PuppEzy: A Game Based Puppet Programming Teaching Tool

Jaka Mohorko

4th Year Project Report
Software Engineering
School of Informatics
University of Edinburgh

2019

Abstract

With a large amount of new system administrators entering the occupation, the need for tools to facilitate their learning has been on the rise. Additionally, many tools used in system administration are often held back by steep learning curves while lacking instructions on how they are to be used. One such tool is Puppet, a configuration management tool used to manage and configure systems.

In this paper I describe how I created a game which teaches its users the basics of the programming language used in Puppet. The game comprises six levels featuring different concepts of the programming language in ascending difficulty. I evaluate the game, testing its usability and suitability as a teaching tool for users with no previous experience in the use of Puppet.

Acknowledgements

I would like to thank my supervisor Dr. Kami Vaniea for her extensive support when I was struggling with finding the correct path for my project.

I would like to thank all the participants and contacts who took part in my studies - without you the creation and evaluation of my system would not be possible.

I would like to thank everyone who took their time to proofread this report and offer suggestions on how to improve it.

Thanks to all my friends and close ones for all their support.

Table of Contents

Chapter 1

Introduction

Configuration errors have been identified to be one of the leading causes of system failures and outages, with several high profile failures being attributed to them. Recently, in 2017, several US internet providers suffered an outage for 90 minutes due to a configuration error by Tier 1 network provider Level3 (Burt, 2017) [1]. In 2010, Facebook was down and unreachable for many for approximately 2.5 hours (Johnson, 2010) [2]. The worst outage Facebook had in over 4 years was caused by a configuration error. A misconfiguration brought down the entire .SE domain in Sweden in 2009 (CircleID, 2009) [3]. Such failures present substantial damage to companies responsible, damaging their image, reputability and incurring large financial and potentially data loss.

With several recorded cases of misconfigurations in systems, studies have been conducted which suggest that not only do configuration errors have high impact but are also prevalent. A study by Yin et al. reported that around 27% of technical support cases in a major US storage company are accounted for by configuration problems (Yin et al., 2014) [4]. In 2003, Oppenheimer et. al reported finding that more than 50% of all errors caused by human operators in Internet services were configuration errors (Oppenheimer et. al, 2003) [5]. As shown by the values, misconfigurations present a large and costly problem to companies. Troubleshooting them is time-consuming and expensive, as configuration settings today are growing increasingly complex. No matter the cause of a problem, be it bugs or misconfigurations, systems usually react in a similar way (eg. crashing, performance degradation), making it hard for system administrators to locate and fix the underlying problem (Yin et. al, 2011) [4].

As human error is such a dominant cause of downtime in computer systems, many configuration management tools such as Puppet, Chef, Ansible and others were developed to reduce the input needed by system administrators and developers when managing system infrastructures. However, these tools can often be daunting and hard to use for inexperienced system administrators. Some systems such as Puppet feature their own configuration language, which often evolve in an ad hoc fashion without much attention to their semantics (Fu et. al, 2017) [6], resulting with many features that are complex and hard to understand. While some of the configuration languages such as

the one used by Puppet may contain features similar to those used in object-oriented programming (inheritance, classes), even these often function and are used differently within the configuration languages.

With a large shift of the administrator population occurring (Xu et. al, 2016) [7], as the system administrator group greatly expanded to include non- and semi-professional administrators, there are many individuals entering the occupation with a lack of background knowledge and skills related to the use of administration tools. The barrier for entry for new system administrators is set as high as ever, with only few tools available to assist their learning process. With this project I intend to create a game-environment used to teach the fundamentals of the Puppet configuration language aimed at new and inexperienced system administrators with little or no programming knowledge.

1.1   Project Aim

With this project I aim to design and develop a system that teaches basic concepts of the Puppet configuration language by presenting a code sample to the user, explaining the contents of the sample and having the user configure a separate code block that meets requirements set in each level, using the knowledge gained. By introducing a new concept of the Puppet language in a puzzle in each game level, I intend to build up the users' understanding of the language from knowing very basic language features to understanding and being able to apply more advanced methods.

The system I am designing is intended to teach the Puppet language from the ground up, assuming no prior knowledge of programming. It does however require the user to have a fundamental understanding of concepts such as "file system" and "terminal command". The game is aimed at new system administrators who are looking to learn how to use Puppet to manage a system but are struggling to navigate through the documentation and online sources and are looking for a way of learning that offers topic focused hands-on experience in a controlled environment.

For my project, I have therefore set the following goals:

- Design code samples that showcase key Puppet language concepts such as *resources, attributes, classes and execution order* in ascending order in difficulty of concepts shown. For each of the samples develop a set of objectives prompting the user to design code blocks using the concepts introduced in the associated code samples.

- Develop a game where the code samples designed would be showcased and explained to the player alongside a system for the user to design their own code, and a solution validation mechanism to give the user feedback about the correctness of their input.

- Carry out an evaluation of the usability of the game developed and its suitability and efficiency as a learning tool.

1.2   Project Outcomes

With my project I achieved the following:

- Researched and compared prior work, incorporating or improving upon features present in related systems

- Developed a game featuring six levels aimed at teaching Puppet programming to new system administrators.

- Evaluated my system by conducting six think-aloud studies accompanied by pre- and post-game questionnaires

With all participants being able to complete my game within the think-aloud studies I conducted, I evaluated my system to be usable. By observing participants during their interaction with my game, I identified faulty aspects of my system to be improved in future iterations. In addition, through the use of post-game questionnaire results, I concluded that my system is successful in teaching its users the basics of the Puppet language.

