during the requirements analysis (§2.6). Of course, there is always room for improvement and the addition of further features. With this in mind, some interesting aspects which could be explored in the future are:

- **Story consistency checking** – Since the Story Language is deterministic by design, it would be possible to apply KR&R or planning algorithms to check if the story is *consistent* – whether all story steps can be reached via normal game-play.
- **Story Language upgrades** – The Story Language could be further developed to include additional game mechanics, as well as to be less cumbersome to use. Similarly, richer error reporting could be implemented here.
- **More physical parameters** – As discussed in §3.2.2.1, any number of additional parameters could be added to the physical model of the Enhanced engine. Of course, this would result in a more complex model, and thus, games which are potentially significantly more difficult than is reasonable.

Bibliography

- [1] Sangtae Ahn and Jeffrey A Fessler. “Standard errors of mean, variance, and standard deviation estimators”. In: *EECS Department, The University of Michigan* (2003), pp. 1–2.
- [2] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558609326.
- [3] Alexa Ray Correia. *BBC launches update of The Hitchhiker’s Guide To the Galaxy text adventure*. Mar. 2014. URL: <http://www.polygon.com/2014/3/10/5490738/bbc-launches-update-of-the-hitchhikers-guide-to-the-galaxy-text> (visited on 03/28/2017).
- [4] Questionnaire Development Documentation System University of Duisburg-Essen. *Questionnaires in LaTeX using the paperandpencil package*. URL: <http://johncanning.net/wp/?p=2126> (visited on 04/07/2017).
- [5] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
- [6] Richard Heiberger and Naomi Robbins. “Design of Diverging Stacked Bar Charts for Likert Scales and Other Applications”. In: *Journal of Statistical Software* 57.1 (2014), pp. 1–32. ISSN: 1548-7660. DOI: 10.18637/jss.v057.i05. URL: <https://www.jstatsoft.org/index.php/jss/article/view/v057i05>.
- [7] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *Proceedings of the National Academy of Sciences* 114.13 (2017), pp. 3521–3526. DOI: 10.1073/pnas.1611835114. eprint: <http://www.pnas.org/content/114/13/3521.full.pdf>. URL: <http://www.pnas.org/content/114/13/3521.abstract>.
- [8] Rensis Likert. “A technique for the measurement of attitudes.” In: *Archives of psychology* (1932).
- [9] Christopher D. Manning et al. “The Stanford CoreNLP Natural Language Processing Toolkit”. In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014, pp. 55–60. URL: <http://www.aclweb.org/anthology/P/P14/P14-5010>.
- [10] Steve Meretzky and Douglas Adams. *Hitchhiker’s Guide To the Galaxy 30th Anniversary Edition*. 1984. URL: <http://www.bbc.co.uk/programmes/articles/1g84m0sXpnNCv84GpN2PLZG/the-hitchhikers-guide-to-the-galaxy-game-30th-anniversary-edition> (visited on 03/24/2017).

- [11] Marvin Minsky. “A framework for representing knowledge”. In: *The psychology of computer vision* 73 (1975), pp. 211–277.
- [12] Nabil Mohammed, Ali Munassar, and A. Govardhan. “A Comparison Between Five Models Of Software Engineering”. In: *International Journal of Computer Science Issues* 7.5 (2010), pp. 95–101. URL: <http://www.IJCSI.org>.
- [13] Nick Montfort. “Toward a Theory of Interactive Fiction”. In: *IF Theory Reader* (2011), pp. 25–58.
- [14] Graham Nelson. *Inform*. URL: <http://inform7.com/> (visited on 03/24/2017).
- [15] Guillaume Nodet. *JLine*. URL: <https://github.com/jline/jline3> (visited on 03/28/2017).
- [16] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999.
- [17] M. J. Robberts. *TADS - the Text Adventure Development System, an Interactive Fiction authoring tool*. URL: <http://www.tads.org/> (visited on 03/24/2017).
- [18] Andrei A. Rusu et al. “Progressive Neural Networks”. In: *Computing Research Repository* abs/1606.04671 (2016). URL: <http://arxiv.org/abs/1606.04671>.
- [19] Student. “The Probable Error of a Mean”. In: *Biometrika* 6.1 (1908), pp. 1–25. ISSN: 00063444. DOI: 10.2307/2331554. URL: <http://dx.doi.org/10.2307/2331554>.
- [20] Kristina Toutanova et al. “Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network”. In: *Proceedings of HLT-NAACL* (2003), pp. 252–259.

