Arkham Horror: Specification and Implementation

Alex Altman
B.A. (Mod.) CS
Final Year Project, April 2011
Supervisor: Dr. Arthur Hughes
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

________________________________________ April 11, 2011
Alex Altman
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

______________________________  April 11, 2011
Alex Altman
Acknowledgements

Thanks first of all to Dr. Arthur Hughes of the Computer Science department for suggesting this project to me and for supporting me throughout its lifetime. On a similar note, my thanks to my family, for supporting me through constant nagging.

Thanks also to Stephen Kenneally, for being such an incredible H.P. Lovecraft fan that he had to play, own, or just know about anything to do with the author, resulting in him hearing of and subsequently playing this game with myself and my friends.

Further thanks to the Department as a whole and other assistance, formal and informal, received from the University, as, without all the help I received throughout the years, I would probably not be here writing my thesis.

Finally, thanks to Fantasy Flight Games for making this game, which has provided me with many hours of amusement that could have been better spent sleeping, rather than swearing loudly at inanimate pieces of card.
The world is indeed comic, but the joke is on mankind. - H.P. Lovecraft
## Contents

1 Introduction .................................................. 1  
  1.1 Overview ................................................. 1  
  1.2 Objectives of Project ................................... 1  
  1.3 Report Outline ........................................... 2  

2 A Brief Introduction to Software Engineering .......... 3  
  2.1 History of Software Engineering ....................... 3  
  2.2 Software Engineering Processes ....................... 3  
  2.3 Software Engineering Models ........................... 4  
      2.3.1 Waterfall Model .................................. 4  
      2.3.2 Iterative Development ............................ 5  
  2.4 Summary ................................................ 6  

3 Basic Formal Methods ........................................ 7  
  3.1 History of Formal Methods .............................. 7  
  3.2 “Myths” of Formal Methods .............................. 8  
  3.3 Specification Languages of Formal Methods .......... 9  
      3.3.1 Algebraic Approach ............................... 10  
      3.3.2 Model-Based Approach ............................ 10  
  3.4 Summary ................................................ 11  

4 The Basics of Z ............................................... 12  
  4.1 Description of Z ......................................... 12  
  4.2 Basic Z State Schema: a Playlist ....................... 13  
  4.3 The Initialisation Schema For a Playlist .............. 13  
  4.4 Operational Schemas For the Playlist ................. 14  
  4.5 Summary ................................................ 15  

5 An Introduction to Arkham Horror ......................... 16  
  5.1 Basic Rules and Goals ................................... 16  
  5.2 The Board ............................................... 17  
      5.2.1 Neighbourhoods ................................... 18  
      5.2.2 Other Worlds ..................................... 20
5.2.3 The City Limits ......................................................... 21
5.2.4 The Terror Track ...................................................... 21
5.3 The Great Old One ....................................................... 23
5.4 Other Items and Characters .......................................... 26
5.5 Summary ................................................................. 31

6 Example 1: Monster Design and Code .................................. 32
   6.1 Description of the Monster In-Game ................................. 32
   6.2 Formal Design of the Monster ........................................ 35
      6.2.1 State Schemas .................................................... 36
      6.2.2 Initialisation Schemas ........................................... 37
      6.2.3 Operational Schema .............................................. 39
   6.3 Java Code .............................................................. 39
   6.4 Summary ............................................................... 44

7 Example 2: Investigator Design and Code ............................... 45
   7.1 A Description of the Investigator ................................... 45
   7.2 Formal Design of the Investigator .................................. 48
      7.2.1 Investigator State Schemas ..................................... 49
      7.2.2 Initialisation Schemas .......................................... 51
      7.2.3 Operational Schemas for the Investigator ..................... 52
   7.3 The Investigator Implementation ..................................... 57
   7.4 Summary ............................................................... 67

8 A Description of the Chat System ....................................... 68
   8.1 Authorisation Server .................................................. 68
   8.2 Chat Server ............................................................ 68
   8.3 Client ................................................................. 69
   8.4 How It Works ............................................................ 70
   8.5 Summary ............................................................... 71

9 Conclusions ....................................................................... 72
   9.1 Benefits of Using Z ..................................................... 72
   9.2 Drawbacks of Using Z .................................................... 73
   9.3 Consideration of Hall’s “Myths” ....................................... 74
   9.4 Future Work .............................................................. 74
   9.5 Final Thoughts ........................................................... 75

Bibliography ...................................................................... 76

A Schemas In Full ................................................................ 78
   A.1 Useful Free Variables .................................................... 78
   A.2 Nodes, Locations, Streets, and Others ............................... 78
      A.2.1 State Schemas ......................................................... 78
List of Figures

2.1 Waterfall model .................................................. 5
2.2 Iterative development model .................................. 6

3.1 Template algebraic structure ............................... 10

5.1 The Board ......................................................... 18
5.2 Detail of the Merchant District ............................... 19
5.3 Detail of the Dreamlands ..................................... 20
5.4 Detail of the City Limits sections ............................ 21
5.5 Detail of the Terror Track ..................................... 22
5.6 Great Old One sheet for Ithaqua ............................. 23
5.7 Detail of Ithaqua’s worshippers and combat rating ...... 24
5.8 Detail of Ithaqua’s defences (none) and his attack ...... 24
5.9 Detail of Ithaqua’s power, Icy Winds ........................ 25
5.10 Detail of Ithaqua’s Doom Track .............................. 25
5.11 Encounter card for the Merchant District ................. 26
5.12 Yellow Other World Encounter card ....................... 27
5.13 The Mythos card “Picnickers Panic!” ...................... 27
5.14 A Common Item: the .45 Automatic weapon ............ 28
5.15 A Unique Item: the Elder Sign ............................... 29
5.16 A Skill card: +1 Will ........................................... 29
5.17 A Spell: Dread Curse of Azathoth .......................... 30
5.18 An Ally: Professor Armitage ................................ 30

6.1 Movement side of the Dhole ................................. 33
6.2 Combat side of the Dhole ....................................... 34

7.1 The Investigator sheet for Amanda Sharpe ................. 46
7.2 Detail of Amanda Sharpe’s Statistics ........................ 46
7.3 Detail of Amanda Sharpe’s ability, Studious ............... 47
7.4 Detail of Amanda Sharpe’s starting possessions and home location . 47
7.5 Detail of Amanda Sharpe’s skill tracks ..................... 48

8.1 The Chat Client GUI ............................................... 70
# Listings

6.1 Movement enumerator ........................................... 39
6.2 Abilities enumerator ........................................... 39
6.3 Symbol enumerator ............................................. 40
6.4 Horror and Combat data types ................................. 40
6.5 Monster private attributes ..................................... 41
6.6 Monster constructor declaration .............................. 41
6.7 Monster constructor part 1: preventing null assignation ... 42
6.8 Monster constructor part 2: enforcing numerical constraints ... 42
6.9 Monster constructor part 4: checking Abilities array ....... 42
6.10 Monster constructor part 5: everything else ................. 43
6.11 Method to simulate an injective sequence and other constraints ... 43
6.12 An example access method .................................... 44
6.13 The setLocation method ...................................... 44
7.1 Skillpair attributes and constructor ........................ 57
7.2 Method to retrieve the current Skill levels ................ 57
7.3 Method to change the Pointer value .......................... 58
7.4 Investigator attributes .......................................... 58
7.5 Investigator constructor declaration .......................... 58
7.6 Constructor part 1: checking if any values are null ....... 59
7.7 Constructor part 2: ensuring values do not fall out of range ... 59
7.8 Constructor part 3: unconventional assignation ............. 60
7.9 Getting the value of a statistic ............................... 60
7.10 Getting a specific pair of Skills .............................. 61
7.11 Getting the current value of a specific Skill ............... 61
7.12 Returning a list of Common Items held by the Investigator ... 62
7.13 First part of focus method ................................... 62
7.14 Changing statistics ............................................. 63
7.15 Blessing or cursing an Investigator ........................... 63
7.16 Delaying and moving .......................................... 63
7.17 Getting a new Monster Trophy ............................... 64
7.18 Spending Monster Trophies .................................... 64
7.19 Getting and paying off loans .................................. 65
7.20 Moving the Investigator ....................................... 65
7.21 Resetting the Investigator at the beginning of a round .................................. 66
7.22 Checking retainers .......................................................................................... 66
7.23 Checking the Investigator’s loan ..................................................................... 67
Abstract

Many commentators, such as Anthony Hall (Hall, 1990), conclude that formal design is the ideal method to create a coding project. In order to test this claim, an attempt has been made to use formal design to create a computer version of the popular board game *Arkham Horror*. This version will allow multiple users to play the game over a network. A chat system was also created to improve usability by allowing players to communicate with each other whilst playing. Every component in the game was successfully modelled, although a working version of the game was not fully realised.
Chapter 1

Introduction

1.1 Overview

Software engineering is a discipline that was devised to ensure that programs would have some form of quality. Software engineering encompasses the maintainability of the code, good coding practices, and team management. The term first appeared in the 1968 NATO Software Engineering Conference, in response to the increase in power of machines at the time and the subsequent delay to adapt to and code understandably for these machines (the so-called “software crisis”) (Dijkstra, 1972). It was created in order to prevent problems such as budget overruns, poor security, and, in extreme cases, death.

Methods to deal with the software crisis included tools such as structured programming, object-oriented programming, and documentation, programmer discipline, defined processes and methodologies, ethics, and, pertinently for this report, formal methods such as those used in engineering. These included the Z notation.

The Z notation was first conceived of in its current form by Jean-Raymond Abrial in 1977, with the assistance of Steve Schuman and Bertrand Meyer. It is named after Zermelo-Frankel Set Theory. Z is based on the standard notation for axiomatic set theory, lambda calculus, and first-order predicate logic. The de facto reference guide to Z is written by J.M. Spivey (Spivey, 2001), and was consulted throughout this project, alongside Using Z by Woodcock and Davies (Woodcock & Davies, 1996).

1.2 Objectives of Project

In this project, it is proposed to examine the efficiency of formal methods in designing a piece of software, using Z notation as the specification language. The board game *Arkham Horror* was chosen as a case study primarily due to personal knowledge of the game, thus

---

eliminating the need for a domain specialist, and also because its complexity offered a significant challenge. In order to better facilitate this, it was decided to create a real-time chat system, also written in Java.

The efficiency of formal methods will be demonstrated by examining the designs of two of the components of the game, before showing their implementations in Java. The intention of the work performed was to explore the initial design and implementation of a networked version of the game through formal modelling.

1.3 Report Outline

Chapter 2 comprises an outline of software engineering: how it started and why, and what it entails.
Chapter 3 consists of a discussion of one small constituent part of software engineering: formal methods.
Chapter 4 is a description of the Z notation specification language, including sample schemas.
Chapter 5 comprises a description of the board-game Arkham Horror, consisting of the basics of play and most of the components.
Chapters 6 and 7 focus on two parts of the game not discussed in the previous chapter: the Monster and the Investigator. These chapters also include the schemas for the components, as well as the Java implementations.
Chapter 8 is an outline of the real-time chat system, showing its current form, which incorporates unique client IDs, logging, and configuration files.
Chapter 9 concludes by considering the successes and challenges of the project and considers the effect of formal design on the project.
Chapter 2

A Brief Introduction to Software Engineering

In this chapter, the theory of software engineering will be discussed, including processes performed and an example of a software engineering model.

2.1 History of Software Engineering

The software crisis mentioned previously was caused by the release of hardware incorporating integrated circuits (Sommerville, 2007). Due to this hardware revolution, software could become far more powerful, but correspondingly grew larger and more complex, resulting in delays of months or years, as well as increasing the number of bugs in the software. As a consequence, whilst hardware costs declined, software costs increased considerably. Software engineering was proposed as a way to streamline development, allowing more complex systems to be developed cheaper and more bug-free.

In addition, further solutions, including object-oriented programming, documentation, and formal methodologies, were created and/or promoted as effective to combat the issues caused by the software crisis.

2.2 Software Engineering Processes

Sommerville (Sommerville, 2007) writes that there are four main processes in software engineering: specification, development, validation, and evolution. The following is an outline of the processes he identified.

**Software specification** is where the client and the engineers define the software and its constraints. Specification consists of carrying out a feasibility study to see whether the software will just be replicating features in pre-existing software, determining the requirements through examination of pre-existing solutions and discussing with
potential users, formalising these requirements into a specification document, and then running the final specifications by the client. If any errors are made, the original requirements must be corrected, followed by the specification document.

Software development is the actual software design and implementation. It consists of identifying what sub-systems are contained in the software system, making an abstract specification of each sub-system, designing the interfaces to allow each sub-system to interact with other sub-systems, designing the component parts of each sub-system, designing the data types in use by the system, and then designing the algorithm. Formal design may be used when designing the interfaces.

Software validation is testing the software to ensure it meets the specifications. This involves testing each individual component to ensure it meets specifications, testing the system as a whole using simulated date to ensure it meets specifications, and finally testing the system as a whole using actual data from the client to ensure it does exactly what the client wants.

Software evolution occurs after the software is released onto the market, and consists of adding features and/or removing bugs that were missed earlier, in order to keep the product current and fully usable. This method usually comprises, firstly, reviewing or changing the system requirements. The existing system is then compared to the requirements. If changes can be made, these are proposed. The system is then modified to incorporate these changes.

2.3 Software Engineering Models

Sommerville (2007) proposed that there are three main models: waterfall, iterative development, and component-based software engineering. By way of example, it is proposed to consider two models in more detail: the waterfall model, as it was one of the more widely-used models, and the iterative development model, as it is the basis for most modern models such as agile and Extreme Programming.

2.3.1 Waterfall Model

The waterfall model consists of taking the aforementioned processes, splitting them into their component phases (specification, design, implementation, testing, etc.), and only proceeding to the next phase once the current phase is completed, with a later option to return to an earlier phase if necessary. Figure 2.1 is an image of the waterfall model1.

The first step shown involves consulting with the client about the specifics of the software, namely what the software needs to do, what it shouldn’t do, and how it should work.

This is then formalised and becomes the specification document for the system.

Next, the system’s architecture is defined, and the software is defined in terms of abstractions and their relationships with each other.

After this, the design is written as code, and each part of the system is tested carefully to ensure it meets the specifications set down for it.

Next, the system as a whole is brought together and tested. Once testing is complete, the software is considered complete, and is delivered to the client.

Finally, the software is installed and put into use. If bugs are found that were not uncovered earlier, these are repaired. If additional features are required, these are implemented. Normally, this phase has the longest lifecycle.

The waterfall model is generally criticised by theorists as it does not properly represent how software development "is a complex, continuous, iterative, and repetative process" and was designed so that projects would meet deadlines (Larman & Basili, 2003). An example of a model that does deal with these issues is iterative development.

### 2.3.2 Iterative Development

Iterative development is a cyclic development cycle. It involves designing a system through multiple iterations, making incremental changes during each cycle. A cycle consists of the design, implementation, and verification steps in the waterfall model above. Unlike waterfall, however, each part is fully tested and confirmed to meet the specifications before a new component is created, whereas waterfall creates all of the components, then tests
The first step is to identify the basic specifications for the project, including what software and features are required.

Within the loop, software is designed, implemented, tested, and then reviewed. If it falls below standards, or is considered to not meet the specifications, the cycle begins again, with components being modified to better fit the specifications given.

Finally, when all of the programming and testing is complete, the loop is exited and the code is finally delivered.

### 2.4 Summary

As can be seen, software engineering allows for a structured approach to programming. In the next chapter, an alternative method of specification and design will be looked at: formal methods.

---

Source image created by user Westerhoff and released to the public domain. [http://upload.wikimedia.org/wikipedia/commons/a/ac/Iterative_development_model_V2.jpg](http://upload.wikimedia.org/wikipedia/commons/a/ac/Iterative_development_model_V2.jpg)
Chapter 3

Basic Formal Methods

In this chapter, the idea of formal methods will be discussed. A brief history is included, as well as a look at seven common myths of formal methods, finishing with a description of the two types of formal specification languages.

3.1 History of Formal Methods

Formal methods is a term used to describe any techniques that use mathematics to represent software design, specification, or verification (Sommerville, 2007). A specification created using formal methods uses a formally defined language.

In the 1980s, formal methods were proposed as an ideal way to produce high-quality software. Further, it was predicted that, by the 21st century, a large amount of software would be developed using formal methods. However, as Sommerville notes, this is evidently not the case, and he identified the following four possible reasons for this.

• Because software engineering itself has improved at a considerable pace, meaning that quality software could be created without the use of formal methods.

• Because of a shift in the market: before, quality of code was the most desirable element of a piece of software; but today, the main concern is how rapidly it can be developed. As formal methods are not particularly compatible with rapid software development methods, formal methods aren’t normally used in a more commercial environment.

• Possibly due to the era in which they were originally conceived, formal methods are rather limited in scope, being unable to fully grasp user interfaces or interaction, which have become important.

• Formal methods is not suited for any kind of large-scale project, due to the exponential increase in the complexity of the design.
CHAPTER 3. BASIC FORMAL METHODS

While this has lead to few people embracing formal methods as a way of designing software, it is still considered an excellent way to verify the correctness of a system.