1.3   Report Structure

The remaining chapters of the report are structured as follows:

Chapter 2: discusses the working and structure of Puppet and explains certain key aspects of its programming language.

Chapter 3: presents systems and studies related to my system, outlining the similarities and differences.

Chapter 4: describes how iterative design was used to develop the requirements and features of my system.

Chapter 5: describes how the system was implemented following the design laid out in previous stages and presents the final interface and level implementation.

Chapter 6: contains the evaluation of the system using Think-Alouds, focusing on usability.

Chapter 7: contains the concluding thoughts about the project and details further work that could be done.

Chapter 2

Puppet

Puppet is the name used as a shorthand for a collection of tools and services that assist the user in defining their desired system state in an infrastructure. It automates the process of creating and maintaining a system configuration (Puppetlabs, 2018) [8]. The specified system state is written in Puppet code, a domain specific language that allows you to define how each device in your infrastructure should be configured.

Within this section, I describe some of the aspects of Puppet relevant to the workings of my game system.

2.1 Puppet Architecture
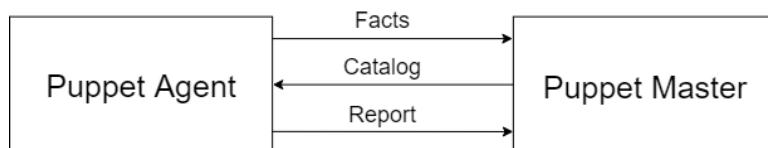
Figure 2.1: Overview of the Puppet architecture.

Puppet follows an *agent-master architecture*, where the *puppet master* holds information on what the desired configuration of a single machine (*node*) is. In the usual work-flow of Puppet, the node running a *puppet agent* periodically requests information (facts) about its configuration from the puppet master that runs on a server. The *puppet agent* collects information about the host machine it is running on and periodically sends it to the server (puppet master).

The master then compiles a *catalog* - a document that describes the system state for one specific computer, which lists all resources that need to be managed, as well as any dependencies between those resources [9]. The puppet master uses data received from the agent, as well as a centrally maintained configuration specification called manifest, to compile a catalog specific to a node. The catalog contains a set of *resources* computed for the target node, alongside other metadata (Fu et. al., 2016) [6].

Once the catalog is received by the agent, the puppet agent applies it to the node by comparing and adjusting the computer state to match the desired state specified in the catalog, sending a status report to the puppet master after finishing.

2.2 Puppet Code

Puppet code, the Puppet programming language, is composed of primarily resource declarations, which are, along with other program constructs, used to describe the system state, such as stating whether a certain user should or should not exist.

I now introduce the Puppet Code concepts that my game system teaches its users - *resources, variables, classes* and *ordering*.

2.2.1 Resources

Resources are a unit of Puppet code used to specify how some aspect of a system should be configured.
They are declared using resource declarations - expressions that describe the desired system state for a resource and tell Puppet to add the resource to the catalog.

A resource is associated with one of many different resource types featured in Puppet. Each resource type determines what kind of configuration concept the resource manages. In Figure 2.2 I present a resource declaration featuring the type `file`, one of many types within Puppet, such as `user`, `package` and `service`.

```
file { '/data/users.txt':
        ensure => present,
        owner => 'root',
        mode => '0600'
}
```

Figure 2.2: Resource declaration of type `file`

In Figure 2.2, a `file` resource with the title `'/data/users.txt'` is declared. Below the title, three attributes are specified, which describe the desired state of the resource. As in the example the path of the file is not specified, the title of the resource will be interpreted as the target of the resource being managed on the system. In order to prevent conflicting values declared for the same attribute, resource titles must be unique for each resource type, as having duplicates will result in a compilation error.

Puppet applies the resources specified in the order they are written in. However,

that setting can be overridden by specifying ordering relationships between resources, which will be explained in Section 2.2.4, or by randomising the application order.

2.2.2 Variables

Variables in Puppet code function differently than in most other languages, as they cannot be reassigned. Attempting to reassign a value to a variable within the same scope will result in a compilation error.

```
$owner = 'guest'

file { '/log.txt':
        owner => $owner,
        mode => $mode
}

$mode = '0666'
```

Figure 2.3: Example use of variables in a resource declaration

As shown in Figure 2.3, variable names are prefixed with a dollar sign ($) and assigned to with the equal sign (=) operator.

Although the Puppet language is considered to be mostly declarative, with the order of code not mattering, variable assignments are evaluation-order dependent. A variable can therefore not be resolved to a desired value before it has been assigned. When attempting to access a variable that has not had any value assigned to them, their value will be undefined, often resulting in unwanted behaviour. Looking at Figure 2.3, the file resource declaration value for owner would resolve to 'guest', however, as the $mode variable was assigned only after the resource declaration, it would resolve to undefined.

2.2.3 Classes

Puppet code can be grouped into classes - collections of resources that can be declared by invoking the class.
Classes can be invoked in two main ways by using *includes* or *resource-like definitions*, introducing two different behaviour modes for classes referred to as *include-like behaviour* and *resource-like behaviour*.

Include-like behaviour uses include functions to declare classes and allows us to declare a class several times safely, ensuring that only one copy of it will be added to

the catalog. This allows for classes to be included in several locations within a manifest without concern for duplicate declarations causing errors. Figure 2.4 shows an example of a `class` using include-like behaviour.

```
class data {
    file { '/data.txt':
        ensure => present ,
        mode => '0600'
    }
}

include data
```

Figure 2.4: Class declaration using include-like behaviour

Resource-like behaviour, unlike include-like, requires that a given class is invoked only once. As the name suggests, defining a class this way uses a similar syntax as a resource declaration. As shown in Figure 2.5, by using resource-like declarations we can override class parameters at compile time.