Appendix A

Complete Story Grammar

```
grammar StoryGrammar;

// lexer rules
OPEN_PAREN_SHARP: '<'; CLOS_PAREN_SHARP: '>';
OPEN_PAREN_CURLY: '{'; CLOS_PAREN_CURLY: '}';
OPEN_PAREN_BLOCK: '['; CLOS_PAREN_BLOCK: ']';
OPEN_PAREN_ROUND: '('; CLOS_PAREN_ROUND: ')';
COMMA: ','; SEMICOLON: ';';
DOUBLEQUOT: '"'; MINUS: '-';
ATOZ: [a-z]+; NUMERIC: [0-9]+;
ALPHANUMERIC: [a-z0-9]+([a-z0-9 ])*;
QUOTED_TEXT: DOUBLEQUOT (~('"'))* DOUBLEQUOT;
ID: OPEN_PAREN_BLOCK ALPHANUMERIC CLOS_PAREN_BLOCK;
TIME: OPEN_PAREN_ROUND NUMERIC CLOS_PAREN_ROUND;
SYNONYM: OPEN_PAREN_SHARP ATOZ CLOS_PAREN_SHARP;
WS: [ \n\t\r\u000C]+ -> skip;
VALUE: NUMERIC;

// main rule identifiers
MESSAGE_: '_message';
WELCOME_: '_welcome';
ROOM_: '_room';
ITEM_: '_item';
SPECIAL_: '_special_command';
STEP_: '_step';

// room related cardinal directions
N: '_N'; S: '_S';
E: '_E'; W: '_W';

// item specific
TAKEABLE: '_takeable';
```

```

FIXED:      '_fixed';
INVENTORY:  '_inv';
PRODUCED:   '_prod'; // item is not in game yet
IN_ROOM:    '_inroom';
IN_CONTAINER: '_incont';
IS_CONTAINER: '_iscont';
IS_ITEM:     '_isitem';
// parameter fields
/*state field*/      /*temperature field*/
STATEID:  '_state';  TEMPID:  '_temp';
SOLID:    '_solid';  HOT:      '_hot';
LIQUID:   '_liquid'; WARM:     '_warm';
GAS:      '_gaseous'; NORMAL:  '_normal';
/*mass field*/      COLD:      '_cold';
MASSID:   '_mass';  FROZEN:  '_frozen';
MIN:      '_min';   VARIABLE: '_variable';
MAX:      '_max';   CONSTANT: '_constant';
EQUAL:    '_equal'; /*volume field*/
SURPRESS: '_surpress'; VOLUMEID: '_vol';

// story step specific
ANDING: OPEN_PAREN_BLOCK 'and' CLOS_PAREN_BLOCK; // and
ORING:  OPEN_PAREN_BLOCK 'or' CLOS_PAREN_BLOCK;   // or
// possible condition types
PLAYER_IN_ROOM:      '_plir';    PLAYER_NOT_IN_ROOM:      '_plnir';
ITEM_IN_ROOM:         '_itir';    ITEM_NOT_IN_ROOM:         '_itnir';
ITEM_IN_INVENTORY:    '_itinv';    ITEM_NOT_IN_INVENTORY:    '_itninv';
ITEM_IN_CONTAINER:    '_iticon';    ITEM_NOT_IN_CONTAINER:    '_itnicon';
ITEM_IS_FROZEN:       '_isfrozen'; ITEM_IS_NOT_FROZEN:       '_isnotfrozen';
ITEM_IS_COLD:         '_iscold';    ITEM_IS_NOT_COLD:         '_isnotcold';
ITEM_IS_NORMAL:       '_isnormal'; ITEM_IS_NOT_NORMAL:       '_isnotnormal';
ITEM_IS_WARM:         '_iswarm';    ITEM_IS_NOT_WARM:         '_isnotwarm';
ITEM_IS_HOT:          '_ishot';     ITEM_IS_NOT_HOT:          '_isnothot';
PLAYER_ON_LEVEL:      '_plilv';

// previous command conditions
CON_COMBINE:          '_combine';  CON_MOVE:                  '_move';
CON_EXAMINE:          '_examine';  CON_SPECIAL:                '_special';
CON_USE:              '_use';       CON_PUTIN:                  '_putin';
CON_USEON:            '_useon';

// consequence types
WAIT:                 '_wait';      ADD_ITEM_TO_ROOM:          '_additr';
TELEPORT:             '_jmp';        ADD_ITEM_TO_INV:           '_additinv';
KILL:                 '_kill';       ADD_ITEM_TO_CONTAINER:     '_additcont';
WIN:                  '_win';        ADD_CONNECTOR:             '_addcon';