3.2 “Myths” of Formal Methods

Anthony Hall laid out seven popular “myths” of formal methods in his paper on the subject (Hall, 1990), which it is appropriate to now consider.

Firstly, that formal methods guarantee perfect software. Upon reflection, this might be considered open to challenge: formal methods requires verification, which is normally performed by humans, who are prone to making errors. Thus, formally-designed software is not completely bug-free, although it is fairly robust. With larger projects, the amount of proofs to check scales exponentially. At which point, users may use a program that formally verifies designs for them. Unfortunately, this leads to another problem: the verifier itself needs to be verified bug-free. Which means either using another, unverified verifier, or fallable human checking. As a result, whether using a code verifier or doing it by hand, it is impossible to 100% guarantee that the software is bug-free. Hall’s argument is much the same, commenting that nothing can possibly be completely infallible (Hall, 1990, p. 12).

Next, that formal methods only exist to produce mathematical verification. In this project, all of the design was carried out using formal methods, but it was not considered necessary to to verify the designs. Hall comments that formal methods consists of four activities: “writing a formal specification, proving properties about the specification, constructing a program by mathematically manipulating the specification, and verifying a program by mathematical arguments.” Thus, he says, verification is not the be-all and end-all of formal methods, as they can be used for one or more of the activities listed.

Thirdly, that formal methods are only useful for safety-critical systems. Sommerville (2007, p.221) states that developing with formal methods results in equivalent specification and implementation costs and reduced validation costs, compared to standard development, where validation is approximately 50% of the development costs and design and implementation are each twice the cost of specification. Thus, formal methods are also useful to those companies wishing to save on costs. Hall’s argument proposes that formal methods should be used for any piece of software where the “cost of failure” is significant, and that any serious piece of software has one or more of the four properties that he mentions for identifying software with a high cost of failure (Hall, 1990, p. 16).

Fourth, that formal methods requires highly-trained mathematicians. This project provides an example of a system not requiring complex mathematics. The most complex comprised constructing a graph using existing nodes in the design of a schema, where some set theory was employed, though it proved to be relatively simple. Hall states exactly the
same, adding that “anyone who can learn a programming language can learn a specification notation like Z” (Hall, 1990, p. 16).

Fifth, the myth that formal methods increase the price of development. As previously noted (see third point above), formal methods actually results in a decrease in software costs, which normally means that using it will actually result in an increase in profit. Hall agrees, and uses a specific case, namely a report from Rolls-Royce and Associates\textsuperscript{1}, which states that when planned testing and formal specification were used in a safety-critical system, productivity was improved over cases where neither were used (Hall, 1990, p. 17).

The sixth myth is that formal methods are not accepted by users. When designing in Z, the result of modelling the specification was found to be clear and concise, and required very little understanding of mathematics or logic to understand; everything that is potentially unclear is easily explainable. Hall agrees, but comments that the specifications may make no sense to the user (Hall, 1990, p. 18). This difficulty did not seem to be present, however, though this may perhaps be ascribed to increasing familiarity with Z.

Finally, the myth that formal methods aren’t used by large corporations. Whilst previous comments about the scalability of formal design still apply, if small segments of the project are designed and verified separately, errors will be kept to a minimum. Hall concurs that formal methods are used by large corporations (p. 19), giving specific examples of companies using formal methods, although he does not go into detail on their methodology.

These myths will be reconsidered in the light of the experiences gained from this project in the final chapter of this project.

### 3.3 Specification Languages of Formal Methods

Two main approaches to formal methods exist, according to Sommerville (2007, p.222): the algebraic approach, where the system is described in terms of operations and relations, and the model approach, where a model of the entire system is built in terms of states, using mathematical constructs, and operations are defined as state-changing operations.

Each approach has a number of different specification languages. These are themselves divided between sequential languages and concurrent languages.

The specification language used in this project is Z notation, which is a model-based language. As such, the algebraic language will be only briefly mentioned here.

3.3.1 Algebraic Approach

The algebraic approach was originally designed to define abstract data types, where the type is represented by operations rather than by representation. This makes it like an object class in object-oriented programming. Figure 3.1 is a template algebraic structure (Sommerville, 2007, p. 224). From top to bottom, these four parts are:

- **An introduction** This contains the line sort, followed by the name of the sort being defined. A sort can be thought of as a class type in an object-oriented language. The name of the sort is not necessarily the same as the specification name. The introduction may also include an imports declaration, which comes after the sort name.

- **A description** This is written in a natural language and describes informally the sort being defined. It also includes an informal definition of each of the operations, and possibly of some more unusual values for variables.

- **The signature** This lists all of the operations on the sort, the number and types of their arguments, and their return value. These are in the form name (argType) → returnType. There are two types of operations: those that return an object of the defined sort, and those that return values of the sort.

- **The axioms** These define the behaviour of the sort by defining how each operation in the signature used to check values in the sort are related to each operation that returns an object of the defined sort.

3.3.2 Model-Based Approach

Algebraic approaches are best-used when describing interfaces where object operations are independent of object state. However, this leads to difficulties when describing system behaviour. An alternative for describing system behaviour is using a model-based language.
Model-based specification describes the system as a state model, where operations are
defined by how they change the state of the system. While most model-based specification
languages are hard to understand, the creators of Z notation made their system remarkably
easy to read.

3.4 Summary

Formal methods, in contrast to some perceptions, need not require complicated mathe-
matics and do not necessarily produce designs unintelligible to users. The following chapter
will comprise an in-depth look at a model-based language, namely Z notation.
Chapter 4

The Basics of Z

In this chapter, the layout of a Z notation schema will be shown, followed by several example schemas representing a music playlist and operations that may be performed on it.

4.1 Description of Z

Z notation is a specification language for formal methods. It is a way to formally represent models inherent in the system being designed. It uses simple mathematical notation, predicate logic, and set theory to create states and operations on those states.

Z notation takes all of these elements and combines them into an entity called a schema. A schema is a basic element of the component being designed. It is normally used to represent what would, in object-oriented programming, be termed a class.

An entity being represented in Z normally has three types of schemas. The first is the state schema, which defines the basic variables and constraints on the entity, but does not set any values. The second is the initialisation schema, which assigns values to the variables in the state schema. Finally, there are operational schemas, which carry out operations using data in the schema. There are two types of operational schema: those that do not modify the data in the schema, and those that do.

This is a dummy schema:

```
Dummy
variables
constraints
```

The name of the schema appears at the top. In this case, the schema is called Dummy. There is a dividing line in the schema. On the top are variables and schema imports (which
are fully discussed in 4.3 below). On the bottom are constraints and operations on the variables.

To illustrate more fully, an example schema will now be discussed, representing a music playlist (i.e. a list of musical tracks that are either played in sequence or in a random order), then its initialisation schema, followed by three operational schemas.

### 4.2 Basic Z State Schema: a Playlist

```
Playlist
name : STRING
tracks : P TRACK
shuffle : BOOLEAN
#tracks ≠ 0
```

Note: the data type TRACK is used to represent music tracks, but the schema for this data type will not be defined here. It is sufficient to note that it has a variable called name that is a String. As this shows, it is possible to use other schemas as data types.

The Playlist object has three variables: the name of the playlist, which is a String of characters, a set of TRACKs called tracks (P signifies a set), and a boolean variable called shuffle that tells the player whether the playlist is in shuffle mode or not.

This schema shows one example constraint: the cardinality of tracks, ie the number of elements in the set tracks, cannot be zero. In other words, there MUST be tracks in a playlist. If there aren’t, an error will occur.

### 4.3 The Initialisation Schema For a Playlist

```
PlaylistInit
Playlist
name? : STRING
tracks? : P TRACK
shuffle? : BOOLEAN
name' = name?
tracks' = tracks?
shuffle' = shuffle?
```

This schema has a different name from the previous one. In order to refer to it, the
name of the original state schema appears on the first line. This is called \textit{schema conjunction}. Using this method, all of the variables contained in the named schema are imported into this new schema, without having to define any new variables. Doing this is necessary if it is required to access or change any variables in a schema. It is possible to perform schema conjunction in one schema using more than one schema.

The three variables in this schema have question marks after their names. This indicates that they are input variables. In the lower half of the schema, they are assigned to primed variables. An initialisation schema is a special type of operational schema that changes the state schema. When the state schema is changed, there are now two sets of variables: the original variables, without primes, and the variables in the new state, which are primed. Since the original variables did not have any old values, this schema assigns values to them.

For all other operational schemas, it is assumed that, for all variables not mentioned, the new value is the same as the old value, i.e. \( \text{variable'} = \text{variable} \).

### 4.4 Operational Schemas For the Playlist

This first operational schema does not change the state of Playlist.

\[
\text{GetTracks} \\
\Xi \text{Playlist} \\
\text{tracks}! : \forall \text{TRACK} \\
\text{tracks}! = \text{tracks}
\]

There is a schema conjunction with Playlist, as evidenced in the first line. However, the \( \Xi \) shows that the state of the Playlist doesn’t change, i.e. that the values of the variables do not change.

The variable with the exclamation mark is an output variable. Once the operation is complete, it should contain data, which is returned to the caller. In this case, the output variable is returned with the contents of the tracks variable. It is possible for an operational schema to have more than one output, but this increases the difficulty in implementing it in code.

A brief note: in Z, the variables \texttt{varname}, \texttt{varname'}, \texttt{varname?}, and \texttt{varname!} are all distinct from each other and are all treated differently. An unprimed variable name is the initial state of the variable, while a primed variable is the final state, and, as mentioned previously, a variable followed by a question mark or an exclamation mark are inputs and outputs respectively.
This operational schema, on the other hand, does change the state of the Playlist.

\[
\text{AddTrack} \\
\Delta \text{Playlist} \\
\text{track}? : \text{TRACK} \\
\text{tracks}' = \text{tracks} \cup \{ \text{track}? \}
\]

As in the previous operational schema, there is some schema conjunction here. The $\Delta$ signifies that, unlike with the previous operational schema, the final state of the Playlist will be changed in some way.

Here, the input track variable is added to the track list. As tracks is a set, it is simple to add a track to it by taking the input TRACK, unifying it with the set of tracks, and then setting the new value of the tracks variable is this new set consisting of the old tracks and one new one.

This chapter concludes with a more complex operational schema with both input and output variables, as well as showing set theory in action.

\[
\text{GetNamedTrack} \\
\Xi \text{Playlist} \\
\text{name}? : \text{STRING} \\
\text{tracks}! : \mathcal{P} \text{TRACK} \\
\text{tracks}! = \{ x | x \in \text{tracks} \land x.\text{name} = \text{name}? \}
\]

The input variable is the name of a track, while the output variable is a set of tracks that contain all tracks with that name. In the constraints section, tracks! is created as a set consisting of elements called x, where x is an element of the Playlist’s tracks variable and x’s name is the same as the input string.

### 4.5 Summary

As discussed, Z notation produces a clear representation of a system’s components and operations on them. The next chapter will consider the game that it was decided to model.
Chapter 5

An Introduction to Arkham Horror

This chapter describes the game, its rules, and objectives of play. Almost all components of the game are described here, and all have been fully designed in Z notation. The full schemas are included in Appendix A.

5.1 Basic Rules and Goals

The game *Arkham Horror* (2007), designed by Richard Launius and published by Fantasy Flight Games, is based on the Cthulhu Mythos, a story setting created by H.P. Lovecraft and used by himself, August Dereleth, Frank Belknap Long, and others.

The theme of the Mythos is that horrible creatures from other worlds and dimensions are innately inimical to human life, as they are completely beyond our comprehension, and that their arrival spells the end of the world, if not the universe. Thus, the goal of the game is to either prevent these creatures from arriving, or, on their arrival, to defeat them, sending them back from where they came.

Each game comprises a series of rounds, with each round itself split into phases, the phases themselves comprising a series of turns. In each round, a new player is chosen as the first player. This player takes their turn first in each phase, with the player to their right going next, and so on. Additionally, in the event of a tie or a choice, such as a monster that attacks the nearest Investigator being equidistant from two, the first player chooses what happens. The phases occur in the following order:

**Upkeep phase**  Each player takes turns performing some basic upkeep on their character.

**Movement phase**  Each player takes turns moving their Investigator around the Board.

**Arkham Encounters phase**  Players still in Arkham take turns to have encounters, depending on where they are.

**Other World Encounters phase**  Players in other worlds take turns to have encounters.
Mythos phase The first player takes the top card of the Mythos deck, reveals it, puts into play any clues, Gates, or monsters mentioned on the card, moves monsters as directed by the card, and then performs any special actions laid out on the card. The round then ends, and the first player is now the player to the right of the previous first player.

Each player controls an Investigator, which has two statistics (Sanity and Stamina) and six skills (Speed, Sneak, Fight, Will, Lore, and Luck). When one of the statistics drops to 0, the Investigator is moved to another Location and loses a number of Items. The Skills are used as part of skill tests: the player must roll a number of standard six-sided dice equal to the Skill’s level, taking into account any modification of the level. For the majority of tests, one or more dice reading 5 or 6 is considered a success. For moving around the board, an Investigator may only move to those Locations or Streets that are at most a number of spaces away equal to his Speed.

The game is won when one of the following occurs: either each Investigator has closed at least one Gate and there are no more open Gates on the Board, or six Gates have been sealed, or the Great Old One has been beaten in combat. The game is lost when the Great Old One defeats the Investigators in combat, or, in one case, simply awakens.

5.2 The Board

Figure 5.1 is an image of the Board. The largest part of the Board is the town of Arkham, which consists of a number of Neighbourhoods, which themselves include Locations and Streets. On the left-hand side are areas called Other Worlds. At the bottom is the Terror Track and the City Limits areas.
5.2.1 Neighbourhoods

Figure 5.2 is a detail of the Neighborhood of the Merchant District. Each round area is called a Location. Each rectangular area is a Street. This area is called the Merchant District streets, named after the Street that all the Locations go to. Each path has two arrows.
on it, a Black and a White. The Mythos cards all have two strips on them, one black and one white. The Symbols that appear on these match Symbols that appear on Monsters. Thus, the Mythos cards control which Monsters move that round, and the direction they move in. Note, however, that Investigators may ignore these arrows and travel in either direction on a path. Further, any path that does not have any arrows cannot be used by Monsters. Thus, Monsters will travel clockwise around the Board when following the White arrows, and anticlockwise when following the Black.

Locations are where Investigators have Encounters, find Clues, buy Items or Spells, or leap into Gates in order to hold back the monstrous enemies flooding the town. There are multiple Arkham Encounter decks, each one with a different colour on its back. During an encounter in a Location in the Merchant District streets, the player would draw the top card and look at the entry for their current Location. They then fulfill the requirements on the card to the best of their abilities. If they succeed, either their Investigator receives something in return, or nothing happens. If they fail, their Investigator normally loses something vital, such as Stamina, Sanity, Items, or may even be removed from the game, requiring the player to use a new Investigator.

Streets, on the other hand, do not have any encounters. All that may occur in Streets is combat with Monsters or certain events called Activity.
5.2.2 Other Worlds

Figure 5.3: Detail of the Dreamlands

Figure 5.3 is a detail of an Other World, in this case The Dreamlands. When an Investigator arrives in this Other World, they have an Other World encounter during the appropriate phase. The player takes the top card of the Other World Encounters deck. If it matches one of the colours shown in the top-right, they look for their current Other World. If it’s listed, they follow the instructions there. If it isn’t, they follow the instructions in the Other section. If the top card doesn’t match the colours of the current Other World, the player sets the card aside, and takes the next one, repeating the process. The discarded or used Encounter cards are then put on the bottom of the deck.

Unlike on the normal Board, Investigators must move one space in the Other World during the Movement phase, namely from the left of the Other World region to the right. They then have another Other World encounter. Finally, on the third round in the Other World, they leave the Other World, returning to Arkham via any Location with a Gate to that Other World.
5.2.3 The City Limits

Figure 5.4 is a detail of the City Limits section of the Board. On the left is the Outskirts. During the game, there is a limit to the number of Monsters allowed on the Board, which is equal to the number of players plus 3. If there are ever that many Monsters on the Board, any additional Monsters are instead put into the Outskirts. The Outskirts has its own limit, which is equal to 8 minus the number of players. When it is filled, all the Monsters in the Outskirts are returned to the pool of Monsters that can be put on the Board, and the Terror Track (more later) increases by one.

In the middle is the Sky. Any Monsters that can fly move here if they can’t attack a nearby Investigator. The Sky has no limit, but Monsters in it count towards the Monster Limit. During the Movement phase, any Monsters that are allowed to move can move into a Street where there is an Investigator. If there are no Investigators in the streets, the Monster stays where it is.

On the right is the Lost in Time and Space area. As a result of some encounters, being reduced to 0 Sanity or Stamina while in an Other World, or being in an Other World when the last Gate to it is closed, Investigators will be moved to this space. After two rounds, the Investigator may finally return to Arkham proper, moving to any Location or Street on the Board.