Resource-like declarations need to be unique to avoid conflicting parameter values. Since the class parameters may vary between class definitions, catalog compilation would prove to be unreliable and order dependent in the case of duplicate resource-like declarations.

```
class data ($mode = '0000'){
    file { '/data.txt':
        ensure => present ,
        mode => $mode
    }
}

class { 'data':
    $mode => '0600'
}
```

Figure 2.5: Class declaration using resource-like behaviour

2.2.4 Ordering

As briefly noted above, Puppet applies resources in the order written in manifests, or if specified, in a random order. However, by defining relationships between resources, the default order of application can be overridden.

Puppet uses four parameters to establish relationships between resources. They are specified within resource declarations as attributes of which values point to the target resource.

The relationship parameters in the Puppet language are, as stated in the puppet language reference (Puppetlabs, 2019) [10]:

- `before`: Applies a resource before the target resource

- `require`: Applies a resource after the target resource

- `notify`: Applies a resource before the target resource. The target resource refreshes[1] if the notifying resource changes.

- `subscribe`: Applies a resource after the target resource. The subscribing resource refreshes if the target resource changes.

```
exec { 'apt-get-update':
        cwd  =>  '/usr/bin',
        user  =>  'root',
        command  =>  'apt-get update'
}

package { 'apache2':
        ensure  => latest,
        require  =>  Exec['apt-get-update']
}
```

Figure 2.6: Ensuring application order using `require`

In Figure 2.6, the `require` parameter is used in order to ensure that the `apache2` package is installed after the `apt-get update` command is executed.

---

[1]Some resource types such as `services` must be 'refreshed' (restarted) when their dependencies change

2.3 Summary

In this chapter, I presented an overview of the agent-master architecture employed by Puppet and its work-flow of data gathering and node-specific catalog compilation, as well as a description of the Puppet language topics used in my system.

I introduced four Puppet code concepts by presenting samples and describing their behaviour and usage. The described concepts are:

- *Resources:* A unit of Puppet code used to specify how some aspect of a system should be configured.

- *Variables:* Puppet variables used to store values for later use. They resemble constants in other languages, as they cannot be reassigned.

- *Classes:* Collections of resources that can be declared by invoking the said class. Classes can be invoked by using includes or resource-like definitions.

- *Ordering:* Overriding of default application order within puppet by using functions to establish order-relationships between resources.

Chapter 3

Related Work and Similar Systems

In this section, I highlight a study exploring the use of *gamification* to teach system administration tasks, the incorporation of puzzles as a means to teach, and tools available that are used to facilitate the learning process of the Puppet programming language. I observe what features are similar and applicable to my system and how they are different.

3.1 Education Through Gamification

Gamification is the implementation of game design elements in real-world contexts for non-gaming purposes (Deterding et. al, 2011) [11].

In 2016, Begnum and Anderssen [12] conducted a study which incorporated system administration tasks into a course taught at BSc level using elements of gamification. The learning environment, *The Uptime Challenge*, tasked students taking the course *Database and application management* to manage a website given to them in a real cloud environment. The websites were subjected to constant usage and sabotage, while the student's performance was measured using site availability and response time.

Begnum and Anderssen (2016) [12] used a pre- and post-questionnaire to measure the effects of the challenge on student engagement and their understanding of the system administrator role. The results indicated that the challenge motivated students to spend more time on the course, while also presenting evidence that showed an increase in understanding of what the system administrator role means as opposed to what it does.

The Uptime Challenge motivated its participants by introducing game elements such as a reward system for website performance, a performance measure using *money* as a representation of points and a leaderboard of all participating students. While my project differs from the study, as it focuses on game-based learning instead of gamification, *The Uptime Challenge* highlights benefits of using game elements to teach

system administration concepts and how those elements can be used to augment the learning process.

3.2   Puzzle-based Learning

Puzzles have been well-known teaching tools in subjects such as mathematics for a long time (Parker, 1955) [13]. Recently, puzzles have made their way into engineering degrees and are being introduced into computer science courses with the aim of teaching critical thinking and problem-solving techniques (Parhami, 2009) [14] (Falkner et. al, 2010) [15] (Merrick, 2010) [16].

The results of studies of these courses suggest that the use of puzzle-based learning motivates students and increases course participation while developing their critical thinking and general problem-solving skills.

Z. Michalewicz (2008) and M. Michalewicz (2008) [17] state that educational puzzles should satisfy most of the following criteria:

1. *Generality:* Educational puzzles should explain some universal mathematical problem-solving principles

2. *Simplicity:* Educational puzzles should be easy to state and easy to remember

3. *Eureka factor:* Educational puzzles should frustrate the problem-solver. Puzzle results should be counter-intuitive, leading the problem-solver astray until they find a key to the solution. At that point the Eureka! moment is reached, accompanied by a sense of relief and reward.

4. *Entertainment factor:* Educational puzzles should be entertaining, or the puzzle-solver might lose interest in them.

The introduction of puzzles into computer science courses has been shown to have a positive impact on student motivation and their problem-solving skills. As my system aims to provide introductory lessons in Puppet within a game environment, incorporating puzzles as levels, with each puzzle introducing a new topic, provides a structure to build my game around while heightening player engagement.

In addition, presenting puzzles in a game environment allows me to facilitate the *eureka* and *entertainment factor* by adding visual rewards for level completion, as well as an interactive sandbox environment for the player to explore when searching for solutions.

3.3   Puppet Learning VM

The Puppet Learning Virtual Machine [18] is a sandbox environment developed to simulate the workings of Puppet within an enclosed system, having the Puppet Enterprise configuration management platform [19] pre-installed.