```



```

REMOVE_ITEM:      '_rmit';      REMOVE_CONNECTOR:      '_rmcon';
NONE:             '_none';

```

```

// parser rules

```

```

level_id:  ID;
item_id:   ID;
room_id:   ID;
step_id:   ID;
message_id: ID;
message_text: QUOTED_TEXT;
command: SYNONYM;
story_elements: welcome (room|item|message|special|step)*;
welcome: WELCOME_
        OPEN_PAREN_CURLY
            step_id SEMICOLON
            room_id SEMICOLON
            description SEMICOLON
        CLOS_PAREN_CURLY;
message: MESSAGE_
        OPEN_PAREN_CURLY
            message_id SEMICOLON
            message_text SEMICOLON
        CLOS_PAREN_CURLY;
room: ROOM_
      OPEN_PAREN_CURLY
          room_id SEMICOLON
          level_id SEMICOLON
          exits SEMICOLON
          brief SEMICOLON
          description SEMICOLON
      CLOS_PAREN_CURLY;
exits: exit (COMMA exit)*;
exit: direction room_id
     | direction MINUS description;
direction: N|E|W|S;
brief: message_text
     | message_id;
description: message_text
           | message_id;
item: ITEM_
     OPEN_PAREN_CURLY
         item_id SEMICOLON
         mobility SEMICOLON
         location SEMICOLON

```

```

        parameter_fields
        description SEMICOLON
    CLOS_PAREN_CURLY;
mobility: TAKEABLE
    | FIXED;
location: IN_ROOM room_id
    | INVENTORY
    | PRODUCED
    | IN_CONTAINER item_id;
itemtype: IS_CONTAINER COMMA holding_type COMMA holding_mass
    | IS_ITEM;
parameter_fields: (itemtype SEMICOLON | )
    (volume_field SEMICOLON | )
    (mass_field SEMICOLON | )
    (temp_field SEMICOLON | )
    (state_field SEMICOLON | );
volume_field: VOLUMEID VALUE;
holding_type: MIN | EQUAL | MAX;
holding_mass: VALUE;
mass_field: MASSID SURPRESS VALUE
    | MASSID VALUE;
temp_field: TEMPID temp_level COMMA temp_variability;
temp_level: FROZEN | COLD | NORMAL | WARM | HOT;
temp_variability: CONSTANT | VARIABLE;
state_field: STATEID scale_field
    ((COMMA scale_field) | )
    ((COMMA scale_field) | );
scale_field: OPEN_PAREN_SHARP
    temp_level COMMA
    state_level
    CLOS_PAREN_SHARP;
state_level: SOLID | LIQUID | GAS;
special_id: ID;
special: SPECIAL_
    OPEN_PAREN_CURLY
        special_id SEMICOLON
    CLOS_PAREN_CURLY;
hint: description;
step: STEP_
    OPEN_PAREN_CURLY
        step_id SEMICOLON
        gate_type SEMICOLON
        required_steps SEMICOLON
        conditions SEMICOLON

```

```

        consequences SEMICOLON
        description SEMICOLON
        (hint SEMICOLON | )
    CLOS_PAREN_CURLY;
step_before: step_id TIME;
required_steps: step_before (COMMA step_before)*;
conditions: condition (COMMA condition)*;
condition: single_arg_cnd
    | double_arg_cnd
    | CON_MOVE direction;
single_arg_cnd: single_arg_cnd_type extra_synonyms_single item_id;
single_arg_cnd_type: PLAYER_IN_ROOM      | CON_EXAMINE
    | PLAYER_NOT_IN_ROOM | CON_USE
    | PLAYER_ON_LEVEL    | CON_SPECIAL
    | ITEM_IN_INVENTORY  | ITEM_NOT_IN_INVENTORY
    | ITEM_IS_FROZEN     | ITEM_IS_NOT_FROZEN
    | ITEM_IS_COLD       | ITEM_IS_NOT_COLD
    | ITEM_IS_NORMAL     | ITEM_IS_NOT_NORMAL
    | ITEM_IS_WARM       | ITEM_IS_NOT_WARM
    | ITEM_IS_HOT        | ITEM_IS_NOT_HOT;
extra_synonyms_single: (command
    | OPEN_PAREN_CURLY
        command
        (COMMA command)+
        CLOS_PAREN_CURLY
    | );
double_arg_cnd: double_arg_cnd_type extra_synonyms_double item_id room_id;
double_arg_cnd_type: ITEM_IN_ROOM      | CON_COMBINE
    | ITEM_NOT_IN_ROOM | CON_USEON
    | ITEM_IN_CONTAINER | CON_PUTIN
    | ITEM_NOT_IN_CONTAINER;
connector: SYNONYM;
double_command: OPEN_PAREN_ROUND
    command
    (COMMA connector)+
    CLOS_PAREN_ROUND;
extra_synonyms_double: (double_command
    | OPEN_PAREN_CURLY
        double_command
        (COMMA double_command)+
        CLOS_PAREN_CURLY
    | );
consequences: consequence (COMMA consequence)*;
consequence: no_arg_cons