5.2.4 The Terror Track

Finally, figure 5.5 is a detail of the Terror Track. This has a counter on it. At the start of the game, the counter is placed on the 0. Certain events and Mythos cards cause it to increase by 1. Whenever it increases, an Ally is taken from the top of the Ally deck and removed from the game. When the Track reaches 3, the General Store in Rivertown closes. When it reaches 6, the Curiositie Shoppe in Northside closes. When it reaches 9, Ye Olde Magick Shoppe closes. At 10, the Monster Limit is removed, and the Great Old One is closer to waking up. The Terror Track is particularly dangerous for players, as there is no
Figure 5.5: Detail of the Terror Track

way to reduce the Terror Track.
5.3 The Great Old One

The *Great Old One*, unlike Monsters, is the primary enemy in the game (the primary antagonist, however, is the game itself). When the Great Old One awakens, the Investigators must fight him, sending him back to sleep. The Great Old One will awaken either when its *Doom Track* is filled, or when there are too many Gates open, or if there are no more Gate markers left, or there are no more Monsters in the *Monster Cup*, or the Terror Level reaches ten and there are too many Monsters on the Board (twice the normal Monster Limit or more).

![Figure 5.6: Great Old One sheet for Ithaqua](image)

Like Monsters, the Great Old One has certain abilities, which are shown in section 5.8. Figure 5.6 is the Great Old One sheet for Ithaqua.
CHAPTER 5. AN INTRODUCTION TO ARKHAM HORROR

Figure 5.7: Detail of Ithaqua’s worshippers and combat rating

On the left is the information on Ithaqua’s worshippers and his Combat Rating (see Figure 5.7). The number on top is the penalty to Combat checks that each Investigator gets when fighting Ithaqua after he awakens. Here, it’s -3. Below that is the benefits his followers, normally Cultists, receive for the entirety of the game. Normally, this is some kind of bonus in Combat. Ithaqua’s Cultists have their Toughness increased by two. This doesn’t affect the Monster Trophy, however.

Figure 5.8: Detail of Ithaqua’s defences (none) and his attack

Next, the right hand side, which consists of Ithaqua’s Defences and what its attacks do (Figure 5.8). The top line normally contains one or two Defences of the kind that Monsters get. In Ithaqua’s case, however, there are none. Most Great Old Ones have an attack and a start of battle ability. In Ithaqua’s case, Investigators must roll a die for each item they have, discarding the item if they roll a failure (normally below 5, unless the Investigator is Blessed/Cursed). Ithaqua’s actual attack requires each Investigator to pass a Fight +1 check, or lose two Stamina. The second round, each Investigator needs to pass a Fight +0 check, the following round, a Fight -1 check, and so on.
Each Great Old One also has a power which is active for the entire game (Figure 5.9. Ithaqua’s power causes any Investigators that stop their Movement in a Street to lose one Stamina, and for all Environment (Weather) Mythos cards to be discarded without their special effects taking place.

Finally, Ithaqua’s Doom Track, shown in Figure 5.10. When a Gate opens, or the Terror Track is filled, or in certain other circumstances, a Doom Token is added to the Doom Track. When it is filled, or one of the other conditions mentioned earlier are fulfilled, the Great Old One awakens, and the final battle begins. If the Great Old One is not awoken by filling his Doom Track, it is filled first. Every time an Investigator succeeds on a Fight check against a Great Old One, or, before it awakens, a Gate is sealed using an Elder Sign, a Doom Token is removed from the Great Old One’s Doom Track. If it is reduced to zero while the Great Old One is awake, it returns to its deep slumber, and the Investigators win the game.
5.4 Other Items and Characters

The play of the game is mostly dictated by various shuffled decks of cards. Each of the nine Neighbourhoods has one deck which dictates the outcome of encounters in that Neighbourhood. There is also a deck for Other World encounters and another for the Mythos cards, which are revealed during the Mythos phase. There are also decks for Common Items, Unique Items, Skills, Spells, and Allies, as well as cards representing Bank Loans, Retainers, membership of the Silver Twilight Lodge, Blessings (or Curses), and being the Sheriff’s Deputy.

Figure 5.11: Encounter card for the Merchant District

Figure 5.11 shows an Arkham Encounter card for the Merchant District. If the Investigator is in the River Docks, they perform the actions listed in that section of the card, and so on. There are multiple cards for each Neighbourhood, each with different encounters.
Figure 5.12: Yellow Other World Encounter card

Figure 5.12, on the other hand, shows an *Other World Encounter*. This card is yellow. It is thus valid for The Dreamlands. However, as the card does not list The Dreamlands, the Encounter will be the one that falls under Other on the card. There is only one deck of Other World Encounter cards, which can each be one of four colours. If the colour of the card doesn’t match a colour on the Other World, a new one is drawn.

Figure 5.13: The Mythos card “Picnickers Panic!”

Figure 5.13 is a *Mythos card*. At the end of each round is the Mythos phase, where one of these is drawn. At the top is the name and the type, which can be a Headline (as here), a Rumour, an Environment (Weather), an Environment (Mystic), or an Environment (Urban). Below this is the text explaining what happens now. Below that is the Location that a Clue appears, unless there’s a Gate open there. At the bottom, the Location of the next Gate is shown, as is the Monster movement. Here, a Gate opens at the Unvisited Isle, and crescent Monsters travel in the White direction, while cross Monsters travel Black. When a Mythos card is revealed, the Gate is opened. If there is no Gate there already and the
Location hasn’t been sealed, then the Gate is opened and one Monster appears there. If the Location is sealed, no Gate appears. If there is a Gate there already, Monsters emerge from every Gate on the Board. Next, the Clue is placed. Then, Monsters move. Here, if any Monsters have a crescent or cross, they move, while other Monsters do not. Finally, the text on the card (the Mythos ability) is activated. Here, more monsters appear in the Downtown Streets.

Next, figure 5.14 is a Common Item. This item is a weapon, the .45 Automatic. It is of the type Physical Weapon. Other possible types are Magical Weapon, Tome, or no type at all. Many Items have some kind of rules text or ability, which give bonuses to Skills or other advantages. This one gives a bonus to Combat checks, which means it gives a bonus to the Fight skill, but only when fighting a Monster. The bottom left represents how many hands are required to use the Item. An Investigator can only use two hands worth of Items at once. Thus, if he is using this Item, the only other Items he can make use of during his turn are those with one hand requirement or no hand requirements. Finally, the number in the bottom-right is, of course, its cost.
Figure 5.15: A Unique Item: the Elder Sign

Figure 5.15 shows the Elder Sign, a *Unique Item*. Unique Items have the same basic design as Common Items, but are normally much more useful. They have hand requirements and costs as well. In this case, the Item can be used without sacrificing a free hand, and costs $5. The rules text states that this is one of the two ways to permanently seal a Gate, the other being to discard six Clue tokens. This Item, however, also removes a Doom Counter from the Old One’s Doom Track, increasing the amount of turns before it awakens.

Figure 5.16: A Skill card: +1 Will

Figure 5.16 is a *Skill*. Skills normally give some kind of passive, always-on benefit, normally an increase to an existing Skill (i.e. part of a Skillpair). This one increases the Investigator’s Will by 1, and allows them to roll additional dice when they spend a Clue token for a Will check.
Figure 5.17 is a Spell. Like Items, Spells normally have a hand requirement. This spell requires both hands free to cast it. Unlike with Items, Spells require successful casting, and a cost. The Investigator must pay the cost, then make a Lore check with the casting modifier. If they pass, they have successfully cast the spell, and gain the benefits listed at the bottom. If they fail, the spell didn’t work. If the Spell is not exhausted, they may attempt to recast it. However, whether the Spell succeeds or fails, the Investigator must still pay the Sanity cost.

Figure 5.18 is an Ally called Professor Armitage. Unlike Skills, Items, or Spells, there are no duplicates of Allies. Allies all boost one or more Skills, and can have passive abilities, abilities that require exhausting, or abilities that require discarding them. In this case, the Ally increases the Investigator’s Lore skill by two, and nullifies the Monster ability Magical Resistance, meaning attacks with magical weapons do not have their damage halved. When the Terror Level increases by one, the top Ally is taken from the deck and removed from the game.
5.5 Summary

The Game itself contains a large amount of components, each of which required modelling. The next two chapters will deal with two components which have not been described in this chapter: the Monster and the Investigator.
Chapter 6

Example 1: Monster Design and Code

This chapter will describe the Monster, examine its formal design schemas, and then show the schemas translated into Java.

6.1 Description of the Monster In-Game

In the game of Arkham Horror, Monsters are a constant threat to the Investigators, acting as stumbling-blocks, preventing them from completing their mission and giving the Great Old One time to awaken. Thus, Investigators are forced to constantly fight off Monsters as they try to close Gates and hunt down clues and weapons.

In the simulation, it was decided that there would only be two types of actors, as in the board game itself. The first, obviously, is the Investigator, of which there can be between one and eight. The second, slightly more surprisingly, is the game itself, which moves Monsters, opens Gates, distributes clues, and actively plays against the Investigators.

When a Gate opens in the game, one or more Monsters are chosen at random from an area called the Monster Cup. When a Monster is removed from the game, it is normally returned to the Cup.
Each Monster is represented by a cardboard token, which contains its characteristics, which govern its performance during the game. Figure 6.1 is the front, or movement side, of one of the Monsters: the Dhole. The border represents the Dhole’s movement; in this case, it is black, representing that the Dhole has a normal speed, namely it can only move one space per round, and only in the direction indicated by the Mythos card. Other alternatives are red, which means a fast speed, i.e. the Monster moves two spaces rather than one; blue, meaning flying, which means the Monster will move to the Sky space when it enters the Street if there are no Investigators in that Street or an adjacent one, and can swoop down out of the Sky when allowed to any Street containing an Investigator; yellow, meaning stationary; and green, which means that the Monster has a unique way of moving.

The symbol in the bottom right is the Dhole’s Dimension symbol. If there is a Gate on the Board with that symbol, closing that Gate will send all Monsters with that symbol, including the Dhole, back to the Monster Cup, providing there are no other Gates with that symbol open.

The top-right number is the Dhole’s awareness. When attempting to Sneak past the Dhole, the Investigator’s Sneak skill is reduced by the awareness value. Most Monsters have a negative modifier, while others will have no modifier, and some rare few having a positive modifier.
Next to be considered is the other side of the token, which is called the combat section, as it has all of the information required to fight the Monster listed (see Figure 6.2). The lines of bold text are the Monster’s abilities. In this case, the Dhole takes half damage from physical weapons, half damage from magical weapons, causes any Investigator to encounter it to automatically lose at least one Sanity, and causes any Investigator to defeat it to automatically lose one Stamina. Other abilities are Magical Immunity and Physical Immunity, which do exactly as they say, and Ambush, which means that, once combat has started, running away is impossible. There are also three unique abilities, which can more accurately be described as types: Endless, which means the Monster cannot be claimed as a trophy after being killed, and instead returns to the Monster Cup; Undead, which means certain weapons work better against them, while others don’t work at all; and Mask, which is only possessed by five Monsters, which can only be in play when the Great Old One is Nyarlathotep.

Below this section, there is usually some flavour text, taken from the story in the Cthulhu Mythos the Monster originated from, but in this case there are too many abilities. There may also be special rules for the Monster, such as in the case of the Dimensional Shambler, which, if it wins one round of a combat, will cause the Investigator to become Lost in Time and Space.

Below this is the encounter stats. On the left is the Horror section. The number on top indicates the penalty to the Investigator’s Horror check. The blue tokens indicate the amount of Sanity the Investigator will lose if they fail the check. If they succeed with the Dhole, however, they still lose one Sanity, due to Nightmarish 1.

On the right of the token is the Combat section. As with the Horror section, the top number indicates the penalty to the Investigator’s Fight roll, while the bottom section indicates the amount of Stamina the Investigator will lose at the end of the Combat round. Due to Overwhelming 1, the Investigator will lose one Stamina even if they defeat it.

The blood drops in the center represent the Monster’s toughness. In a fight, the In-
vestigator must roll that many successes in order to defeat the Monster. On doing so, unless the Monster has the Endless ability or has special rules relating to it being defeated, he claims the token as a *Monster Trophy*. Monster Trophies are categorised by the Monster’s toughness. Thus, when spending trophies, the Location may ask for five toughness worth of Monster Trophies. When the Investigator spends Monster Trophies, the tokens are returned to the Monster Cup.

### 6.2 Formal Design of the Monster

The following were the steps followed to design the Monster formally.

- First of all, all elements of the Monster were defined. These elements were the Monster’s name, its awareness, its rules, its dimensional symbol, its movement type, its abilities, its Combat section, its Horror section, its Toughness, and its static types (Undead, Mask, and Endless).

- As both the Combat and Horror sections had more than one variable in them (Rating and Damage), it was decided that it would be better to create separate entities for each.

- As a Monster may have only one movement type, and there are a limit number of movement types, and only one variable would be used, it was decided to make movement type an enumeration. The same logic was applied to the dimensional symbol.

- As above, it was decided that abilities should be an enumeration. However, since Monsters may have more than one ability, abilities in a Monster were represented by a set of abilities. However, the elements in this set must be unique. This is called an *injective sequence*.

- The static types were considered to be the same as flags, i.e. a Monster may be undead or not. Thus, these three elements were represented as boolean variables. Everything else was representable by standard data types: name and rules were represented by Strings, awareness and toughness by integers.

- The next step was to identify upper and lower bounds on awareness, toughness, and the two variables in Combat and Horror. In order to choose these, every single Monster was inspected, with the lowest and highest values amongst every Monster being chosen as the lower and upper bounds respectively of each variable, whether the bound only appeared on one Monster or multiple.

With all of these actions performed, the following schemas were designed.
6.2.1 State Schemas

This free variable represents the different types of movement a Monster may possess.

\[ MOVE ::= Fast \mid Flying \mid Normal \mid Stationary \mid Unique \]

While this one represents the Monster’s Dimension.

\[ SYM ::= Circle \mid Crescent \mid Diamond \mid Hexagon \mid Plus \mid Slash \mid Square \mid Star \mid Triangle \]

And this one represents the Monster’s abilities.

\[ DEFENCES ::= PResistance \mid PImmunity \mid MResistance \mid MImmunity \mid Nightmarish\langle1\rangle \mid Overwhelming\langle1\rangle \mid Ambush \]

This schema represents the Horror section of a Monster. The rating is the penalty to the Investigator’s Will check, while the damage is how much Sanity the Investigator will lose if they fail.

```
Horror
rating : Z
damage : Z
rating ∈ \{-3..1\}
damage ∈ \{0..4\}
```

This schema represents the Combat section. As with the Horror schema, the rating is the penalty, while the damage is how much Stamina will be removed if the Investigator fails to kill the Monster in one round.

```
Combat
rating : Z
damage : Z
rating ∈ \{-4..1\}
damage ∈ \{1..4\}
```

This is the Monster proper. It contains all of the variables mentioned previously in the design. The injective sequence representing the abilities has a constraint preventing both Physical Resistance and Physical Immunity from appearing, and a similar arrangement with Magical Resistance and Magical Immunity. This is because Immunity trumps Resistance, and no Monster has both the Immunity and the Resistance.
CHAPTER 6. EXAMPLE 1: MONSTER DESIGN AND CODE

6.2.2 Initialisation Schemas

The initialisation of Horror is rather straightforward; all values are passed to the variables at initialisation. There are no derived variables which rely on the values of other variables, and no complex assignments. This holds true for Combat and Monster, although Monster has initialisation of its Horror and Combat variables.

---

**Monster**

- `name, rules : STRING`
- `location : NODE`
- `abilities : iseq DEFENCES`
- `awareness, toughness : Z`
- `movetype : MOVE`
- `symbol : SYM`
- `mask, endless, undead : BOOL`
- `horror : HORROR`
- `combat : COMBAT`

```
awareness ∈ {−3..1}
toughness ∈ {1..4}
if #defences ≥ 2
  then {PResistance, PImmunity} ∩ defences ≤ 1
      ∧ {MResistance, MImmunity} ∩ defences ≤ 1
  else defences = defences
```

---

**6.2.2 Initialisation Schemas**

The initialisation of Horror is rather straightforward; all values are passed to the variables at initialisation. There are no derived variables which rely on the values of other variables, and no complex assignments. This holds true for Combat and Monster, although Monster has initialisation of its Horror and Combat variables.