Figure 3.1: Screenshot of the Puppet Learning VM quest interface.

The VM uses a *Quest Guide* to teach users features of Puppet, with each *Quest* specifying a series of interactive tasks to be completed within the VM. Figure 3.1 shows the quest commands available within the VM.

The VM focuses on teaching its users how the Puppet infrastructure works and how to use it, while also providing instructions on how to write Puppet code [20]. Starting from teaching the user how to install and run *Agents*, it later also explains several aspects of the Puppet DSL (domain specific language) such as manifests, variables, classes and conditional statements. It presents an in-depth view of the Puppet architecture and language while providing a sandbox environment to explore.

While my system and the learning VM share the goal of providing an introductory lesson to system administrators attempting to learn the Puppet language, the VM focuses on giving the user freedom in exploring the system. It provides code and explanations to the user, with the ability to execute the code within the VM and observe the effects. The system developed in my project, in contrast, abstracts the Puppet tool environment and focuses only on the Puppet language. Both systems explore similar topics within the language, however, my system uses example code and explanations to assist the user in solving puzzles aimed at the topic of the code sample, utilizing aspects of puzzle-based learning.

In addition, my system aims to achieve ease of use for beginners unfamiliar with concepts of system administration. As the learning VM simulates the use of Puppet in a real environment, it assumes that its users are familiar with concepts such Linux terminal operations, how to use SSH to connect to the VM and the use of in-terminal editors such as 'vim'. I attempt to provide an intuitive system interface requiring no initial knowledge of those concepts, with all information required to proceed through the levels being presented to the users within my system.

3.4 Summary

Using game elements in a learning environment yielded positive results when used to teach system administration concepts, as described in a study by Begnum and Anderssen (2016) [12], by increasing student engagement and motivation. Similarly, my system uses game-based learning to keep users engaged as they are studying the Puppet language. Additionally, puzzles are used within my system to present knowledge to users from a problem-solving perspective, making use of Puzzle-based learning aspects such as the Eureka and entertainment factor.

With the most similar system used for teaching the Puppet language being the Puppet Learning VM [20], I aim to improve on the design by incorporating puzzle-based learning and game elements, as well as abstracting the Puppet tool environment and focusing on the Puppet language. In addition, I aim to make my system beginner friendly, assuming no previous knowledge of its users.

Chapter 4

Design

In this section, I detail the design process and highlight how my project changed from its inception as a game system aimed at testing the usability of the Puppet programming language to its final variant - a game system used for teaching basic concepts of the Puppet language.

4.1 Initial Project Design - Testing Tool

With the initial goal of the project being the testing of the usability of the Puppet language, I explored several of its advanced features and various game designs that would allow me to identify any potential shortcomings of the language and difficulties Puppet users might encounter.

4.1.1 Initial Concept Design

The issue I attempted to address with my initial project aimed at testing the usability of the Puppet language, is the fact that Puppet and other configuration languages contain several inconsistencies and hard to grasp concepts, sparking confusion among even the most experienced users. To identify the usability issues of the Puppet language, I attempted to provide an answer to the research question:

"How can we design a tool used for testing the usability of advanced Puppet language concepts?"

My proposed solution was to create a game system in which users are presented with puzzles featuring advanced Puppet language concepts. The usability of each concept would be evaluated using data such as time spent and number of errors made in tasks focused on specific Puppet features.

As my knowledge of Puppet at the start of the project was lacking, I consulted two contacts with advanced knowledge of the language, the Puppet reference manual which

contains details of complex Puppet behaviour, as well as online forums to identify ambiguous and confusing components of the Puppet programming language which would be the focus of my research. Using the data gathered, I identified the concepts to be tested as:

- Scope

- Inheritance

- Containment

With the Puppet concepts chosen, briefly described in Section 4.1.2, I aimed to design a system which would test how well experienced Puppet users perform when making use of the above concepts. My design would present uniquely solvable puzzles, each aimed at testing a particular Puppet language concept to users. This would allow me to gauge the usability of said concepts by measuring the users' performance when solving the puzzles.

From here, an initial set of high-level goals for my system was devised:

- create a set of puzzles, each featuring a specific advanced Puppet concept to be tested

- design the puzzles to be uniquely solvable

- create a game environment that presents the puzzles to the users while collecting solving time and input error data

4.1.2   Puppet Language Concept Selection

As was discussed in Section 4.1.1, the Puppet language concepts chosen to be tested were *Scoping*, *Inheritance* and *Containment*.

4.1.2.1   Scoping

Scoping was chosen as a topic of interest, as, although it functions in a similar way as in most other programming languages, scoping in Puppet sometimes behaves differently, thus often being the cause of errors when users are unaware of the differences.

Scopes in Puppet impose a limit on where variables and Resource defaults - default attribute values for given resource types - can be accessed from. However, scopes do not affect the reach of resource titles, which are global, and references to resources, which can refer to a resource declared in any scope (Puppetlabs, 2019) [10].

As in most other programming languages, scoping in Puppet employs a Parent-Child infrastructure, where children inherit the scope of their parent. However, in the case of variables, by specifying the *global name* of a scope, out-of-scope variables can be accessed.

As is apparent, the Puppet language implements scoping with features diverging from usual implementations, thus creating opportunities for errors when users are not aware of the differences.

4.1.2.2 Inheritance

Inheritance within the Puppet language allows classes to be derived from other classes. The inheriting (child) class inherits all variables of its parent and is given permission to override any resource attributes of the base class. In addition, inheritance in Puppet does not allow parameters to be provided for inherited classes.