```

```

    | single_arg_cons
    | double_arg_cons
    | four_arg_cons
    | WAIT TIME;
no_arg_cons: NONE | KILL | WIN;           // no arguments
single_arg_cons: single_arg_cons_type item_id; // single argument
single_arg_cons_type: TELEPORT
    | ADD_ITEM_TO_INV
    | REMOVE_ITEM;
double_arg_cons: double_arg_cons_type item_id room_id; // two arguments
double_arg_cons_type: ADD_ITEM_TO_ROOM
    | ADD_ITEM_TO_CONTAINER;
four_arg_cons: four_arg_cons_type room_id direction room_id direction;
four_arg_cons_type: ADD_CONNECTOR
    | REMOVE_CONNECTOR;
gate_type: ANDING | ORING;

```

Appendix B

NLPtest – Example Batch File

B.1 Test Input

Take the laser.
Pick up the laser.
Claim the beam.
Try to collect the laser
Drop the beam.
Leave the laser behind.
Quickly examine the laser beam!
Inspect that beam over there.
Observe the shiny laser.
Utilise just the laser.
Use the laser on the beam.
Merge laser with beam.
Look around the room.
Show my inventory.
Wait for some time to pass.
Display command history.
Combine that laser with this beam.
Try to combine the laser and beam.
Try combining beam with the laser.
Apply the laser beam to the laser.
Please employ laser with laser beam.
Try using laser with laser beam.
Make a move north.
Try moving towards east.
I wish to move south!
Please move west.
Meld laser beam.
Do not use the laser.
Please place the laser in the beam.
May I put the laser in the beam?
I insert the laser into the beam.
Remove the laser from the beam.
Obtain the laser from the laser beam.
Restart the game.
Quit.

B.2 Expected Output

take [laser]
take [laser]
take [beam]
take [laser]
drop [beam]
drop [laser]
examine [laser beam]
examine [beam]
examine [laser]
use [laser]
useon [laser] [beam]
combine [laser] [beam]
look
inventory
wait
history
combine [laser] [beam]
combine [laser] [beam]
combine [beam] [laser]
useon [laser beam] [laser]
useon [laser] [laser beam]
useon [laser] [laser beam]
move [n]
move [e]
move [s]
move [w]
badcomm
badcomm
putin [laser] [beam]
putin [laser] [beam]
putin [laser] [beam]
remove [laser] [beam]
remove [laser] [laser beam]
restart
exit

Interactive Fiction Game Session executed on the Enhanced Engine

[illegible]

63

notice a dusty metal fusebox in close proximity of the freezer. You are not sure what it could house inside, because it too is closed, but you think that it could somehow help you in solving the power issue. As you glance over the fireplace again, you notice two strange indentations in its front facing wall: a hexagonal lock and a rectangular hole. The former is rather small and dusty, and looks to be carefully carved in the brick face. The latter has a loose brick sticking out of it and is scruffy around the edges. That loose brick seems to you like it could come in handy, so you plan to start by taking it...

There also seems to be an interesting window, seemingly looking out the west side of the room, but there is something off about it. It looks to you like the image flickers every now and then, as if it were a screen instead of a window. Either way, it gives you a sense of unease and you feel watched.

Suddenly, you feel a shake in the ground and the whole manor stutters for a second. The chandelier is still swinging slightly, looking even more unstable now. You feel that this place could collapse any minute, and this brings with it a certain sense of anxiety. Adrenaline is still rushing through your veins, and you feel that you really should get on with somehow getting out of here...