**HorrorInit**

- `Horror`
- `rating? : Z`
- `damage? : Z`

```
rating' = rating?
damage' = damage?
```

---

**CombatInit**

- `Combat`
- `rating? : Z`
- `damage? : Z`

```
rating' = rating?
damage' = damage?
```
The most complicated part of the Monster initialisation schema is that, in the beginning, the Monster isn’t on the board, and so doesn’t have a location yet. This is set later, when it appears on the Board. Thus, it is the Game’s responsibility to set the Monster’s location when it is created, as the Monster is completely passive, i.e. it performs no actions of its own, being manipulated by the Game. See overleaf for the full initialisation schema for the Monster.

```plaintext
MonsterInit

Monster
name?, rules? : STRING
awareness?, horrrrat?, horrordmg?, toughness?, combatrat?,
    combatdmg? : Z
movetype? : MOVE
symbol? : SYM
mask?, endless?, undead? : BOOL

name' = name?
location' = null
awareness' = awareness?
movetype' = movetype?
symbol' = symbol?
rules' = rules?
mask' = mask?
endless' = endless?
undead' = undead?
horror' = ⟨horrrrat?, horrordmg?⟩
toughness' = toughness?
combat' = ⟨combatrat?, combatdmg?⟩
```
6.2.3 Operational Schema

There is only one operational schema that changes the state of a Monster. As previously noted, this is because the Monster is completely passive, being created, moved, and removed by the Game, rather than initiating such actions itself. This is because it does not change its own attributes. This is the schema that sets a Monster’s current location.

\[
\text{SetLocation} \quad \Delta \text{Monster} \\
\text{location?} : \text{NODE} \\
\text{location'} = \text{location}?
\]

6.3 Java Code

In order to change the schemas into code, state schemas were converted into separate classes, with variables becoming private attributes, initialisation schemas became constructors, and operational schemas became methods, with those that didn’t change the state of the schema returning void, and those that returned values returning the specified values. Enumerations in Z were converted in a reasonably straightforward manner into enumerations in Java. For the datatypes, the types mapped one to one, i.e. strings in Z became Strings in Java.

The one complication was constraints. These were converted into elaborate if statements, with each illegal value for the variable throwing an exception in Java; only if the value of the variable was legal would it be assigned to an attribute in the class.

There now follows a list of the code for the Monster. First of all, the enumeration types used for a Monster. Here is the Movement type:

```java
public enum Movement {
    Fast, Flying, Normal, Stationary, Unique
}
```

Listing 6.1: Movement enumerator

Next, the Abilities type (referring to Defences in the original design):

```java
public enum Abilities {
    PResistance, PImmunity, MResistance, MImmunity, Nightmarish, Overwhelming, Ambush
}
```

Listing 6.2: Abilities enumerator
And finally, the Symbol type:

```java
public enum Symbol {
    Circle, Crescent, Diamond, Hexagon, Plus, Slash, Square, Star, Triangle
}
```

Listing 6.3: Symbol enumerator

Next, the Horror and Combat data types. The only methods these have are access methods (methods that take no arguments and return one of the private attributes of the data type), and thus they have not been included them. First, the Horror type:

```java
public class Horror {
    private int rating, damage;

    public Horror(int r, int d) throws BoundingException {
        if (1 < r || r < -3)
            throw new BoundingException("Input Horror rating; value is " + r);
        else
            this.rating = r;

        if (4 < r || r < 0)
            throw new BoundingException("Input Horror damage; value is " + d);
        else
            this.damage = d;
    }
}
```

```java
public class Combat {
    private int rating, damage;

    public Combat(int r, int d) throws BoundingException {
        if (1 < r || r < -4)
            throw new BoundingException("Input Combat rating; value is " + r);
        else
            this.rating = r;

        if (4 < d || d < 1)
            throw new BoundingException("Input Combat damage; value is " + d);
        else
            this.damage = d;
    }
}
```

Listing 6.4: Horror and Combat data types

Note that, for Combat and Horror, a substantial portion of the constructor is used to ensure that all of the input values are within the bounds defined by the game and, subsequently, by the schema.
Now, at last, the Monster proper. First, the attributes of the class:

```java
public class Monster {
    private String name;
    private Node location;
    private int awareness;
    private Movement movement;
    private Symbol dimension;
    private Abilities[] abilities;
    private String rules;
    private boolean mask, endless, undead;
    private Horror horror;
    private int toughness;
    private Combat combat;
}
```

Listing 6.5: Monster private attributes

These private attributes are all listed in the original schema, and retain the same types. Unfortunately, due to the amount of constraints, the size of the Monster’s constructor is such that it was necessary to break it down and examine each section. First of all, the declaration for the method:

```java
public Monster (String n, int aw, Movement move, Symbol dim, Abilities[] ab, String r,
    boolean m, boolean e, boolean u, Horror h, int tough, Combat c) throws
    BoundingException, ConstraintException, NullException {
```

Listing 6.6: Monster constructor declaration

The Monster has no derived or fixed values, with the exception of location, which is set later when it arrives on the board. Thus, the values for all of the attributes (bar one) are passed to the constructor. The one difference between this and the original schema is that the Horror and Combat attributes are defined outside of the class rather than inside, in an attempt to cut down the number of parameters passed.

The next section (Listing 6.7) is that part of the constructor that prevents null assignment (ie testing certain parameters to ensure they aren’t null). Note that, while Horror and Combat cannot be null, the values inside them can be 0.
CHAPTER 6.  EXAMPLE 1: MONSTER DESIGN AND CODE

```
if (n == null)
    throw new ArgumentNullException("Input Monster name");
else if (move == null)
    throw new ArgumentNullException("Input Monster movement");
else if (dim == null)
    throw new ArgumentNullException("Input Monster dimension");
else if (h == null)
    throw new ArgumentNullException("Input Monster Horror section");
else if (c == null)
    throw new ArgumentNullException("Input Monster Combat section");
else {
    this.name = n;
    this.movement = move;
    this.dimension = dim;
    this.horror = h;
    this.combat = c;
}
```

Listing 6.7: Monster constructor part 1: preventing null assignation

The next section prevents certain values from either being too large or too small:

```
if (1 < aw || aw < -3)
    throw new BoundingException("Input Monster awareness; value is " + aw); 
else if (4 < tough || tough < 1)
    throw new BoundingException("Input Monster toughness; value is " + tough);
else {
    this.awareness = aw;
    this.toughness = tough;
}
```

Listing 6.8: Monster constructor part 2: enforcing numerical constraints

Next, the constructor checks the Abilities array to ensure that it is both the right length (ie between 0 and 4) and that it is an injective sequence (note that the second part uses a function we haven’t seen; it’ll appear later in the chapter):

```
if (ab.length > 4)
    throw new ConstraintException("Input Monster abilities array is too long; length is " + ab.length);
else if (this.checkIllegality(ab))
    throw new ConstraintException("Illegal elements in Monster abilities");
else
    this.abilities = ab;
```

Listing 6.9: Monster constructor part 4: checking Abilities array
Finally, the constructor assigns those parameters that do not need checking to their appropriate attributes. These are all boolean values, with the exception of location, which starts as null, and rules, which, if the monster has no rules, is null.

```java
this.location = null;
this.rules = r;
this.mask = m;
this.endless = e;
this.undead = u;
```

Listing 6.10: Monster constructor part 5: everything else

This is the method mentioned previously that ensures that Abilities is an injective sequence. It also checks for the constraint on the abilities set defined in the schema, which prevents a Resistance and an Immunity of the same type appearing in the set. If the set of Abilities contains an illegal element, i.e. a duplicate value or a Resistance and Immunity of the same type, it returns true, signifying that the set is illegal. Otherwise, it returns false. If the set has only one element, it cannot contain a Resistance and an Immunity, nor can it contain any duplicates, so the set is automatically legal, and the method returns false, i.e. the set is not illegal.

```java
public boolean checkIllegality(Abilities list[]) {
    if (list.length > 1) {
        for (int i = 0; i < list.length; i++)
            for (int j = 1; j < list.length; j++)
                if (list[i] == Abilities.PImmunity &&
                    list[j] == Abilities.PResistance)
                    return true;
                else if (list[i] == Abilities.PResistance &&
                         list[j] == Abilities.PImmunity)
                    return true;
                else if (list[i] == Abilities.MImmunity &&
                         list[j] == Abilities.MResistance)
                    return true;
                else if (list[i] == Abilities.MResistance &&
                         list[j] == Abilities.MImmunity)
                    return true;
                else if (list[i] == list[j])
                    return true;
    }
    return false;
}
```

Listing 6.11: Method to simulate an injective sequence and other constraints
Every other method in the class is an access method. This is an example of an access method. Note that it calls an access method from the Combat class as well.

```java
public int getCombatDamage() {
    return combat.getDamage();
}
```

Listing 6.12: An example access method

This is the only other method that isn’t an access method. It sets the Monster’s location.

```java
public void setLocation(Node loc) {
    this.location = loc;
}
```

Listing 6.13: The setLocation method

### 6.4 Summary

The Monster was one of the more straightforward components to model, being entirely passive. The only difficulties that arose related to representing the constraints in Java. The next chapter will consider a more complex component: the Investigator.
Chapter 7

Example 2: Investigator Design and Code

This chapter will describe the Investigator, show its formal specification, and then its Java implementation.

7.1 A Description of the Investigator

The Investigator is the second part of the case-study that will be examined in greater detail. The Investigator is the character controlled by the player in-game. This Investigator, along with its fellows, must try and stop the Great Old One from awakening and destroying Arkham, and subsequently the world.

The Investigators must wade through hordes of Monsters in order to seal shut Gates to Other Worlds, delaying the awakening of the Great Old One. If they fail, they have one last chance: to confront the Great Old One themselves, and send it back from where it came.
Each player is allocated a different Investigator at the start of the game. Figure 7.1 is the sheet for the Investigator Amanda Sharpe, which sets out her particular attributes, which control her performance.
Looked at in detail, the first important part is the top right of the sheet (figure 7.2). This is the Investigator’s emphname, occupation, and Statistics. Stamina represents health, while Sanity is self-explanatory. Each of these spaces have a number of counters on them for each number; so, in this case, there would be five hearts on Stamina and five brains on Sanity. Every time a point of damage is done to either of these Statistics, you take a counter off the space. The Investigator’s Stats cannot go above the maximums printed on the sheet. However, these maximums may be changed, either by a Mythos card or a Great Old One.

![Figure 7.3: Detail of Amanda Sharpe’s ability, Studious](image)

Next, the section below the Statistics, which is the Investigator’s Ability (figure 7.3). This is the ability unique to Amanda Sharpe, though other Investigators may have a similar ability. For instance, one Investigator can heal one Stamina to himself or another Investigator in their location, while another heals one Sanity.

![Figure 7.4: Detail of Amanda Sharpe’s starting possessions and home location](image)

To the left of the Ability is the section with Possessions and Home (figure 7.4). Home indicates the Location the Investigator starts in at the beginning of the game, or when they enter the game (for example, when a player starts using a new Investigator).

Possessions are Skills, Items, money, Spells, and Allies the Investigator starts the game with. Each Investigator starts with a certain amount of money, and a number of additional fixed items, like a Colt .45. The Investigator also starts with a number of random possessions. These are drawn from the appropriate deck at the beginning of the game. Thus, this Investigator starts with $1 and a Clue token, as well as a Common Item, a Unique
CHAPTER 7. EXAMPLE 2: INVESTIGATOR DESIGN AND CODE

Item, one Spell, and two Skill cards.

Figure 7.5: Detail of Amanda Sharpe’s skill tracks

Focus relates to the Skills, which are at the bottom of the sheet (figure 7.5). It will be noted that each Skill is paired with another Skill. At the beginning of the game, each player receives three ovals which are placed on these tracks. Thus, if this Investigator’s Fight skill is at 4, her Will skill is at 1. Further, note that each Investigator may have different tracks compared to other Investigators, or even compared to each of their own tracks. During the Upkeep phase, each player may adjust the Skills a number of spaces equal to the Investigator’s Focus. So, in this case, the player may adjust a maximum of either one pair by three spaces, one pair by two spaces and another pair by one space, or adjust all three by one space. The player need not use the entirety of the Investigator’s Focus, or indeed use any of it.

Each Investigator also has one or two paragraphs on the back of the sheet detailing their background, current life, and how they ended up investigating the events of the game. However, as this was not considered essential for the playing of the game, it was excluded from the schema.

7.2 Formal Design of the Investigator

As with the Monster, it was important before beginning to represent each element correctly. The elements identified from the sheets were the pairs of skills, the name, the occupation, the ability, the maximum sanity and stamina (and the current sanity and stamina), the focus (and how much had been used), the home location, the current location, and the Investigator’s Items, Allies, Spells, Skills, money, and Clue tokens, as well as if the Investigator had any retainers or was Blessed. From domain knowledge, other elements identified were the presence of a Bank Loan, whether the Investigator was Deputy or member of the Silver Twilight Lodge, the Investigator’s Monster Trophies and Gate Trophies, the amount of possessions (ie Items, Allies, Spells, and Skills) the Investigator had, and whether the
Investigator was delayed or not, and whether they’d been denied a bank loan. These are looked at in more depth in the next subsection.

7.2.1 Investigator State Schemas

As pairs of skills could be represented by two integer arrays and a pointer to the current value in both, pairs of skills were represented by an entity called a Skillpair. From comparing each Investigator, it was determined that no skill went below 0 and no skill went above 6, so these were set as bounds for the skill arrays. Thus, it was determined that the Investigator would need a set of three Skillpairs, which would be used throughout the Game. A non-empty sequence of Skillpairs was determined to be the best choice for this.

```
Skillpair
  top : seq_1 N
  bottom : seq_1 N
  pointer : N

  #top = #bottom
  0 ≤ pointer < #top
  ∀ x : top, y : bottom • x, y ∈ {0..6}
```

It was decided to represent the Monster Trophies as a bag, sorted by toughness. A bag is a data type which represents multiple entities. Thus, when a Monster Trophy of toughness 2 is added, another 2 is added to the bag. The benefit of a bag is that it allows for querying how many of a given element exist within it.

Having reviewed every Investigator, it was discovered that every Investigator’s home was a Location. However, as an Investigator can move to Other Worlds, Locations, Streets, and Lost in Time and Space, which are all Nodes, the Investigator’s location has to be a Node.

As an Investigator can have multiples of the same Items and Spells, the best thing to do for those is to simply represent each of them as a bag containing a data type called Stuff, which Items and Spells both used as an archetype. Skills and Allies, on the other hand, contain no duplicates, and so these were representable as normal sets.

The blessing was originally going to be a simple boolean, but then two would be needed, one for being Blessed and another for being Cursed, where neither could be true at once. The easier method was to store the minimum number the Investigator’s player needed to roll on a die to succeed at any kind of check. By default, this is 5, i.e. a roll on a six-sided die of 5 or 6 results in a success. When Blessed, the success variable is set to 4. Conversely, if Cursed, the success variable would be 6. If the success variable was set to 4 when
the Investigator was Cursed, or was set to 6 when Blessed, the variable would be reset to 5.

When an Investigator is delayed, they spend one round incapable of moving or having any encounters, another round incapable of moving, but capable of having encounters, and are then able to move in the following round. So, it was decided to represent delaying as a natural number from 2 to 0. If the delay variable is 0, the Investigator may move, if it’s 1, the Investigator may have an encounter but not move, and if it’s 2, the Investigator may not move or have an encounter.

Every other variable was fairly self-explanatory, with name, occupation and ability being represented by Strings; maximum (and current) sanity and stamina, current focus, money, Gate Trophies, Clues, card number, and retainers being represented by natural numbers (with focus itself being a non-zero natural number); and the flags denied, lodge, deputy, and loan being boolean variables.

The next step, deciding on the constraints, was easier. There are only three Skillpairs, so the skillpairs set must contain three Skillpairs. Current sanity must be less than or equal to maximum sanity, and the same for stamina. Finally, as already stated, the success variable can only be a value between 4 and 6, and the delay variable can only be a value between 0 and 2.

```
Investigator
name, occupation, ability : STRING
maxsan, maxsta, cursan, cursta, curfocus, money, gatetrophies,
    clues, cards, delayed, success, retainers : N
home : LOCATION
location : NODE
focus : N1
skillpairs : seq1 SKILLPAIR
montrophies : bag N
skills : P SKILL
allies : P ALLY
spells, items : bag STUFF
lodge, deputy, bankloan, denied : BOOL

#skillpairs = 3

cursan ≤ maxsan

cursta ≤ maxsta

delayed ∈ {0, 1, 2}

success ∈ {4, 5, 6}
```
### 7.2.2 Initialisation Schemas

The following is the initialisation of the Investigator. Those variables that are set by inputs are set, while those that are derived from said inputs or are set to default values are set as such as well. The former occur near the beginning of the schema, while the latter occur near the end of the schema (with the exception of current Stamina, current Sanity, and current Focus). The “...” represents where ability’ = ability?, skills’ = skills?, and so on.

```plaintext
InitInvestigator
Investigator
name?, occupation?, ability? : STRING
maxsta?, maxsan?, focus?, money?, clues? : N
home? : LOCATION
skillpair1?, skillpair2?, skillpair3? : SKILLPAIR
skills? : P SKILL
allies? : P ALLY
spells?, items? : bag STUFF
blessed? : BOOL

name' = name?
occupation' = occupation?
maxsta' = cursta' = maxsta?
maxsan' = cursan' = maxsan?
home' = location' = home?
focus' = curfocus' = focus?
skillpairs' = ⟨skillpair1?, skillpair2?, skillpair3?⟩
money' = money?
...
items' = items?

if blessed? = TRUE
    then success' = 4
    else success' = 5

cards' = #skills + #allies + #spells + #items
gatetrophies' = delayed' = retainers' = 0
montrophies' = []

deputy' = FALSE ∧ lodge' = FALSE ∧ bankloan' = FALSE ∧ denied' = FALSE
```
7.2.3 Operational Schemas for the Investigator

This schema simply returns the values for each skill in the Skillpair that the pointer is currently pointing at.

\[
\text{GetSkills} \\
\text{\( \Sigma \text{Skillpair} \)} \\
\text{top!, bottom! : N} \\
\text{top!} = \text{top}(\text{pointer}) \\
\text{bottom!} = \text{bottom}(\text{pointer})
\]

This schema increases or decreases a given statistic (i.e. Stamina or Sanity) based on its inputs. If stat? is “Sta”, then Stamina, otherwise Sanity. If gain? is TRUE, then the current stat variable is increased by one, otherwise it’s decreased by one. It’s better to increase or decrease by one, as it allows for more control.

\[
\text{GainorLoseStat} \\
\Delta \text{Investigator} \\
\text{stat? : STRING} \\
\text{gain? : BOOL} \\
\text{if stat? \equiv} \text{Sta} \\
\text{then if gain \equiv} \text{TRUE} \\
\text{then cursta'} = \text{cursta} + 1 \\
\text{else cursta'} = \text{cursta} - 1 \\
\text{else if gain \equiv} \text{TRUE} \\
\text{then cursan'} = \text{cursan} + 1 \\
\text{else cursan'} = \text{cursan} - 1
\]

This schema either Blesses an Investigator, or Curses them. It should be noted that Blessings and Curses cancel each other out. If bless? is FALSE, the Investigator is being Cursed, and the success variable is checked. If it’s not 6, it means it must be either 5 (normal) or 4 (Blessed). Either way, the variable is increased by 1. Otherwise, it means the Investigator is already Cursed, and success remains the same.