When declaring a derived class, the base class is automatically declared first, unless it has already been declared elsewhere. In that case, the main class is declared within the current scope, with its parents set accordingly, effectively inserting the base class between the derived class and the current scope (Fu et. al, 2017) [6].
As such, inheritance may lead to unintended consequences due to the scope of a declared class being dependant on compilation order.

It is currently advised to use inheritance in Puppet very sparingly, with the main uses of it being:

- Overriding resource attributes in the base class

- Obtaining default values for a class's parameters from a *parameter class* containing class parameters and their values

With inheritance often being the source of errors and unintended behaviour, it was chosen as one of the topics to be evaluated within my system.

4.1.2.3 Containment

Within the Puppet language, classes *contain* resources that are declared within them, ensuring the resources are not applied before the application of the class begins and that the application of said resources finishes before the application of the class.

While resources are automatically contained, classes behave differently, thus diverging from the usual containment behaviour of most programming languages.
If a class is declared within another class, it is not automatically contained unless explicitly specified using the `contain` function. This can result in unintuitive behaviour, as classes declared within another class might be applied before or after the class in which they are declared.

With containment being a commonly used feature in common programming languages such as Java, the inconsistency within the Puppet language can lead to user error. Thus, it was selected as a concept to be tested within my system.

4.1.3   UI Design Iterations

Having conducted research into the Puppet language concepts to be tested, I started designing my interface by drawing sketches and creating mock-ups of my system interfaces. I evaluated and obtained feedback on my designs by presenting the sketches to Computer Science students and a PhD student with experience in system administration experienced in Puppet. In addition, the designs were presented to my supervisor Dr. Kami Vaniea, an expert in Human-Computer Interaction, at weekly intervals.

Using my goals described in Section 4.1.1, I started iterating on possible designs while obtaining feedback on each version.

4.1.3.1   Design 1 - Pre-set Commands

The first design tasked the users to configure several groups of PC's to adhere to specifications set by the objectives of each puzzle. The game screen would include a list of computer groups of which system states could be inspected and adjusted by applying pre-set commands representing blocks of Puppet code. In Figures 4.1 and 4.2 we can see two design variations showcasing different layouts and representations of the system.

Although the design seemed intuitive to contacts assisting me with its evaluation, it came with a design problem of not accurately representing the creation process of Puppet code as the users would have to select pre-set input commands - abstractions of Puppet code - instead of actual lines of the Puppet language. In addition, having users fully configure groups of computers would require the puzzles to make use of several Puppet concepts, making the isolation of each concept difficult.

4.1.3.2   Design 2 - Identification and Revision of Incorrect Code

The second design featured pre-written Puppet code containing errors which would have to be identified and marked by users through the use of checkboxes located next to lines of code and revised using a selection of different options available in drop-down menus. Figure 4.3 shows a mock-up of the design made during the design process. The feedback on this design was positive and all test users described it as intuitive. It improved on Design 1 by allowing the use of actual Puppet code while also focusing on specific aspects of the language.

However, presenting potential examples of malformed code featuring the concepts I selected highlighted an issue with both the design of my system and the use of it for testing Puppet concepts. The design of code samples containing errors, which can be mended within the boundaries imposed by this system design presented a tall obstacle. Presenting samples to a contact experienced in Puppet, highlighted the issue of solutions either being too obvious to an experienced user, or too difficult to identify even by advanced Puppet users.
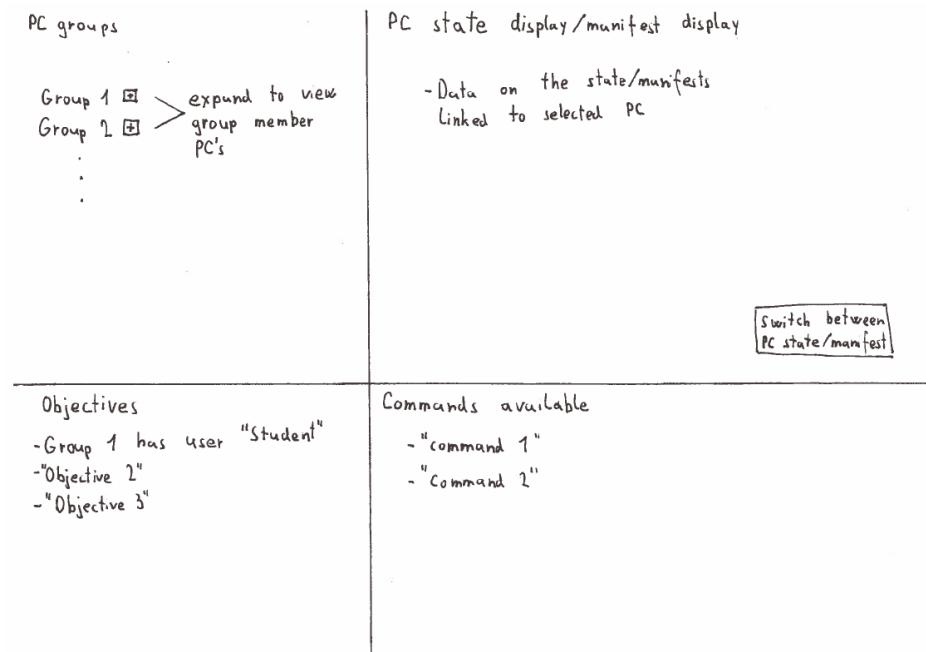
Figure 4.1: Sketch of the first Design 1 variant. The user must configure each PC group to match the system state described in the objectives by using commands available to them.
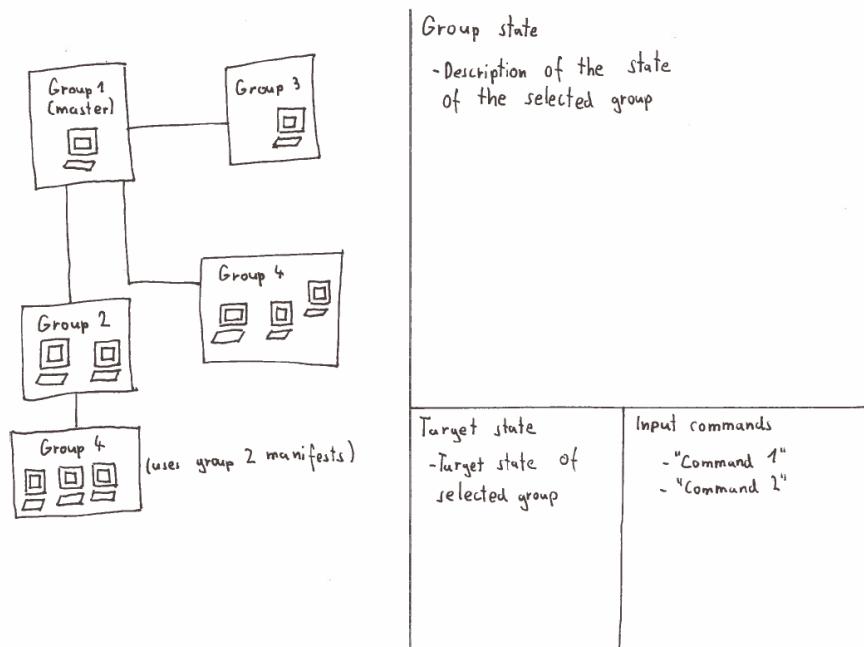