> Collect the brick.

You remove the warm loose brick from the rectangular hole. You have a brick now, yay!

```

      \      |      /
      \-----/
      /-----/\
      /-----//
  - - |      \ \ - -
      |  B R I C /
      |-----\
      /      |      \

```

As the light enters the hole in the fireplace, you notice that there is something else inside, lying at the far end. You consider examining the rectangular hole to see what it is.

> Investigate the hole.

```

~--'-----'--~
~ | /          | ~   A brick-sized hole in the
~ | /          | ~   fireplace's front-facing wall.
~--'-----'--~

```

Contains the following:

- warm solid golden key

The rectangular hole is always warm.

> Take the key from the hole.

You remove the warm solid golden key from the rectangular hole.
It glimmers under the light of the fireplace.

```

      \      |      /
      \-----/
      /-----\
      /-----\
  -  \ \ \  \ \ \ \ \ \ - -
      / / /  / / / / / /
      \ \ \  / / /
      /-----\
      /      |      \

```


> Try to insert the key into the lock.

The warm solid golden key is too big to fit into the hexagonal lock.

> I wish to wait.

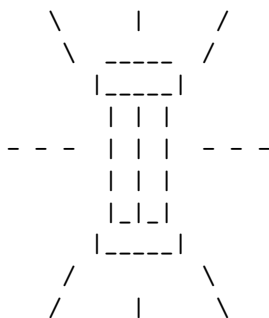
Time passes.

click *click* *click*

You hear a strange clicking noise, and notice that a minor hole has opened in the ceiling. You can make out a small object being lowered through it on a thin silver string. You watch in a combination of fear and excitement as it comes down, and wait to see what will happen next. To your disappointment, the little object softly touches down on the ground. The string then detaches and swiftly returns to its ceiling hole, which closes seamlessly behind it. As it is now a lot closer to you, you realise that the small object is in fact an old fuse! You feel oddly lucky, but have a hard time believing that this was just a mere coincidence. Oh well, better act now and ask questions later, you think to yourself.

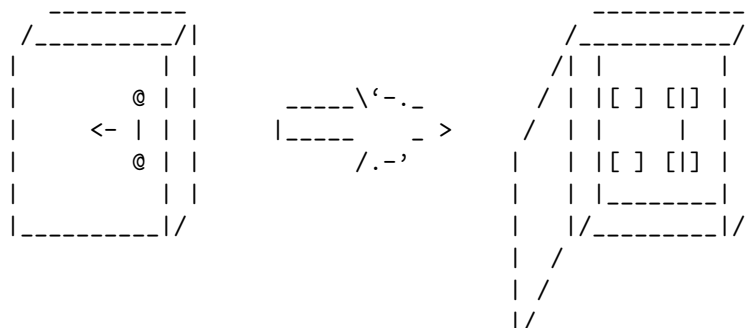
> Pick up the old fuse.

You take the cold old fuse.
It's a bit oldfashioned, but it should do the trick.



> Open the fusebox.

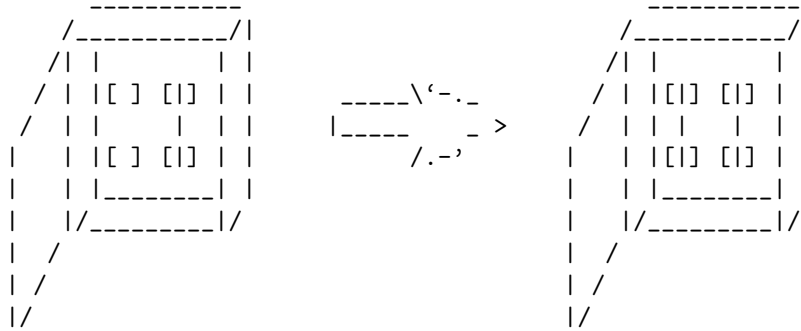
You find a way to grip the front panel and pull strongly. The front panel flies open, accompanied by a loud, scratching noise.



> Place the fuse in the fusebox.

You place the cold old fuse in the open fusebox, and now the old fuse is at normal temperature and volume.