If bless? is TRUE, however, then the Investigator is being Blessed, and the success variable is checked again. If it’s not 4, then either the hapless Investigator is Cursed (6) or returned to normal (5). Either way, the success variable is decreased. If, however, the Investigator’s success variable is 4, then the Investigator is already Blessed, and cannot become luckier.
This schema delays an Investigator. It’s remarkably straightforward. The reason, as stated before, that delayed is a natural number is because a delayed Investigator cannot act for one turn, cannot move for one turn, and then has no restrictions. These three states relate to the value of delayed: if 2, the Investigator has just become delayed, and cannot perform any actions. If 1, the Investigator can have an Encounter, but nothing more. This value is set after an Investigator steps into an Other World and has their encounter, since an Investigator must remain in an Other World for two rounds. The final value, of course, means there are no restrictions, and the Investigator may proceed as normal.

This schema, too, is self-explanatory. The Investigator gains clues, the Location is emptied of them.
This schema increases or decreases a chosen skill by one, depending on the value of direction?. If it’s ‘+’, then the pointer for the skillpair will be increased by one. Otherwise, it will be decreased by one. If type? is “Speed” or “Sneak”, the first skillpair will be dealt with; if it’s “Fight” or “Will”, the second; otherwise, the third.

\[
\text{FocusByOne}
\]
\[
\Delta \text{Investigator}
\]
\[
direction? : \text{CHAR}
\]
\[
type? : \text{STRING}
\]
\[
\text{if type?} = \text{Speed} \lor \text{type?} = \text{Sneak}
\]
\[
\quad \text{then if direction?} = +
\]
\[
\quad \quad \text{then skillpairs'(1).pointer} = \text{skillpairs(1).pointer} + 1
\]
\[
\quad \quad \text{else skillpairs'(1).pointer} = \text{skillpairs(1).pointer} - 1
\]
\[
\quad \text{else if type?} = \text{Fight} \lor \text{type?} = \text{Will}
\]
\[
\quad \quad \text{then if direction?} = +
\]
\[
\quad \quad \quad \text{then skillpairs'(2).pointer} = \text{skillpairs(2).pointer} + 1
\]
\[
\quad \quad \quad \text{else skillpairs'(2).pointer} = \text{skillpairs(2).pointer} - 1
\]
\[
\quad \text{else if direction?} = +
\]
\[
\quad \quad \text{then skillpairs'(3).pointer} = \text{skillpairs(3).pointer} + 1
\]
\[
\quad \quad \text{else skillpairs'(3).pointer} = \text{skillpairs(3).pointer} - 1
\]

If the Investigator is not delayed in any way, and either the location is a number of spaces away less than or equal to the Investigator’s Speed skill, or the Investigator is the Deputy (allowing travel to anywhere on the board instantly), then the Investigator moves to the new location. Otherwise, the Investigator stays put.

\[
\text{Move}
\]
\[
\Delta \text{Investigator}
\]
\[
\text{location?} : \text{NODE}
\]
\[
\text{distance?} : \text{N}
\]
\[
\text{if delayed} = 0
\]
\[
\quad \text{then if skillpairs(1).bottom(skillpairs(1).pointer)} \leq \text{distance?} \lor \text{deputy} = \text{TRUE}
\]
\[
\quad \quad \text{then location'} = \text{location?}
\]
\[
\quad \quad \text{else location'} = \text{location}
\]
\[
\quad \text{else location'} = \text{location}
\]
Provided an Investigator has not been previously marked as denied, which occurs if they roll a 1, 2, or 3 during their upkeep while they have a bankloan, and they don’t already have a loan, they may get a bankloan and instantly receive $10.

```plaintext
LoanGET
ΔInvestigator
if bankloan = FALSE ∧ denied = FALSE
  then bankloan' = TRUE ∧ money' = money + 10
else bankloan' = bankloan
```

If they have a loan, they can pay it off at any time by spending $10. If they do, they discard the loan, and are eligible to receive a new loan when they want.

```plaintext
PayOffLoan
ΔInvestigator
if money ≥ $10
  then money' = money − 10 ∧ bankloan' = FALSE
else money' = money
```

Finally, the schema on the next page sets up the Investigator at the beginning of each round. First, it unexhausts every Skill, Spell, or Item that has been exhausted. Next, it resets the current focus counter back to the same value as focus. Next, it decreases the delayed variable by 1. It then checks how many retainers the Investigator has, giving them $2 for each retainer. It checks the retainer? input, which is a set of numbers between 1 and 6 culled from rolling dice, and, for each 1, reduces the amount of retainers the Investigator has. It then checks if the Investigator’s loan? input is between 1 and 3. If it is, and the Investigator has a loan, then it either reduces the Investigator’s money by $1, or it removes all their items. Either way, the Investigator’s loan is removed, and they are denied, which prevents them from getting another loan for the rest of the game.
\[
\text{RoundStart}
\]

\(\Delta\text{Investigator}\)

\(\text{retainer}\? : \mathbb{P}N\)

\(\text{loan}\? : \mathbb{N}\)

\(\#\text{retainer}\? = \text{retainer}'\)

\(\forall x : \text{retainer}\? \mid x \in \{1..6\}\)

\(\text{loan}\? \in \{1..6\}\)

\(\text{if } \#\text{skills}' \neq 0\)

\(\text{then } \forall i : 1..\#\text{skills}' \cdot \text{skills}'(i).\text{exhaust} = \text{FALSE}\)

\(\text{else } \text{skills}' = \text{skills}\)

\(\text{if } \text{dom spells}' \neq 0\)

\(\text{then } \forall i : 1..\text{dom spells}' \cdot \text{spells}'(i).\text{exhaust} = \text{FALSE}\)

\(\text{else } \text{spells}' = \text{spells}\)

\(\text{if } \text{dom items}' \neq 0\)

\(\text{then } \forall i : 1..\text{dom items}' \cdot \text{items}'(i).\text{exhaust} = \text{FALSE}\)

\(\text{else } \text{items}' = \text{items}\)

\(\text{curfocus}' = \text{focus}\)

\(\text{if } \text{delayed} \neq 0\)

\(\text{then } \text{delayed}' = \text{delayed} - 1\)

\(\text{else } \text{delayed}' = \text{delayed}\)

\(\text{if } \text{retainer} \geq 1\)

\(\text{then } \text{money}' = \text{money} + \text{retainer} \cdot 2\)

\(\text{else } \text{money}' = \text{money}\)

\(\forall x : \text{retainer}\? \mid x = 1 \cdot \text{retainer}' = \text{retainer} - 1\)

\(\text{if } \text{loan}\? \in \{1..3\} \land \text{bankloan} = \text{TRUE}\)

\(\text{then } \text{if } \text{money} \geq 1\)

\(\text{then } \text{money}' = \text{money} - 1 \land \text{bankloan}' = \text{FALSE} \land \text{denied}' = \text{TRUE}\)

\(\text{else } \text{items}' = \langle \rangle \land \text{bankloan}' = \text{FALSE} \land \text{denied}' = \text{TRUE}\)

\(\text{else } \text{loan}' = \text{loan}\)
7.3 The Investigator Implementation

As with the Monster, the design mapped fairly easily to the code. The enumerations were simply changed to Java enumerations, and state schemas became classes, with the initialisation schemas becoming the classes’ constructors, and the operational schemas becoming methods on the class.

First of all, the Skillpair implementation. Here is the class’s private attributes and constructor.

```java
private int[][] skills;
private int pointer;

public SkillPair(int[] top, int[] bottom, int current) throws ConstraintException, BoundingException {
    if (top.length > 4)
        throw new ConstraintException("Input SkillPair top skill; length is " + top.length);
    else if (bottom.length > 4)
        throw new ConstraintException("Input SkillPair bottom skill; length is " + bottom.length);
    else {
        this.skills[0] = top;
        this.skills[1] = bottom;
    }

    if (3 < current || current < 0)
        throw new BoundingException("Input SkillPair pointer; value is " + current);
    else
        this.pointer = current;
}
```

Listing 7.1: Skillpair attributes and constructor

There are three methods for this class. The first two are access methods, one for the pointer’s location, and the other which returns the values at each location. Since this is slightly more complicated than the average access method, it is included below.

```java
public int[] getValues() {
    int[] foo = {this.skills[0][this.pointer], this.skills[1][this.pointer]};
    return foo;
}
```

Listing 7.2: Method to retrieve the current Skill levels
Finally, the last method changes the pointer's value, which occurs when the character focuses during the Upkeep phase.

```java
public void changePointer(int newVal) {
    this.pointer = newVal;
}
```

Listing 7.3: Method to change the Pointer value

Next, the actual Investigator. As this was considered to be one of the actual actors in the game, it has a large amount of methods. First of all, the attributes.

```java
private String name;
private String occupation;
private int maxsan, maxsta, cursan, cursta;
private Location home;
private Node location;
private String ability;
private int focus, curfoc, money, gtrophies;
private SkillPair[] skillpairs = new SkillPair[3];
private ArrayList<Monster>[] mtrophies = new ArrayList[4]; // element 0 is 1 toughness monsters, etc.
private int clues;
private ArrayList<Skill> skills;
private ArrayList<Ally> allies;
private ArrayList<Spell> spells;
private ArrayList<Item> items;
private int cards, delayed, success;
private boolean deputy, lodge, bankloan, denied;
private int retainers;
```

Listing 7.4: Investigator attributes

In Java, an ArrayList is a special type that defines a dynamically-sized array. It can be passed a starting size on construction, or can be initialised empty. Elements are added on, increasing the size.

Here is the constructor. As the class has an inordinate amount of attributes, it has a large amount of arguments for the constructor.

```java
public Investigator(String n, String o, int san, int sta, String ab, int foc, int dollars, int cl, int retainer, boolean blessed,
    SkillPair pair1, SkillPair pair2, SkillPair pair3,
    ArrayList<Skill> sk, ArrayList<Ally> al, ArrayList<Spell> sp,
    ArrayList<Item> it) throws BoundingException, ConstraintException, NullException {
```

Listing 7.5: Investigator constructor declaration
Next is a test for nullity. This prevents important values from being null, which would cause havoc with the system. Note that the constructor checks if each individual SkillPair is null, then creates the array of SkillPairs.

```java
if (n == null)
    throw new_NullException("Input Investigator name");
else if (o == null)
    throw new_NullException("Input Investigator occupation");
else if (ab == null)
    throw new_NullException("Input Investigator ability");
else if (pair1 == null || pair2 == null || pair3 == null)
    throw new_NullException("Input Investigator SkillPair(s)");
else {
    this.name = n;
    this.occupation = o;
    this.ability = ab;

    SkillPair[] foo = {pair1, pair2, pair3};
    this.skillpairs = foo;
}
```

Listing 7.6: Constructor part 1: checking if any values are null

After this, the constructor tests to see if any numerical value falls outside of its predefined range. The starting money for each Investigator has more eccentric bounds than the others, so it needs a special section to test it.

```java
if (7 < san || san < 3)
    throw new_BoundingException("Input Investigator maximum sanity; value is " + san);
else if (7 < sta || sta < 3)
    throw new_BoundingException("Input Investigator maximum stamina; value is " + sta);
else {
    this.maxsan = san;
    this.maxsta = sta;
}
```

```java
if ((dollars != 0 || dollars != 1) && (10 < dollars || dollars < 4))
    throw new_ConstraintException("Invalid amount of money; value is " + dollars);
else if (retainer > 1)
    throw new_ConstraintException("Too many retainers; amount is " + retainer);
else {
    this.money = dollars;
    this.retainers = retainer;
}
```

Listing 7.7: Constructor part 2: ensuring values do not fall out of range
Finally, the remaining values are assigned to attributes, with all boolean attributes set to false, and all cur attributes set to equal their max values. The assignments below, however, are not as obvious:

```java
for (int i = 0; i < 4; i++)
    this.mtrophies[i] = new ArrayList<Monster>();

if (blessed)
    this.blessOrCurse(true);

this.cursan = this.maxsan;
this.cursta = this.maxsta;
this.curfoc = this.focus;
this.gtrophies = 0;
this.cards = this.skills.size() + this.allies.size() + this.spells.size()
    () + this.items.size();
this.delayed = 0;
```
The `getFocus` method is similar to the above method, except it doesn’t have a `stat` parameter, as the only choice is focus.

Another unusual access method is that for getting the current Skill values of a given `SkillPair`. As this is returning one of three possible values, it, too, requires an identifying parameter. This parameter can be one or the other of the Skills in a `SkillPair`, and the method will return that `SkillPair`.

```java
public int[] getSkillPair(String pair) throws ConstraintException {
    if (pair.equals("Speed") || pair.equals("Sneak"))
        return this.skillpairs[0].getValues();
    else if (pair.equals("Fight") || pair.equals("Will"))
        return this.skillpairs[1].getValues();
    else if (pair.equals("Lore") || pair.equals("Luck"))
        return this.skillpairs[2].getValues();
    else
        throw new ConstraintException(pair + " is not a recognised skill!");
}
```

Listing 7.10: Getting a specific pair of Skills

The next complicated access method involves returning the specific value of one Skill. The reason this method was complicated was because the data stored within the `SkillPair` was returned as a two-element array representing the current levels for each Skill. Therefore, the same theory as above applies, but the result of `SkillPair.getValues()` is examined closer to get the result.

```java
public int getSkill(String skill) throws ConstraintException {
    if (skill.equals("Speed"))
        return this.skillpairs[0].getValues()[0];
    else if (skill.equals("Sneak"))
        return this.skillpairs[0].getValues()[1];
    else if (skill.equals("Fight"))
        return this.skillpairs[1].getValues()[0];
    else if (skill.equals("Will"))
        return this.skillpairs[1].getValues()[1];
    else if (skill.equals("Lore"))
        return this.skillpairs[2].getValues()[0];
    else if (skill.equals("Luck"))
        return this.skillpairs[2].getValues()[1];
    else
        throw new ConstraintException(skill + " is not a recognised skill");
}
```

Listing 7.11: Getting the current value of a specific Skill

The last two methods are those for returning specific Items, either Common or Unique. Each Item has a boolean value called `unique`. If it’s true, then the Item is Unique. Otherwise, it’s a Common Item. This method returns Common Items the Investigator is
carrying; the method for returning Unique ones is identical, except it returns those items that are Unique.

```java
public ArrayList<Item> getCommonItems() {
    ArrayList<Item> foo = new ArrayList<Item>();
    for (int i = 0; i < this.items.size(); i++)
        if (!this.items.get(i).isUnique())
            foo.add(this.items.get(i));
    return foo;
}
```

Listing 7.12: Returning a list of Common Items held by the Investigator

For actual functional methods, only two are simple: the method that sets an Investigator’s denied flag, the method that sets an Investigator’s Lodge membership, and the method that increases the Investigator’s amount of clues by the number passed to the method.

One of the longer methods is the focus method, which accepts a character and a String as parameters, and returns nothing. The character can be either + or -, and the String must be the name of a Skill. Here are the first few lines:

```java
public void focus(char direction, String type) {
    if (this.focus > 0) {
        if (type.equals("Speed") || type.equals("Sneak")) {
            int oldVal = this.skillpairs[0].getPointer();
            if (direction == '+') {
                int newVal = oldVal++;
                this.skillpairs[0].changePointer(newVal);
            } else if (direction == '-') {
                int newVal = oldVal--;
                this.skillpairs[0].changePointer(newVal);
            }
        }
    }
}
```

Listing 7.13: First part of focus method

As long as the current amount of focus is over 0, then the pointer for the SkillPair type refers to is modified; either it’s increased if direction is +, or decreased if it’s -. There are two other if clauses, dealing with each of the other SkillPairs. At the end of the entire if (focus > 0) is the instruction this.focus--; and that ends the method.