Figure 4.2: Sketch of the first Design 1 variant. The user must configure each PC group to match the target state of a group shown using pre-set commands. The groups within this variant have dependence between each other, where groups inherit configuration files of their parent groups.

```
        /bicycle/params:                              /bicycle/init:

1   Class bicycle::params{              1   ☐    $tyre_size     =    undef
2      $tyre_size     =   "15"          2   ☐    $bicycle_type =    undef
3      $bicycle_type =   "mountain_bike" 3   ☐    $frame_size   =    undef
4      $frame_size   =   "24"           4   ☐    $suspension   =    undef
5      $suspension   =   "present"      5   ☐
6   }                                   6   ☐    Class bicycle::init{
                                        7   ☐       include bicycle::params
                                        8   ☒       ensure      =>   [        ∨]
                                        9   ☒       frame_size  =>   [        ∨]
                                        10  ☒       tyre_size   =>   [        ∨]
                                        11  ☒       suspension  =>   ┌──────────────┐
                                        12  ☐    }                  │$bicycle::params::sus│
                                                                    │pension              │
                                                                    │                     │
                                                                    │$params::suspension  │
        Manifests:                          Objectives:             │$params->suspension  │
                                                                    └──────────────┘
        - bicycle                           We want the bicycles we are configuring to be
             |- params                      of the type specified in /bicycle/params, but
             |- init                        are encountering errors. Mark which lines of
                                            code are causing the errors and correct them
```

Figure 4.3: Mock-up of the second design variant using an abstraction of Puppet resource attributes as bicycle features. Incorrect lines of code were marked, and the user is given input options on how to fix the code.

4.1.4   Discussion of the Testing Tool Project Direction

The Puppet language research and UI design process highlighted several flaws in the premise of my project. Here I explain what some of the issues encountered were and how my new project direction of building a teaching tool was chosen.

4.1.4.1   Project Design Issues

Puzzle Design: As was briefly highlighted in Section 4.1.3.2, the design of code samples containing errors focusing on Puppet language concepts chosen for evaluation, while maintaining a reasonable difficulty level for advanced Puppet users proved to be harder than predicted. Creating code samples allowing for only one solution which could be evaluated within my system, while also testing only the exact concepts highlighted in Section 4.1.2 was identified to be beyond the scope of my project and Puppet expertise.

As, at the beginning of the project, I had no experience in the use of Puppet, the design issues and obstacles in puzzle design only became apparent after having studied Puppet for an extensive period of time.

Research Participant Gathering: The discussion of potential puzzle designs with a PhD student experienced in Puppet highlighted the issue of gathering enough participants for my research. With Puppet being a tool used for the specific purpose of administrating several systems, it is mostly only known and used by a fraction of system

administrators. Thus, finding participants with advanced knowledge of the language posed a significant obstacle.

In addition, as there are multiple methods of achieving the same configuration of a system within the Puppet language, depending on the coding style an advanced Puppet user employs, the resulting solutions might vary between participants resulting in my system not being able to evaluate the input. While this can be alleviated by introducing constraints on user input, doing so would obstruct users when trying to make use of their pre-existing knowledge of Puppet, thus affecting results.

Focus on Specific Puppet Concepts: With the system design aimed at testing specific Puppet concepts, the Puzzles needed to be designed accordingly. As Puppet code often requires the use of a combination of various elements of the language, it proved to be very difficult, or almost impossible to create puzzles where the distinction between concepts used could be clearly drawn.

Therefore, puzzles created would require the users to have knowledge of concepts of the Puppet language which are not being tested, potentially influencing their performance and perception of the code presented to them. Additionally, when evaluating the results, the causes of error and confusion would be hard to attribute to a specific concept due to their overlap in puzzles.