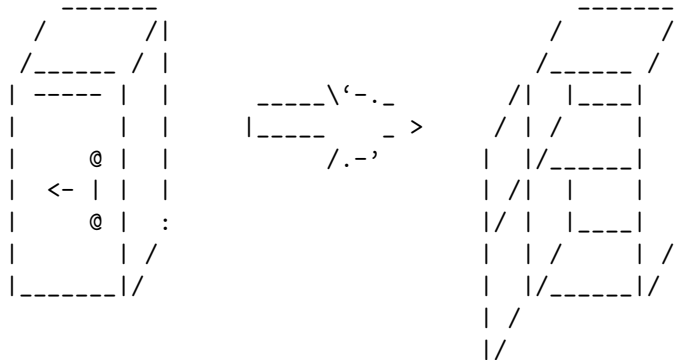
You hear a fritzy sound of electronics coming to life.



Soon after the freezer turns on and hums away while it cools down to its working temperature.

> Open the freezer.

You find a way to grip the door, and after a little bit of elbow grease the freezer is wide open.



You are surprised by just how cold the insides of this thing are.

> Put the key into the freezer.

You put the warm solid golden key into the open freezer, and now the golden key is frozen.

The volume of the golden key has decreased to 92% its normal size.

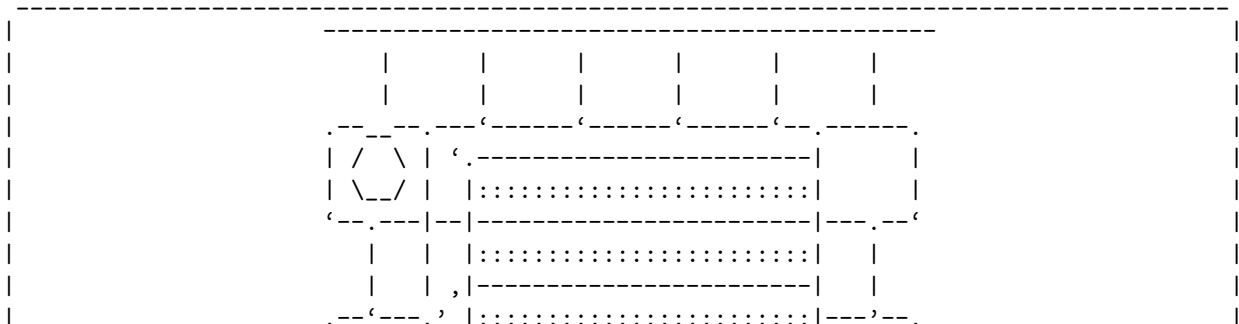
> Take back the key.

You remove the frozen solid golden key from the open freezer.

> Insert the key into the hexagonal lock again.

You insert the frozen solid golden key into the hexagonal lock, and now the golden key is warm.

The volume of the golden key has increased to 104% its normal size.



Appendix D

Project Proposal

Milos Stanojevic
Trinity
ms2239

Computer Science Tripos - Part II - Project Proposal

Interactive Fiction

20 October 2016

Project Originator:

Dr Sean Holden

Project Supervisor:

Dr Sean Holden

Director of Studies:

Dr Sean Holden

Project Overseers:

Dr T. G. Griffin

Dr Pietro Liò

Introduction and Description of the Work

Interactive fiction (IF) is a category of gaming software which falls under the adventure genre. Works in this category usually present players with an environment, which they can then interact with and influence using text commands. It first became popular during the 80's, however it has not matured much in complexity since.

Possibly the biggest issue with IF at the moment is that there is a limited set of responses it can give to player input. This is mostly because standard IF game engines consist of a reasonably simple parser and a 'story state machine'. Here, the parser is capable of interpreting words from a pre-determined set, defined by the creator. Similarly, the state machine is also hard-coded, and is only as complex as the creator makes it. This means that an IF game responds with a dull 'I dont understand' to any unexpected input, that was not implemented as a transition in the state machine.

This project is thus intended to apply Knowledge Representation and Reasoning (KR&R) to IF in an attempt to improve its playability. The idea is to use general inference at the core of the engine to allow for more creative responses.

To give an example, a standard IF engine would respond to the query 'put the elephant in the fridge' with 'I don't understand'. However, with the use of KR&R, the response would for example be 'the elephant is too big to fit in the fridge'. The idea explored here is that certain physical aspects of the real world, such as volume or mass, would be quantised in a knowledge base in the IF engine. This Knowledge base would then be consulted using general inference every time a new query is submitted. More details of the exact plan of implementation are given in the **Substance and Structure of the Project** section.