Listing 7.14 is a method that will increase or decrease the current value of a given statistic. The first parameter defines which statistic to change, while the second determines whether to increase it or decrease it (as before, polarity can only be + or -).
CHAPTER 7. EXAMPLE 2: INVESTIGATOR DESIGN AND CODE

public void gainOrLose(String stat, char polarity) {
    if (stat.equalsIgnoreCase("Stamina"))
        if (polarity == '+')
            this.cursta++;
        else if (polarity == '-')
            this.cursta--;
    else if (stat.equalsIgnoreCase("Sanity"))
        if (polarity == '+')
            this.cursan++;
        else if (polarity == '-')
            this.cursan--;
}

Listing 7.14: Changing statistics

The next method is fairly simple: it will either bless or curse the Investigator. The only parameter is true if the Investigator is being blessed, and false otherwise. If the Investigator has already been blessed or cursed, then nothing happens when they are blessed or cursed. If they are cursed and then blessed, or vice-versa, their success number (i.e. the lowest number that needs to turn up on a die to signify success) is reset to 5, i.e. they succeed on a roll of 5 or 6. If the Investigator is not already cursed or blessed, then their success number is either raised (curse) or lowered (blessing).

public void blessOrCurse(boolean blessing) {
    if (blessing == true && this.success < 6)
        this.success++;
    else if (blessing == false && this.success > 4)
        this.success--;
}

Listing 7.15: Blessing or cursing an Investigator

This method relates to delaying. If the Investigator is delayed, they stay where they are for two rounds. They may have an encounter after the first, and move after the second. So, the delayed variable is a number from 0 to 2. If delay is 2, then the Investigator cannot move or have encounters. If it’s 1, they have encounters. If it’s 0, the Investigator may move again. The first method sets the delay attribute to 2.

public void delay() {
    this.delayed = 2;
}

Listing 7.16: Delaying and moving

The next method adds Monsters to the Monster Trophy list. As mentioned in the comment in the constructor, the 0th element is for 1 toughness monsters, and so on, so this method checks the Monster’s toughness, then subtracts 1 from it and stores it in that
element of the array.

```java
public void addMonsterTrophy(Monster m) {
    this.mtrophies[m.getToughness() - 1].add(m);
}
```

Listing 7.17: Getting a new Monster Trophy

Next is the method to actually spend these Monster Trophies. This takes in the amount of toughness to spend and a list of Monsters in the set of Trophies that the player wants to spend. If the toughness of these Monsters isn’t greater than or equal to the toughness passed, the method returns -1. Otherwise, it removes the selected Monsters, then returns how many times the transaction paid for, i.e. for toughness 4, 8 toughness worth of Monsters returns 2.

```java
public int spendMonsterTrophies(int toughness, ArrayList<Monster> trophies) {
    int total = 0, amount = 0;
    for (int i = 0; i < trophies.size(); i++)
        total += trophies.get(i).getToughness();

    if (total >= toughness) {
        for (int i = 0; i < trophies.size(); i++) {
            Monster foo = trophies.get(i);
            this.mtrophies[foo.getToughness() - 1].remove(foo);
        }
        amount = toughness / total;
    } else
        amount = -1;

    return amount;
}
```

Listing 7.18: Spending Monster Trophies

The methods in listing 7.19 are involved with loans. The first describes what happens when the Investigator gets a loan, while the second describes the Investigator trying to pay off the loan.
public void getLoan() {
    this.money += 10;
    this.bankloan = true;
}

public void payLoan() throws ConstraintException {
    if (this.money >= 10) {
        this.money -= 10;
        this.bankloan = false;
    } else throw new ConstraintException("Not enough money to pay off loan!");
}

Listing 7.19: Getting and paying off loans

This method moves the Investigator to the location passed as a parameter. If the Investigator is in no way delayed, and the Investigator is inside an Other World, then the Investigator is moved to an open Gate on the Board that leads to this Other World. Otherwise, if the the distance to the target Location is less than or equal to the Investigator’s Speed, then the Investigator’s Location is changed to the new one. If this Location has a Gate, the Investigator is then moved into the Other World, and their delay flag is set to 1.

public void move(Node loc, int dist) {
    if (this.delayed == 0) {
        if (this.location.getClass().toString().contains("OtherWorld"))
            this.location = ((OtherWorld) this.location).getSource().get(0);

        try {
            if (dist <= this.getSkill("Speed")) {
                this.location = loc;

                if (((Location) loc).getPortal() != null) {
                    this.location = ((Location) loc).getPortal();
                    this.delayed = 1;
                }
            }
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.err.println(e.getCause());
            e.printStackTrace();
        }
    }
}

Listing 7.20: Moving the Investigator
This method is called at the beginning of each round. It resets the Investigator’s current Focus, decreases their delay variable by 1, unexhausts their Items, Spells, and Skills, and gives them any money they receive from being the Deputy.

```java
public void roundStart() {
    this.curfoc = this.focus;

    for (int i = 0; i < this.items.size(); i++)
        if (this.items.get(i).isExhausted())
            this.items.get(i).exhaust();

    for (int i = 0; i < this.spells.size(); i++)
        if (this.spells.get(i).isExhausted())
            this.spells.get(i).exhaust();

    for (int i = 0; i < this.skills.size(); i++)
        if (this.skills.get(i).isExhausted() && this.skills.get(i).isEffect())
            this.skills.get(i).exhaust();

    if (delayed != 0)
        delayed--;

    if (this.deputy)
        this.money++;}
```

Listing 7.21: Resetting the Investigator at the beginning of a round

If the Investigator has any retainers, this method should be called. The parameter r is an array of numbers between 1 and 6, with one element for each retainer the Investigator has, i.e. if the Investigator has three retainers, the array will contain three rolls. For each element, the Investigator receives money. If the element is 1, then the Investigator loses a retainer.

```java
public void checkRetainers(int[] r) {
    for (int i = 0; i < this.retainers; i++) {
        this.money += 2;
        if (r[i] == 1)
            this.retainers--;}
}
```

Listing 7.22: Checking retainers
Finally, this method checks an Investigator’s loan. The parameter l is a roll from 1 to 6. If it is 1, 2, or 3, then the Investigator has failed to pay off their loan. If they have any money, they lose $1. If they have no money, they lose all their Items. Either way, the Investigator loses their loan, and their denied flag is set to true, meaning that they will be unable to get a loan for the rest of the game.

```
public void checkLoan(int l) {
    if (l == 1 || l == 2 || l == 3) {
        if (this.money >= 1)
            this.money--;
        else
            this.items.clear();
    }
    this.bankloan = false;
    this.denied = true;
}
```

Listing 7.23: Checking the Investigator’s loan

### 7.4 Summary

The Investigator, unlike the Monster from the previous chapter, is remarkably more complex, due to being one of the actors in the Game (the other, as mentioned previously, being the Game itself).

Originally, the spending Monster Trophies method was very difficult, due to attempting to account for all possible variations of spending. For example, spending 5 toughness worth of Monster Trophies could be one three-toughness and two one-toughness Monsters, or one three-toughness and one two-toughness Monsters, etc. As a result, the original schema to deal with this was removed due to being too lengthy, but was accidentally never rewritten. Thus, the programmed version was written without any access to formally designed specifications. Eventually, it was decided to check if the Monsters picked by the player would have enough toughness, rather than telling the player all legal combinations, thus putting the onus on choosing legal values on the player rather than the code.

Another difficulty, which may have been clear from reading the Investigator schema and the Investigator class carefully, is that the Monster Trophies were changed from a bag of natural numbers to an array of ArrayLists of Monsters. This is because, during implementation, it was remembered that Monsters spent by Investigators were returned to the Monster Cup, and thus a record of the Monsters had to be kept. This was an oversight, rather than an actual error.

The next chapter will discuss an additional component created for the final game: a networked real-time chat system.
Chapter 8

A Description of the Chat System

As previously mentioned, to facilitate the play of the game, it was decided to create a real-time chat system which implemented a GUI. A working system was created, and forms the subject of this chapter.

The chat system allows for secure connections, with each Client (namely, each player in the game) having a unique ID assigned to them, which permits Clients to disconnect and reconnect, and prevents unauthorised users from entering the channel.

8.1 Authorisation Server

The Authorisation Server is responsible for maintaining the list of unique IDs. When it is first run, it creates eight SecureRandom variables. It stores these in a two-column, eight-row array of Strings. The first element of each row is the SecureRandom, while the second is the String “No”. It stores the unique IDs in a file, then waits for the Chat Server to connect (it assumes that, if it hasn’t been run yet, neither has the Chat Server).

If the Authorisation Server has already run, it reads the data out of the file, and stores it in its array of unique IDs. It then waits for a Client to connect and request a unique ID.

8.2 Chat Server

The Chat Server package contains four classes: the Chat Server proper, the ServerThread class, the ChatProtocol class, and the Observer class. Each of these shall be discussed in turn.

---

1A SecureRandom is a Java data type found in a Java library. Namely, it’s a 36-character random string containing the digits 1-9, the letters a-f, and hyphens
CHAPTER 8.  A DESCRIPTION OF THE CHAT SYSTEM

The Chat Server is responsible for maintaining the threads and ServerSocket that the Clients connect to. It does very little on its own, bar handing off all connections to an Observer object. The Server also creates an array of ServerThreads. This array contains only eight elements.

If the Chat Server hasn’t already been started, it connects to the Authorisation Server and sends a message consisting of the String “ACK”. The Authorisation Server then sends the Chat Server the first element in the unique IDs table. The Chat Server sends it straight back in order to be sure that it hasn’t been corrupted in transit. The Authorisation Server checks it; if it’s not OK, it resends the unique ID. If it’s correct, it sends back “ACK”. The Chat Server then writes the verified ID into its own array, then sends “ACK” itself. The Authorisation Servers sends it the next unique ID, and so on. When the Chat Server has the maximum number of IDs, it disconnects from the Authorisation server, then writes the array to an external file.

If the Chat Server has been run already, it reads the unique IDs from the file it maintains.

Either way, the Chat Server then initialises its Observer object, passing the Observer the array of unique IDs, as well as the ServerThread array. The Chat Server then idles, waiting for a Client to connect.

The ServerThread accepts the connection and initialises a ChatProtocol object. Other than that, it mostly calls methods from the Observer passed during construction. The only method it has that is important is a method that sends a message to the Client its connected to.

The ChatProtocol, on the other hand, is very important. It takes input from the ServerThread, parses it, then returns it to the ServerThread that created it. Its output is what appears on every Client’s screen after a message is sent.

Finally, the Observer. The Observer created by the Chat Server is passed to every single ServerThread on construction. It checks if a Client is using a correct unique ID, assigns ServerThreads to every Client, sends the Client’s output to every single other Client that’s connected to it, and, when the Client quits, it kills its ServerThread.

8.3 Client

The Client is the graphical front-end for the the system. It is a simple construct, consisting of two textboxes and a menubar (see figure 8.1).
When the Client is started, it checks if it’s been run before\textsuperscript{2}. If it hasn’t run before, it connects to the Authorisation Server, which sends it the first unique ID with a “No” beside it. The Server then changes the String to “Yes”. The Client, meanwhile, stores this unique ID as an attribute and in its settings file, and then connects to the Chat Server.

If the Client has already run, it simply pulls out the unique ID from the file, as well as any nickname that had been assigned previously, then connects to the Chat Server.

The Client has its own inbuilt text parser, which is run on input from the user, then displayed on the upper screen. This parsed text is then written to the Client’s logfile, as is all input from the socket.

The menubar contains two options: File and Help. The former contains only one option: Quit. The latter also contains only one option, which is a list of acceptable commands for the program.

8.4 How It Works

When the Client connects to the Chat Server, it calls up a method in its Observer to assign a ServerThread to the Client. The ServerThread then parses the first input the Client sends using its ChatProtocol object, which is the Client’s unique ID and, if it has

\footnotetext[2]{When writing about something “checking if it’s been run before”, this consists of the class checking for its settings file. If the file is empty or nonexistant, it hasn’t run before}
one, nickname. After doing this, the Observer checks if the unique ID is valid and not already in use. If both of these hold, it sets the entry in its unique ID table to “Yes”. The ChatProtocol then assigns a nickname to the Client if it doesn’t already have one; this is sent back to the Client and written to the Client’s settings file. If no exception have been thrown by this stage, and nothing has crashed, the GUI is started up, the name of the window being changed to include the Client’s nickname.

When the Client enters text in the lower textbox and hits Enter, the message is sent, unparsed, over the socket. It is then parsed inside the Client, then written to its logfile and upper textbox. When the message arrives in the ServerThread, it calls on the ChatProtocol to parse the message correctly. After it has done so, the ServerThread calls an Observer method to send the message to every other Client connected to the Chat Server. When the Client receives any input from the ServerThread, it immediately writes it to its logfile and its upper textbox.

The ChatProtocol parses input in different ways. If the message is a simple message, the ChatProtocol prefixes the message with the Client’s nickname and a colon. Namely, if the Client is called Bob, then it will return “Bob: message”. If the message is an action (ie a String consisting of “/me action”), it will return the String “Bob action”. If the message is a nickname change, ie “/nick KB”, it will return the String “Bob has changed his name to KB”, and change the nickname variable it has stored. Finally, if the message is a quit message, ie “/quit” or “/quit Bye”, it will return either “Bob has quit” or “Bob has quit (reason: Bye)” respectively.

If the Client quits, either by using /quit or by clicking Quit in the File menu, the ServerThread will close all sockets, the Observer will then free up the UID, and the Client will close all sockets and Stream objects, then close the GUI and stop running. If the Client is ever restarted, the Observer will assign it to the same ServerThread as before.

8.5 Summary

The Chat System was designed to be used as an aid for players in order to allow them to communicate in-game and come up with strategies. It is currently fully functional. Unlike the rest of the project, it was not formally specified. This was because of a concern that it relied far too much on concurrency to be accurately modelled using Z.
Chapter 9

Conclusions

In this chapter, it is proposed to conclude by discussing the successes and issues that occurred with Z in general and this project in particular. There will also be a reflection on the "myths" covered in chapter 4. As a learning exercise, there will also be a reflection on what, in hindsight, might have been done differently.

9.1 Benefits of Using Z

Beyond its complexity and due to a familiarity with the game, this game was chosen as a case study because it uses an intricate rule set combined with straightforward mathematics. When it came to designing the schemas, it was found that, following analysis (examples of which have been seen in the previous two chapters), that they were reasonably straightforward to design. The only three data types needed were String, integer, and boolean, and all of those were present.

This ease of design was also noted when implementation commenced on the components of the game. Generally, a model can be roughly translated into a class in Java (however, an exception is considered in 9.3 below). Thus, when a model called for a String attribute, so too did the implementation of that model. If new models or enumerations needed to be designed in Z, identical classes and, obviously, enumerations were created in Java. The basic data types in the models mapped exactly to Java’s data types, i.e. string mapped to String, etc.

Comparing the use of formal design to any method not using any form of design or model, the use of formal methods proved to be a welcome change, allowing the architecture of each component to be clearly seen, as well as how it interacted with its sub-components, and without the need to worry about any issues which might occur in the components. Inevitably, some omissions occurred in the design of the schema, but usually this was not too onerous to correct in both Z and Java.
The constraints placed on the game components were found to be relatively simple to represent in the models. Instead of entering upper and lower bounds, which may not include legal elements that fall outside of the bounds, Z allowed for the creation of a set of values which are legal. Z also permitted size limits to be placed on arrays, and preconditions for variable assignment. While they were comparatively easy to represent in Z, however, these constraints and preconditions were somewhat harder to implement in Java.

Finally, undertaking the project did result in a number of unanticipated achievements, including gaining a working knowledge of LATEX, Java sockets and GUI design using the GroupLayout layout manager.

9.2 Drawbacks of Using Z

Despite its numerous benefits, several challenges were encountered which arose specifically from the use of Z.

Firstly, it had to be realised that schema conjunction is not the same as inheritance. Schema conjunction, as previously mentioned, is when a schema is imported into another schema, allowing access to all of the variables of that schema. However, while this simulates object-oriented inheritance, it is in fact more akin to simply creating variables with those names and types in your current schema; namely, schema conjunction is a way to make schemas neater and simpler without sacrificing any of the functionality. Thus, in the schemas, when a data type was created which used variables from another data type as a form of primitive inheritance, it was necessary to create a type variable, while Java allowed for the use of inheritance with the only requirement being to check the object’s Class attribute.

Another issue which arose was that it was difficult to see how the parts of the game came together. The game has been described as a “terrible alien intelligence” (Florence, 2010). Thus, all of the pieces need to work together, like cogs in a machine or cells in a body. However, due to inexperience with Z, or perhaps a misconception of the results, the way the schemas would work together proved difficult to visualise. For example, attempting to reconcile how Monsters and Locations would react to each other involved a considerable period of time.