4.1.4.2 New Project Aim

As highlighted above, the aim of my project to design a game system used for testing the usability of advanced Puppet concepts proved to be beyond the scope of this project, as several of the obstacles present were insurmountable. Therefore, I decided to re-purpose my project towards creating a teaching game system aimed at beginner Puppet users. The new project aim alleviated the issues present in the testing tool design, while still making use of most design choices and research done.

While the teaching tool would make use of puzzles, these would be used to teach users instead of testing them. With their design focused on teaching basic to intermediate Puppet concepts instead of highly advanced ones, the puzzle design process was simplified. In contrast to the previous project direction where the inclusion of concepts other than the ones focused on in each puzzle presented a design obstacle, within a teaching environment, it instead allows for reinforcement learning where the users can rehearse previously gained knowledge while also learning about new topics.
With the number of experienced Puppet users being sparse, a tool aimed at beginner users offers a much wider pool of potential users available for evaluation and testing, alleviating the issue of gathering enough test subjects.

In addition, as I had no experience with the Puppet language when starting the project, I had to learn the language from scratch, going through the steps a beginner user would,

were they to attempt to learn how to write Puppet code. Using that experience allows me to gauge the shortcomings of Puppet learning tools available and improve on them within my system.

4.2 Final Project Design - Teaching Tool

4.2.1 Initial Concept Design

The issue I attempt to address is the fact that there are very few resources and entry-level learning tools available for new system administrators. With many new administrators lacking background skills entering the occupation (Xu et. al, 2016) [6], the absence of learning resources with a low barrier of entry becomes ever so apparent. Therefore, the research question I aim to answer is:

"How can we design a learning tool for teaching the Puppet language to new system administrators with lacking background knowledge?"

My proposed solution is to create a system which allows its users to create Puppet *manifests* and configure resources within them to meet objectives of each level by drawing upon information and samples presented within the system.

As the initial goal of my project was to *"test the usability of advanced Puppet concepts within a game environment"* and since I had no background knowledge of configuration languages, I experienced the learning process of a beginner Puppet user at first hand. Having to resort to hard to use tools such as the Puppet learning VM described in Section 3.3 and unstructured tutorials, revealed the need for a beginner level learning tool which assumes no knowledge of its users and teaches them the basic concepts they need, to be able to understand other available resources.

As such, I decided to re-purpose my project and create a Puppet language learning tool. In my system, I aim to create game levels where basic concepts of the Puppet language are introduced to the users. Using examples and explanations as support, the system aims to teach simpler concepts in earlier levels and later transition into teaching more complex features of Puppet. In addition, I aim to present the language in a self-contained game environment accessible to beginner system administrators, requiring no additional setup or tools. Finally, I aim to, through engaging game-play and level design showcasing possible uses of Puppet, generate interest in the usage of Puppet and incite users to further explore configuration languages and topics related to system administration.

From here, an initial set of high-level goals for my system can be devised:

- create a set of levels, each introducing a separate Puppet language concept

- have a low barrier to entry with all knowledge required to play contained within the system

- generate interest to further explore Puppet or other configuration languages

4.2.2 Requirements

Using the results of my research conducted on the Puppet language, related work and the design process and evaluation of my system as a testing tool, as well as the high-level goals devised for the teaching tool, a set of requirements for my system was created:

Requirement 1: The system will present a set of levels containing puzzles.

As mentioned in Section 3.2, incorporating puzzles into the learning process affects the students' motivation and stimulates critical thinking and problem-solving skills. My system will make use of puzzles to keep its users engaged while applying knowledge gained to solve Puppet code puzzles.

Requirement 2: Levels in the system progressively introduce new Puppet concepts in order of difficulty while still using concepts taught in previous levels.

Using elements of scaffolded assignments by designing a sequence of levels which gradually increase in complexity allows for a graduated structure where students can master easier tasks before moving on to more advanced ones (Schroeder, 2012) [21]. My system will make use of scaffolded assignments by breaking up the Puppet language into concepts and introducing them in separate levels, starting with basic and easy concepts and progressing to more advanced ones in later levels. In addition, making use of concepts learned in earlier levels allows the users to reinforce the knowledge gained through repeated use of said knowledge.

Requirement 3: The system will have a low barrier to entry.

As my system is aimed at new system administrators with lacking background knowledge, it must be accessible to beginners and easy to use. Therefore, the system must contain full explanations of all concepts and must make little to no assumptions of previous knowledge.

Requirement 4: The system will reward the user upon level completion

In order to facilitate the *eureka factor* described in Section 3.2, the user will be presented with a reward screen at the end of each level, indicating they've successfully overcome the challenges within a level.

Requirement 5: The system will give feedback to the user based on their input

In order to assist users when they cannot progress and cannot identify a mistake in their proposed solutions, the system will provide them with hints based on the mistakes they make. Doing so allows the users to seek help from the system whenever they encounter obstacles to ensure that they do not get stuck without being able to progress.

4.2.3 Final Design

Using the design described in Section 4.1.3.2 as reference, I developed a design using most aspects of the previous variant. As the new project aim requires users to create Puppet code based on the objectives of each level, the pre-existing code section with checkboxes was removed and replaced with a user code input area where they will able to craft their own Puppet code.

As the users will be inputting all their code within a single screen representing a manifest, the directory tree of manifests was no longer necessary and was replaced with function buttons allowing access to several level components such as objectives and a knowledge base of explanations of Puppet concepts. The submit button will allow users to check whether their solution is correct or not and offer hints on what is wrong.