Starting Point

Since the implementation of the parser is out of the scope of the project, the public domain Stanford CoreNLP library will be used here. The resulting parse trees obtained from CoreNLP will then be processed and utilised by the game engine. Only one more library is planned, and that is the Princeton WordNet Lexical database for English whose intended use will be to recognise synonyms. This should allow for a broader range of user input to be correctly interpreted. Both of the mentioned libraries will only be interfaced with and the remaining components of the system will be implemented from scratch.

In addition to this, the knowledge gained in the Part IB Artificial Intelligence course will be of great use to the project, since the basic concepts of Knowledge Representation and Reasoning were covered there. The Part II Natural Language Processing and the Part IB Compilers courses will be useful for utilizing and understanding the inner-workings of the parser component (intended as a library). The Part II Human Computer Interaction course will be important for the correct execution of the Comparative Study at a later stage of the project. Finally, the part IA and IB courses on Java, Algorithms and C/C++ will be useful as the knowledge gained there directly applies to the implementation aspect of the project.

Substance and Structure of the Project

The planned components of the system are:

1. The Parser - This component will be created by interfacing with the Stanford CoreNLP and the Princeton WordNet libraries. It will most likely be a Java wrapper for the aforementioned libraries. Its intended use is to extract a single statement from the user input (e.g. from the sentence 'please examine the green monkey' it should extract 'examine green monkey').
2. The Standard Engine - This component will represent a standard IF engine with no integrated AI components. The main purpose of The Standard Engine is to be the initial bare-bones engine that can be separately tested. It will be used as a base for comparison later, when The Inference Module is created. It is intended to be implemented in C++ or Java, which will be determined later.
3. The Inference Module - This component is at the core of the project and where most of the complexity will be situated. It is the upgrade for The Standard Engine, and will introduce Knowledge Representation and Reasoning. It will introduce a single physical parameter to every object and will be concerned with their physical position.
4. The Story Editor - This component will allow the creation of a custom IF story file, which would then later be executed using the engines. Should the story file be intended for use with the Inference Module, the Story Editor would also allow the specification of the physical parameter for objects in the environment. It is intended to take input in textual format (possibly using a custom pseudo-language).

In terms of the system flow as well as the finer aspects of execution, the complete system would behave as follows:

The parser component would receive user input and process it using the external libraries to produce a parse tree. It would then internally analyse the obtained tree and extract statements from it (e.g. 'take cat' or 'push green button'). Here the synonyms library will be utilised to obtain as wide an understanding as possible.

These statements would then be handed over to the Engine. In case of the Standard engine, the statement would be used in an attempt to advance to a new state in the story state machine. Here, in case a transition matches the statement, the story advances and the appropriate text is printed. Otherwise, the output of the engine would be 'I do not understand.'

On the other hand, if the Standard Engine is enhanced with the Inference Module, the attempting to advance the story state machine would only be the first step. Then, general inference would be used on the statement and the story knowledge base. Regardless of the success or failure of the inference procedure, an informative message (within reason) would be assembled using a dictionary and returned to the user. For example, if the input is 'put the cat in the matchbox', the output would be 'the cat is too big to fit in the matchbox'.

The Inference Module is planned to have a single physical parameter in its main form. As an extension, a second physical parameter might be added to the Inference Module. The additional physical parameter may interfere with the original one, for example 'use freeze-gun on Yeti' would result in 'The Yeti is now frozen, and has reduced in volume'.

Finally, the project consists of the following main sections:

1. A study of the parser libraries and further research into the finer details of Knowledge Representation and Reasoning.
2. Developing and testing the system components outlined earlier in this section.
3. Evaluation the quality of the improvements made to the playability of IF, by conducting a Comparative Study experiment involving human participants. The experiment would involve participants playing two identical games, one executed with the Standard Engine, and one executed with the Inference Module incorporated, and then filling out a carefully devised questionnaire. More details of the Comparative Study are given in the **Success Criteria**.
4. Writing the dissertation.

Success Criteria

The following should be achieved:

- Implement and create unit tests for the Parser
- Implement the Story Editor and successfully construct and store a simple story file
- Implement the Standard Engine and the Inference Module, and test them on human participants in the Comparative Study (details outlined below)

The Comparative Study experiment would be performed as follows. Participants would be sat in front of a computer and given two identical games to play, one executing on the Standard Engine, and one executing on the Standard Engine with the Inference Module incorporated. They would then be asked to fill out a carefully devised questionnaire. The questionnaire would allow them to contrast the game-play experiences, and give a numerical verdict on the overall improvement that the Inference Module made, or the lack of it.