Finally, an issue that became a central problem: Z is a model-based specification language. Thus, it handles components all accepting input from each other without any difficulty. However, its primary weakness is that it cannot support any form of concurrency modelling. Since the idea for this project was to create a program that would allow users on multiple machines to play with each other, this was a significant issue, which it was, perhaps naively, concluded could be resolved merely by proceeding with the implementation.
9.3 Consideration of Hall’s “Myths”

In Chapter 3, seven myths of formal methods were considered (Hall, 1990). Here, those myths relevant to the project are reexamined as a result of the experiences gained in this project.

**Formal methods leads to faultless software**  As outlined in Chapter 7, while the schema used a bag of natural numbers to represent Monster Trophies, the Java implementation uses an array of ArrayLists of Monsters, which simulates a bag of Monsters. This change was considered more useful due to further research. Namely, the difference shows the result of correcting human error. This shows that, as long as human error exists, nothing can be declared as perfect.

**Formal methods is used only to validate code**  In this project, Z notation was used to design the schemas, but never to test them, as it was considered unnecessary. Thus, formal methods was not used at all for validation, confirming this as myth.

**Only safety-critical systems benefit from formal methods**  Had formal methods not been used, this system would most likely not have had any implementation. While formal methods has more general application on team projects, as would hold for safety-critical systems, it can still be used by a single programmer for small projects.

**Complex mathematics is always involved**  This system required nothing more complex than set comprehension, which is not overly complex to learn. So, this is also confirmed as a myth.

The remainder of these myths (increasing development cost, incomprehensible to clients, and not used outside of universities) are outside the scope of this project.

9.4 Future Work

Although this project comprised considerable effort and extensive work (as reference to Appendix A can confirm), a working project was unfortunately never fully completed. Although desirable, this was not the primary goal of this project. Consequently, there follows a consideration of what might have been done differently if starting this project again.

As previously mentioned, Z is incapable of modelling concurrency, which is a significant part of the original design. As a result, it was decided to try to program the concurrency on the fly. Another specification language called Circus was identified, which combines the model-based Z with the process-based CSP (Communicating Sequential Processes), the latter of which is used for modelling concurrency (Woodcock & Cavalcanti, 2001). With hindsight, it might have been more fruitful to have started the project as before, i.e. modelling the components, then implementing them in Java. However, after implementing them in code, rather than proceed to try and program the entire game, a better approach...
would have been to then start using Circus to design the concurrency aspect of the game, only implementing everything else after ensuring that the concurrency was properly designed.

It will have been noted that the discussion has focused on designing and implementing the components of the game, but has not mentioned the actual rules. This is because, as with concurrency, modelling the rules was deferred in the belief that it would be a trivial task to do in Java. Unfortunately, a number of unanticipated problems arose and hampered progress on the development of the software. Instead, a better strategy would have been to formally design the rules the game would use instead of anticipating resolution in proceeding with the implementation.

9.5 Final Thoughts

Z is a remarkable tool for programming. One issue encountered was that, if the programmer is used to less design-oriented programming styles (such as build-and-fix), the programmer will be unused to knowing exactly how the code will look before starting, and may waste time constantly check if they’re following the design perfectly.

However, the actual design work for such a large project is remarkably large, taking up nearly thirty pages (see Appendix A for all the schemas). As a result, the sheer scale of the project was, at times, most daunting. It was concluded that formal methods, and specifically Z notation, are far better when there’s a larger development team, rather than just one person, and only for smaller-scale projects where limited to a single designer.

Within this project, as outlined above, a lot of coding was undertaken based on formal design, whilst some was not. While the one section that was not formally designed (namely, the chat system) operates acceptably (possibly due to the smaller size of this subsystem), the remainder of the project struggled somewhat to reach implementation. This project, therefore, could be seen as an example of the benefits of formal methods and the disadvantages of coding on-the-fly.

In conclusion, Z was found to be a remarkable aid to programming, and will most likely be used again on smaller projects. However, on larger solo projects, it was concluded that it would be more appropriate to use software engineering, but not Z modelling, as the greater the number of objects to be modelled, the greater the size of the schemas that have to be implemented, and the greater the size of the code.
Bibliography


Appendix A

Schemas In Full

A.1 Useful Free Variables

\[
\text{COLOUR ::= Black | Purple | Blue | Brown | Red | Yellow} \\
\text{  | Green | Orange | White}
\]

\[
\text{SYM ::= Circle | Crescent | Diamond | Hexagon | Plus} \\
\text{  | Slash | Square | Star | Triangle}
\]

\[
\text{DEFENCES ::= PResistance | PImmunity | MResistance | MImmunity} \\
\text{  | Nightmarish⟨⟨1⟩⟩ | Overwhelming⟨⟨1⟩⟩ | Ambush}
\]

A.2 Nodes, Locations, Streets, and Others

A.2.1 State Schemas

<table>
<thead>
<tr>
<th>Lost Node</th>
<th>monsters = NULL</th>
<th>name = Lost in Time and Space</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Sky Node</th>
<th>investigators = NULL</th>
<th>name = Sky</th>
</tr>
</thead>
</table>
Outskirts
limit : \mathbb{N}
contents : \mathbb{P} \text{MONSTERS}
#contents \leq \text{limit}

Node
name : STRING
monsters : \mathbb{P} \text{MONSTER}
investigators : \mathbb{P} \text{INVESTIGATOR}

Street
Node
activity : BOOL
acttext : STRING
colour : COLOUR

Location
Street
portal : \text{OTHERWORLD}
clues : \mathbb{N}
sealed : BOOL
unstable : BOOL

if sealed \equiv \text{TRUE}
  then portal = \text{NULL}
else if portal \neq \text{NULL}
  then clues = 0
else clues = clues

OtherWorld
Node
source : \text{LOCATION}
colours : \mathbb{P} \text{COLOUR}

#colours \equiv 2 \lor #colours \equiv 4
\forall c : \text{colours} \mid c \in \{ \text{Blue}, \text{Red}, \text{Yellow}, \text{Green} \}
Connection
Location
Street
\( \text{start} : \text{NODE} \)
\( \text{end} : \text{NODE} \)

Neighbourhood
Street
Location
locations : \( \mathcal{P} \) LOCATION
hub : STREET
directed : \( \mathcal{P} \) CONNECTION
fromlost : \( \mathcal{P} \) CONNECTION
fromsky : CONNECTION
adjacent : \( \mathcal{P} \) ADJACENCY

\( \forall x : \text{locations} \mid x.\text{colour} \equiv \text{hub.\colour} \)
\( \forall x : \text{directed} \mid x.\text{start.\colour} \equiv x.\text{end.\colour} \)
\#directed = \#locations
\#fromlost = \#locations + 1
\#adjacent \in \{ 2, 3, 4 \}

Adjacency
Neighbourhood
adjacent : \( \mathcal{P} \) NEIGHBOURHOOD

Board
Neighbourhood
white : \( \mathcal{P} \) CONNECTION
black : \( \mathcal{P} \) CONNECTION
undirected : \( \mathcal{P} \) ADJACENCY
sky : SKY
lost : LOST
outskirts : OUTSKIRTS
limited : BOOL
limit : \mathbb{N}

white \equiv \text{rev black}
A.2.2 Initialisation Schemas

**LostInit**

```
Lost

investigators' = { }
```

**SkyInit**

```
Sky

monsters' = { }
```

**OutskirtsInit**

```
Outskirts

players? : N

limit' = 8 - players?
contents' = { }
```

**StreetInit**

```
Street

name? : STRING
colour? : COLOUR

name' = name?
colour' = colour?
activity' = FALSE
acttext' = NULL
monsters' = NULL
investigators' = NULL
```
LocationInit
Location
name? : STRING
investigators? : P INVESTIGATOR
colour? : COLOUR
unstable? : UNSTABLE

name' = name?
investigators' = investigators?
colour' = colour?
unstable' = unstable?
activity' = NULL
acttext' = NULL
monsters' = NULL
sealed' = FALSE
portal' = NULL
if unstable? ≡ TRUE
  then clues' = 1
  else clues' = 0

OtherWorldInit
OtherWorld
name? : STRING
colours? : P COLOUR

name' = name?
colours' = colours?
investigators' = monsters' = source' = NULL

ConnectionInit
Connection
start? : NODE
end? : NODE

start' = start?
end' = end?
**NeighbourhoodInit**

**Neighbourhood**

locations? : \( \mathbb{P} \) LOCATION  
hub? : STREET  
lost? : LOST  
sky? : SKY  
adjacent? : \( \mathbb{P} \) ADJACENCY

\[
\text{locations}' = \text{locations}?  
\text{hub}' = \text{hub}?  
\text{adjacent}' = \text{adjacent}?  
\text{directed}' = \{ \langle x, \text{hub}? \rangle \mid x \in \text{locations}? \}  
\text{fromlost}' = \{ \langle \text{lost}? , x \rangle \mid x \in \text{locations}? \} \cup \{ \langle \text{lost}? , \text{hub}? \rangle \}  
\text{fromsky}' = \langle \text{sky}? , \text{hub}? \rangle
\]

**BoardInit**

**Board**

white? : \( \mathbb{P} \) CONNECTION  
black? : \( \mathbb{P} \) CONNECTION  
undirected? : \( \mathbb{P} \) ADJACENCY  
sky? : SKY  
lost? : LOST  
players? : \( \mathbb{N} \)

\[
\text{white}' = \text{white}?  
\text{black}' = \text{black}?  
\text{undirected}' = \text{undirected}?  
\text{sky}' = \text{sky}?  
\text{lost}' = \text{lost}?  
\text{limit}' = \text{players}? + 3  
\text{outskirts} = \langle \text{players}? \rangle  
\text{limited} = \text{FALSE}
\]
A.2.3 Operational Schemas

- **ActivityHappens**
  - \( \Delta Street \)
  - \( text? : STRING \)
  - \( activity' = TRUE \)
  - \( acttext' = text? \)

- **ActivityOver**
  - \( \Delta Street \)
  - \( activity' = FALSE \)
  - \( acttext' = NULL \)

- **PlaceClue**
  - \( \Delta Location \)
  - \( clues' = clues + 1 \)

- **LoseClues**
  - \( \Delta Location \)
  - \( clues' = 0 \)

- **OpenPortal**
  - \( \Delta Location \)
  - \( location? : LOCATION \)
  - \( portal? : OTHERWORLD \)
  - \( monsters? : \mathbb{P} MONSTER \)
  - if sealed \( \equiv TRUE \)
    - then \( portal' = portal \)
    - else \( portal' = portal? \land portal'.source = location? \land clues' = 0 \)
  - \( monsters' = monsters \cup monsters? \)

- **ClosePortal**
  - \( \Delta Location \)
  - \( portal? : OTHERWORLD \)
  - \( portal'' = NULL \)
  - \( portal?.source = NULL \)
SealGate

\[ \Delta Location \]
\[ \Delta GreatOldOne \]
\( \text{sign}? : BOOLEAN \)

\( \text{sealed}' = \text{TRUE} \)
\( \text{portal}' = \text{NULL} \)

\[ \text{if sign}? \equiv \text{TRUE} \]
\[ \quad \text{then doomtrack}' = \text{doomtrack} - 1 \]
\[ \quad \text{else doomtrack}' = \text{doomtrack} \]

InvestigatorEnters

\[ \Delta Node \]
\( \text{investigator}? : \text{INVESTIGATOR} \)

\( \text{investigators}' = \text{investigators} \cup \{ \text{investigator}'? \} \)

InvestigatorLeaves

\[ \Delta Node \]
\( \text{investigator}? : \text{INVESTIGATOR} \)

\( \text{investigators}' = \text{investigators} \setminus \{ \text{investigator}'? \} \)

MonsterEnters

\[ \Delta Node \]
\( \text{monster}? : \text{MONSTER} \)

\( \text{monsters}' = \text{monsters} \cup \{ \text{monster}'? \} \)

MonsterLeaves

\[ \Delta Node \]
\( \text{monster}? : \text{MONSTER} \)

\( \text{monsters}' = \text{monsters} \setminus \{ \text{monster}'? \} \)

AddMonster

\[ \Delta Outskirts \]
\( \text{monster}? : \text{MONSTER} \)

\( \text{contents}' = \text{contents} \cup \text{monster}? \)
A.3 Monsters

A.3.1 State Schemas

MOVE ::= Fast | Flying | Normal | Stationary | Unique

Horror
rating : Z
damage : Z
rating ∈ {−3..1}
damage ∈ {0..4}

Combat
rating : Z
damage : Z
rating ∈ {−4..1}
damage ∈ {1..4}
\textbf{Monster}
\begin{verbatim}
name, rules : STRING
location : NODE
abilities : iseq DEFENCES
awareness, toughness : \mathbb{Z}
movetype : MOVE
symbol : SYM
mask, endless, undead : BOOL
horror : HORROR
combat : COMBAT
\end{verbatim}
\begin{verbatim}
awareness \in \{-3..1\}
toughness \in \{1..4\}
if \#\text{defences} \geq 2
\quad then \{ PR, PI \} \cap \text{defences} \leq 1
\qquad \wedge \{ MR, MI \} \cap \text{defences} \leq 1
\quad else \text{defences} = \text{defences}
\end{verbatim}

\section{Initialisation Schemas}
\begin{verbatim}
\textbf{HorrorInit}
\textbf{Horror}
\textbf{rating} : \mathbb{Z}
\textbf{damage} : \mathbb{Z}
\textbf{rating}' = \textbf{rating}
\textbf{damage}' = \textbf{damage}
\end{verbatim}

\begin{verbatim}
\textbf{CombatInit}
\textbf{Combat}
\textbf{rating} : \mathbb{Z}
\textbf{damage} : \mathbb{Z}
\textbf{rating}' = \textbf{rating}
\textbf{damage}' = \textbf{damage}
\end{verbatim}
APPENDIX A. SCHEMAS IN FULL

---

MonsterInit
---

```plaintext
Monster
name?, rules? : STRING
awareness?, horrrrat?, horrrdmg?, toughness?, combatrat?,
    combatdmg? : Z
movetype? : MOVE
symbol? : SYM
mask?, endless?, undead? : BOOL
```  

```plaintext
name' = name?
lcoation' = null
awareness' = awareness?
movetype' = movetype?
symbol' = symbol?
rules' = rules?
mask' = mask?
endless' = endless?
undead' = undead?
horror' = ⟨horrorrat?, horrordmg?⟩
toughness' = toughness?
combat' = ⟨combatrat?, combatdmg?⟩
```

---

A.3.3 Operational Schema

```plaintext
SetLocation
---

```  

```
∆Monster
location? : NODE
```

```
location' = location?
```
### A.4 Great Old One

#### A.4.1 State Schema

```plaintext
GreatOldOne
atkrat : Z
attack : STRING
defences : iseq DEFENCES
power : STRING
start : STRING
worshippers : STRING
doomtrack : N

atkrat ∈ \{-∞, -10..0\}

if #defences ≡ 2
  then \{ PResistance, PImmunity \} ∩ defences ≤ 1
      ∧ \{ MResistance, MImmunity \} ∩ defences ≤ 1
else defences = defences

doomtrack ∈ \{0..14\}
```

#### A.4.2 Initialisation Schema

```plaintext
GreatOldOneInit
atkrat? : Z
attack? : STRING
defences? : iseq DEFENCES
power? : STRING
start? : STRING
worshippers? : STRING

atkrat' = atkrat?
attack' = attack?
defences' = defences?
power' = power?
start' = start?
worshippers' = worshippers?
doomtrack = 0
```
A.5 Investigator