Before the Comparative Study is performed, several pilot runs of it will be executed. These will serve to test the design of the questionnaire. Finally, the success of the project will not depend on the results produced by the Comparative Study.

Resources Required

A set of libraries will be required for the Parser. The Stanford CoreNLP library, is available for download from <http://stanfordnlp.github.io/CoreNLP/>. The Princeton

WordNet Lexical database for English is available from <https://wordnet.princeton.edu/>. Both of these libraries belong to the public domain.

A personal Asus PC (2.40GHz Intel(R) Core(TM) i7-4700HQ CPU, 16.0GB RAM and 500GB Disk) will be used for primary development. I accept full responsibility for this machine and I have made the following contingency plans to protect myself against hardware/software failures:

- A personal GitHub account will be used for revision control of the projects codebase and as a means of online backup.
- A personal Google Drive and Dropbox account will be used for regular synchronisation as means of online backup.
- A personal hard drive (1.36TB) will be used for regular primary backup of the entire machine as well as the project files themselves utilising a personal copy of Acronis True Image.

The Comparative Study which will be performed at a later stage in the project will involve human subjects. I will need to obtain permission from the Computer Lab's Ethical Committee for this.

Timetable and Milestones

- Slot 1: *20 October – 3 November*
 - Finalise technicalities, and get approval from the Computer Lab's Ethics Committee to perform the Comparative Study
 - Fully investigate the capabilities of the libraries
 - Start work on the Standard Engine

Milestone: Various bits and pieces of code, nothing complete yet

- Slot 2: *3 November – 17 November*
 - Start work on the Parser
 - Start work on the Story Editor
 - Complete a very basic version of the Standard Engine

Milestone: Incomplete Parser and Story Editor, very basic Standard Engine

- Slot 3: *17 November – 1 December*
 - Finalise work on the Parser and the Story Editor
 - Connect all three components
 - Start work on the Inference Module

Milestone: Parser, Standard Engine and Story Editor created in a basic form and connected

- Slot 4: *1 December – 15 December*
 - Finalise the Inference Module
 - Debug and build up the rest of the system

Milestone: Functioning Inference Module (core development completion)

- Slot 5: *15 December – 29 December*
 - Start work on the upgrade to the Inference Module (addition of another physical parameter)
 - Polish rest of code as needed

Milestone: Fully functional core system and bits of the upgrade implemented

- Slot 6: *29 December – 12 January*
 - Write draft Progress Report and Presentation
 - Start work on Comparative Study
 - Finalise and debug the extension to the Inference Module (stretch-goal completion)

Functioning upgrade to Inference Module **Milestone:**

- Slot 7: *12 January – 26 January*
 - Prepare the questionnaire for the Comparative Study
 - Run pilot study on friends and analyse results in preparation for the real thing
 - Add obtained results to the Progress Report and submit it
 - Add obtained results to the Presentation and submit it

Milestone: Questionnaire ready, pilot study performed and documents submitted

- Slot 8: *26 January – 9 February*
 - Perform the Comparative Study and analyse its results
 - Perform the presentation in the second week of the slot

Milestone: Comparative Study executed, data obtained and analysed

- Slot 9: *9 February – 23 February*
 - First buffer zone, only to be used if the previous stages are overrunning
 - If all is in order, start work on the dissertation early

Milestone: None

- Slot 10: *23 February – 9 March*
 - Start work on the dissertation

Milestone: Introduction and preparation chapters completed

- Slot 11: *9 March – 23 March*
 - Complete first draft of dissertation
 - Submit the initial draft to the supervisor

Milestone: Implementation, Evaluation and Conclusion chapters completed

- Slot 12: *23 March – 6 April*
 - Create second draft using guidance obtained from responses
 - Submit the second draft to the supervisor

Milestone: Second draft submitted

- Slot 13: *6 April – 20 April*
 - Incorporate any further guidance given by the supervisor into the final version of the dissertation as appropriate
 - Send the final version to the supervisor for a final check

Milestone: Final version of the dissertation ready for submission

- Slot 14: *20 April – 4 May*
 - Perform final checks and submit the dissertation

Milestone: Dissertation Submission

- Slot 15: *4 May – 18 May*
 - Buffer slot to accommodate for any unexpected trouble with writing the dissertation.

Milestone: Dissertation Submission Deadline (19 May)