A.5.1 State Schemas

```
Skillpair

\[ top : \text{seq}_1 \mathbb{N} \]
\[ bottom : \text{seq}_1 \mathbb{N} \]
\[ pointer : \mathbb{N} \]

\[ \#top = \#bottom \]
\[ 0 \leq pointer < \#top \]
\[ \forall x : top, y : bottom \ni x, y \in \{0..6\} \]
```

```
Investigator

name, occupation, ability : STRING
maxsan, maxsta, cursan, cursta, curfocus, money, gatetrophies,
  clues, cards, delayed, success, retainers : \mathbb{N}
home : LOCATION
location : NODE
focus : \mathbb{N}_1
skillpairs : \text{seq}_1 \text{SKILLPAR}
monotrophies : \text{bag} \mathbb{N}
skills : \mathbb{P} \text{SKILL}
allies : \mathbb{P} \text{ALLY}
spells, items : \text{bag} \text{STUFF}
lodge, deputy, bankloan, denied : BOOL

\[ \#\text{skillpairs} = 3 \]
\[ \text{cursan} \leq \text{maxsan} \]
\[ \text{cursta} \leq \text{maxsta} \]
\[ \text{delayed} \in \{0, 1, 2\} \]
\[ \text{success} \in \{4, 5, 6\} \]
```
A.5.2 Initialisation Schema

\[
\text{InvestigatorInit}
\]
Investigator
\[
\text{name}, \text{occupation}, \text{ability} : \text{STRING}
\]
\[
\text{maxsta}, \text{maxsan}, \text{focus}, \text{money}, \text{clues} : \text{N}
\]
\[
\text{home} : \text{LOCATION}
\]
\[
\text{skillpair1}, \text{skillpair2}, \text{skillpair3} : \text{SKILLPAIR}
\]
\[
\text{skills} : \mathbb{P} \text{ SKILL}
\]
\[
\text{allies} : \mathbb{P} \text{ ALLY}
\]
\[
\text{spells}, \text{items} : \text{bag STUFF}
\]
\[
\text{blessed} : \text{BOOL}
\]
\[
\text{name}' = \text{name}
\]
\[
\text{occupation}' = \text{occupation}
\]
\[
\text{maxsta}' = \text{cursta}' = \text{maxsta}
\]
\[
\text{maxsan}' = \text{cursan}' = \text{maxsan}
\]
\[
\text{home}' = \text{location}' = \text{home}
\]
\[
\text{focus}' = \text{curfocus}' = \text{focus}
\]
\[
\text{skillpairs}' = \langle \text{skillpair1}, \text{skillpair2}, \text{skillpair3} \rangle
\]
\[
\text{money}' = \text{money}
\]
\[
\text{ability}' = \text{ability}
\]
\[
\text{clues}' = \text{clues}
\]
\[
\text{skills}' = \text{skills}
\]
\[
\text{allies}' = \text{allies}
\]
\[
\text{spells}' = \text{spells}
\]
\[
\text{items}' = \text{items}
\]
\[
\text{if blessed} = \text{TRUE}
\]
\[
\quad \text{then success}' = 4
\]
\[
\quad \text{else success}' = 5
\]
\[
\text{cards}' = \#\text{skills} + \#\text{allies} + \#\text{spells} + \#\text{items}
\]
\[
\text{gatetrophies}' = \text{delayed}' = \text{retainers}' = 0
\]
\[
\text{montrophies}' = [\]
\]
\[
\text{deputy}' = \text{FALSE} \land \text{lodge}' = \text{FALSE} \land \text{bankloan}' = \text{FALSE} \land \text{denied}' = \text{FALSE}
\]
A.5.3 Operational Schemas

- **GetSkills**
  - `Ξ Skillpair`
  - `top!, bottom! : N`
  - `top! = top(pointer)`
  - `bottom! = bottom(pointer)`

- **GainorLoseStat**
  - `∆ Investigator`
  - `stat? : STRING`
  - `gain? : BOOL`
  - `if stat? ≡ Sta`
    - `then if gain ≡ TRUE`
      - `then cursta' = cursta + 1`
      - `else cursta' = cursta - 1`
    - `else if gain ≡ TRUE`
      - `then cursan' = cursan + 1`
      - `else cursan' = cursan - 1`

- **BlessorCurse**
  - `∆ Investigator`
  - `bless? : BOOL`
  - `if bless? ≡ TRUE`
    - `then if success ≠ 6`
      - `then success' = success + 1`
      - `else success' = success`
    - `else if success ≠ 4`
      - `then success' = success - 1`
      - `else success' = success`

- **Delay**
  - `∆ Investigator`
  - `delayed' = 2`
### APPENDIX A. SCHEMAS IN FULL

<table>
<thead>
<tr>
<th>Schema</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ClueGET</strong></td>
<td></td>
</tr>
</tbody>
</table>
ΔInvestigator  
ΔLocation  
clues? : N  

\[
\text{clues}' = \text{clues} + \text{clues}'  
\text{location}'.\text{clues} = 0
\] |
| **FocusByOne** | 
ΔInvestigator  
skillpair : SKILLPAIR  
direction? : CHAR  
type? : STRING  

\[
\text{if type}? = \text{Speed} \lor \text{type}? = \text{Sneak}  
\begin{align*}
\text{then if direction}? = + & \quad \text{then skillpairs}'(1).\text{pointer} = \text{skillpairs}(1).\text{pointer} + 1 \\
\quad & \quad \text{else skillpairs}'(1).\text{pointer} = \text{skillpairs}(1).\text{pointer} - 1 \\
\text{else if type}? = \text{Fight} \lor \text{type}? = \text{Will} & \quad \text{then if direction}? = + \\
\quad & \quad \text{then skillpairs}'(2).\text{pointer} = \text{skillpairs}(2).\text{pointer} + 1 \\
\quad & \quad \text{else skillpairs}'(2).\text{pointer} = \text{skillpairs}(2).\text{pointer} - 1 \\
\quad & \quad \text{else skillpairs}'(3).\text{pointer} = \text{skillpairs}(3).\text{pointer} + 1 \\
\quad & \quad \text{else skillpairs}'(3).\text{pointer} = \text{skillpairs}(3).\text{pointer} - 1
\end{align*}
\] |
| **Move** | 
ΔInvestigator  
location? : NODE  
distance? : N  

\[
\text{if delayed} = 0  
\quad \text{then location}' = \text{location}?  \\
\quad \text{else location}' = \text{location}
\] |
| **LoanGET** | 
ΔInvestigator  

\[
\text{if bankloan} = \text{FALSE}  
\quad \text{then bankloan}' = \text{TRUE} \land \text{money}' = \text{money} + 10  \\
\quad \text{else bankloan}' = \text{bankloan}
\] |
APPENDIX A. SCHEMAS IN FULL

\[ \text{RoundStart} \]
\[ \Delta \text{Investigator} \]
\[
\begin{align*}
\text{retainer}^? & : \mathbb{P} \mathbb{N} \\
\text{loan}^? & : \mathbb{N}
\end{align*}
\]
\[
\#\text{retainer}^? = \text{retainer}'
\]
\[
\forall x : \text{retainer}^? \mid x \in \{1..6\}
\]
\[
\text{loan}^? \in \{1..6\}
\]
\[
\text{if } \#\text{skills}' \neq 0 \\
\quad \text{then} \forall i : 1..\#\text{skills}' \bullet \text{skills}'(i).\text{exhaust} = \text{FALSE} \\
\quad \text{else } \text{skills}' = \text{skills}
\]
\[
\text{if dom spells}' \neq 0 \\
\quad \text{then} \forall i : 1..\text{dom spells}'. \bullet \text{spells}'(i).\text{exhaust} = \text{FALSE} \\
\quad \text{else } \text{spells}' = \text{spells}
\]
\[
\text{if dom items}' \neq 0 \\
\quad \text{then} \forall i : 1..\text{dom items}'. \bullet \text{items}'(i).\text{exhaust} = \text{FALSE} \\
\quad \text{else } \text{items}' = \text{items}
\]
\[
\text{curfocus}' = \text{focus}
\]
\[
\text{if delayed} \neq 0 \\
\quad \text{then } \text{delayed}' = \text{delayed} - 1 \\
\quad \text{else } \text{delayed}' = \text{delayed}
\]
\[
\text{if retainer} \geq 1 \\
\quad \text{then } \text{money}' = \text{money} + \text{retainer} \times 2 \\
\quad \text{else } \text{money}' = \text{money}
\]
\[
\forall x : \text{retainer}^? \mid x = 1 \bullet \text{retainer}' = \text{retainer} - 1
\]
\[
\text{if loan}^? \in \{1..3\} \land \text{bankloan} = \text{TRUE} \\
\quad \text{then if money} \geq 1 \\
\quad \quad \text{then } \text{money}' = \text{money} - 1 \land \text{bankloan}' = \text{FALSE} \land \text{denied}' = \text{TRUE} \\
\quad \quad \text{else } \text{items}' = \langle \rangle \land \text{bankloan}' = \text{FALSE} \land \text{denied}' = \text{TRUE} \\
\quad \text{else } \text{loan}' = \text{loan}
\]
## A.6 Skills, Allies, Spells, and Items

### A.6.1 State Schemas

**PHASE** ::= Movement | Any | NULL

| Skill       | name : STRING  |
|            | skill : STRING |
|            | sbonus : BOOL  |
|            | ceffect : BOOL |
|            | eeffect : BOOL |
|            | exhaust : BOOL |

| Ally       | name : STRING |
|           | saffected : iseq STRING |
|           | sbonus : N |
|           | onjoin : STRING |
|           | ondiscard : STRING |
|           | ongoing : STRING |

- #saffected ∈ \{ 0, 1, 2 \}
- sbonus ∈ \{ 1, 2 \}

| Stuff      | name : STRING |
|           | cardtype : STRING |
|           | phase : PHASE  |
|           | hands : N |
|           | exhaust : BOOL |

- cardtype ∈ \{ Item, Spell \}
- hands ∈ \{ 0, 1, 2 \}

| Spell      | Stuff |
|           | castmod : Z |
|           | sancost : N |

- castmod ∈ \{ −4.1, 4 \}
- sancost ∈ \{ 0, 1, 2 \}
ITYPE ::= Physical | Magical | Tome | NULL

**Item**

- **Stuff**
  - type : ITYPE
  - unique : BOOL
  - cost : \( \mathbb{N} \)

  \[ cost \in \{1..8\} \]

### A.6.2 Initialisation Schemas

**SkillInit**

**Skill**

- name? : STRING
- skill? : STRING
- sbonus? : BOOL

\[ name' = name? \]
\[ skill' = skill? \]
\[ sbonus' = sbonus? \]

\[ \text{if } sbonus? = \text{TRUE} \]
\[ \quad \text{then } e\text{ffect}' = \text{FALSE} \]
\[ \quad \text{else } c\text{ffect}' = \text{FALSE} \]

\[ \text{exhaust}' = \text{FALSE} \]
APPENDIX A. SCHEMAS IN FULL

AllyInit

Ally
name? : STRING
saffected? : iseq STRING
onjoin? : STRING
ondiscard? : STRING
ongoing? : STRING

name' = name?
saffected' = saffected?

if onjoin? ≠ NULL
    then ondiscard' = ongoing' = NULL ∧ onjoin' = onjoin?
else if ondiscard? ≠ NULL
    then onjoin' = ongoing' = NULL ∧ ondiscard' = on discard?
else on discard' = onjoin' = NULL ∧ ongoing' = ongoing?

if saffected? = ⟨Sanity⟩ ∨ saffected? = ⟨Stamina⟩ ∨ #saffected? = 2
then sbonus' = 1
else sbonus' = 2

SpellInit

Spell
name? : STRING
phase? : PHASE
hands? : N
castmod? : Z
sancost? : N

name' = name?
phase' = phase?
hands' = hands?
castmod' = castmod?
sancost' = sancost?
exhaust' = FALSE
cardtype' = Spell
APPENDIX A. SCHEMAS IN FULL

A.6.3 Operational Schemas

ExhaustItem
\[ \Delta \text{Item} \]
\[ \text{exhaust}' = \neg \text{exhaust} \]

ExhaustSpell
\[ \Delta \text{Spell} \]
\[ \text{exhaust}' = \neg \text{exhaust} \]

ExhaustSkill
\[ \Delta \text{Skill} \]
\[ \text{exhaust}' = \neg \text{exhaust} \]
A.7 Encounters

A.7.1 State Schemas

EncounterText
  location : NODE
  encounter : STRING

ArkhamEncounter
  colour : COLOUR
  locations : iseq ENCOUNTERTEXT
  \#locations = 2 \lor \#locations = 3
  \forall x : locations \mid x.colour = colour

OWEncounter
  colour : COLOUR
  locations : iseq EncounterText
  colour \in \{ \text{Red}, \text{Blue}, \text{Green}, \text{Yellow} \}
  \#locations = 3
  \forall x : locations \mid \exists y : x.location.colours \mid y = colour

A.7.2 Initialisation Schemas

EncounterTextInit
  EncounterText
  location? : NODE
  encounter? : STRING
  location' = location?
  encounter' = encounter?

ArkhamEncounterInit
  ArkhamEncounter
  colour? : COLOUR
  locations? : iseq ENCOUNTERTEXT
  colour' = colour?
  locations' = locations?
APPENDIX A. SCHEMAS IN FULL

A.8 Mythos Cards

A.8.1 State Schemas

\[ TTYPE ::= \text{Rumour} \mid \text{MEnvironment} \mid \text{UEnvironment} \mid \text{WEnvironment} \mid \text{Headline} \mid \text{NULL} \]

\[ \text{MythosText} \]
\[ \text{type} : TTYPE \]
\[ \text{abtext} : \text{STRING} \]
\[ \text{activity} : \text{NODE} \]

\[ \text{MythosCard} \]
\[ \text{MythosText} \]
\[ \text{gate} : \text{LOCATION} \]
\[ \text{clue} : \text{LOCATION} \]
\[ \text{black} : \text{P SYMBOL} \]
\[ \text{white} : \text{P SYMBOL} \]
\[ \text{gate} \neq \text{clue} \]
\[ \text{black} \neq \text{white} \]
\[ \#\text{black} \in \{0..3\} \]
\[ \#\text{white} \in \{0..3\} \]
A.8.2 Initialisation Schema

MythosCardInit
MythosCard
  type? : TTYPE
  abtext? : STRING
  activity? : NODE
  gate? : LOCATION
  clue? : LOCATION
  black? : P SYMBOL
  white? : P SYMBOL
  type' = type?
  abtext' = abtext?
  activity' = NODE
  gate' = gate?
  clue' = clue?
  black' = black?
  white' = white?
A.9 The Game

**Game Board**

- *mythosdeck*: \( \mathbb{P} \) MYTHOS
- *downtowndeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *easttowndeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *freihilldeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *merdistdeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *miskudeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *northsidedeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *rivertowndeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *southsidedeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *uptowndeck*: \( \mathbb{P} \) ARKHAMENCOUNTER
- *owdeck*: \( \mathbb{P} \) OWENCOUNTER
- *spelldeck*: \( \mathbb{P} \) SPELL
- *citemdeck*: \( \mathbb{P} \) ITEM
- *uitemdeck*: \( \mathbb{P} \) ITEM
- *allydeck*: \( \mathbb{P} \) ALLY
- *skilldeck*: \( \mathbb{P} \) SKILL
- *lodgedeck*: \( \mathbb{N} \)
- *deputyexists*: \( \text{BOOL} \)

\[ \forall x : \text{downtowndeck} \mid x.\text{colour} = \text{White} \]
\[ \forall x : \text{easttowndeck} \mid x.\text{colour} = \text{Black} \]
\[ \forall x : \text{freihilldeck} \mid x.\text{colour} = \text{Blue} \]
\[ \forall x : \text{merdistdeck} \mid x.\text{colour} = \text{Green} \]
\[ \forall x : \text{miskudeck} \mid x.\text{colour} = \text{Yellow} \]
\[ \forall x : \text{northsidedeck} \mid x.\text{colour} = \text{Orange} \]
\[ \forall x : \text{rivertowndeck} \mid x.\text{colour} = \text{Purple} \]
\[ \forall x : \text{southsidedeck} \mid x.\text{colour} = \text{Brown} \]
\[ \forall x : \text{uptowndeck} \mid x.\text{colour} = \text{Red} \]
### Outskirts

```latex
textname : STRING
monster : \text{\textlangle} MONSTERS \\
limit : \mathbb{Z}

\#monsters = \text{limit} \\
name = Outskirts
```
Appendix B

Contents of Attached Disc

The attached disc includes the following:

**Chat System** The entirety of the Chat System. This contains executable .jar files of the code (see README.txt before running) and the original source code files.

**Demonstration** The \LaTeX files for the first demonstration (21st March 2011), including the handout and the slides. Also included is the script for the presentation.

**Implementation** All of the code for the implementation, including the original source code. This contains

- **board** Locations, Streets, Neighbourhoods, and everything on the Board that Monsters and Players can move to.
- **cards** The many cards in the game: Arkham Encounter cards, Other World Encounter cards, Item cards, Spell cards, etc.
- **exceptions** The multiple exception classes that would be thrown if some kind of error occurred.
- **factories** These would read data in from files and generate Neighbourhoods and Investigators, amongst other things. Not fully completed.
- **freetypes** The various enumerations used with multiple classes.
- **investigator** The class representing the Investigator, as well as that representing SkillPairs.
- **monster** The classes that are combined to create a Monster.
- **toplevel** The classes that would actually play the game. Not fully completed.

**Poster Pitch** The finished poster and slides for the poster pitch (8th April 2011).

**Report** The original \LaTeX files for the report, including the Z style file, images, the modified source code included in the report, and a bat file for compiling select chapters or the whole document.
APPENDIX B. CONTENTS OF ATTACHED DISC

Sources Various digital sources listed in the bibliography. These are

Arkham Horror A voluminous discussion and review of the game by Robert Florence, which helped to inspire this project, and the fully revised version of the rules for Arkham Horror.

Formal Methods Ten Commandments of Formal Methods by Bowen and Hinchey (which might have benefitted the project from more use), and Seven Myths of Formal Methods, by Anthony Hall.

LaTeX J.M. Spivey’s guide to using the \texttt{L\LaTeX} Z style file, the complete description of the Listings package by Carsten Heinz and Brooks Moses, and a guide to using Beamer called \textit{Beamer by Example}, by Andrew Mertz and William Slough.

Z Notation The self-published pdf of J.M. Spivey’s reference manual, as well as the pdf version of Using Z by Woodcock and Davies, and a copy of Darren Slevin’s undergraduate thesis, which, while interesting, was only used for reference as to the layout of this project.

Software Engineering A text document containing E.W. Dijkstra’s The Humble Programmer (retrieved from the website for the Computer Science Department of the University of Texas), and a pdf copy of Larman and Basili’s paper on the history of iterative and incremental development.