While the malformed code selection mechanic using checkboxes was removed in this design, the code selection through a limited set of options in a dropdown menu was kept. In order to ease the implementation of the input evaluation method and restrict the user to choices using concepts taught by the system, the system will provide a set of code options to users to choose from in each level. Limiting the input options ensures that there is a unique solution to each level/puzzle making use of all Puppet concepts taught in that level, allowing users to practice and develop their understanding of said concepts.
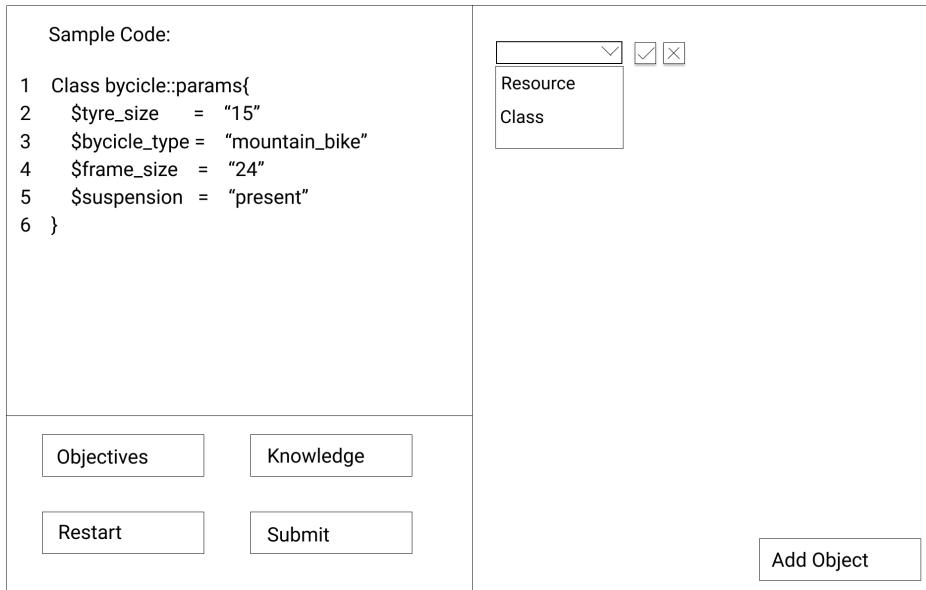
Figure 4.4: Mock-up of the final design variant using an abstraction of Puppet resource attributes as bicycle features in the sample code.

4.3 Summary

In this chapter, I described how the direction of my project changed from the development of a Puppet testing tool to a Puppet teaching system for beginner users.

Using iterative design methods, several designs were developed and evaluated to meet goals set for the initial project direction. Through the analysis of my designs, I decided to switch the aim of my project, creating a new set of high-level objectives. With the use of my research done on the Puppet language, related work and the design process, as well as the high-level objectives for my system, I defined set of requirements:

1. The system will present a set of levels containing puzzles.

2. Levels in the system progressively introduce new Puppet concepts in order of difficulty.

3. The system will have a low barrier to entry

4. The system will reward the user upon level completion

5. The system will give feedback to the user based on their input

Using the requirements and previous designs as guidelines, I created a final design of my system as a teaching tool.

Chapter 5

Implementation

5.1 Back-end

With my system being developed on the game development platform Unity [22], the back-end was implemented using C# in conjunction with the Unity scripting API. The scripts written for the system were used to control all aspects of the UI - displaying text, button functions, UI element creation and modification, user input, as well as solution analysis and hint/feedback generation. In total 3625 lines of code were written for the project.

5.1.1 Development Platform Choice

As briefly mentioned, Unity was chosen as the game development platform on which the system was built. As I had little experience in game development at the start of the project, Unity was chosen due to its widespread use and extensive documentation. In addition, its scripting API offers broad customization support for any game elements implemented, allowing for the implementation of most design choices made. Unity allows the building of games for both Windows and as WebGL applications, allowing games to be run in HTML5, thus easing distribution.

However, having built the system within Unity revealed certain flaws of my choice of development environment. Although Unity offers extensive support for graphical game design, my system does not make use of it, with its core game-play mechanics making use of buttons which add or remove UI elements in specific locations. Relative UI element locations in Unity are often not intuitive and require several tricks and additional nesting of elements (in some cases of my system up to 12 layers of nesting) to achieve basic features such as indentation of objects. As such, were I to build the project anew, more research time would be dedicated to the selection of a framework with greater UI creation support.

5.1.2   User Input Analysis

In order to analyze the user input submitted in each level, a solution analysis script was created. Having stored all level solutions within lists containing resource, class and variable names alongside all their attributes, the user submitted solutions are compared to the list entries and evaluated, determining whether the correct solution was submitted. This is done by matching the stored solutions to the user submitted one and verifying if all required entries are present.

To emulate the declarative nature of Puppet, when evaluating the users input, all solutions containing the required entries are allowed, regardless of their class and resource declaration order.

5.1.3   Feedback Generation

When a solution is submitted by the user, a unique exit code is generated based on the last input mistake detected. The code is then used to generate feedback in the form of hints given to the user.

In order to ensure that hints are given in a sensible order, the evaluation order of the input was adjusted to first provide feedback on the outer-most scope entries. The feedback order was implemented as follows:

1. Class declarations

2. Resource declarations

3. Variable declarations

4. Class invocations (includes, resource-like declarations)

Only when no error is present in objects with higher priority, errors for lower priority objects will be displayed.

5.2   Front-end

Being familiar with the tool, Adobe Photoshop was used to create images used for UI elements such as the background and buttons. Unity placeholder objects were used during development, with their images later being replaced with appropriate assets. Please refer to Appendix A for an overview of UI element images created.

All UI elements were created as Unity game objects, using boundary constraints and coordinates to position them within the game. Each UI element was controlled by associated back-end scripts defining their behaviour such as actions performed on button presses.
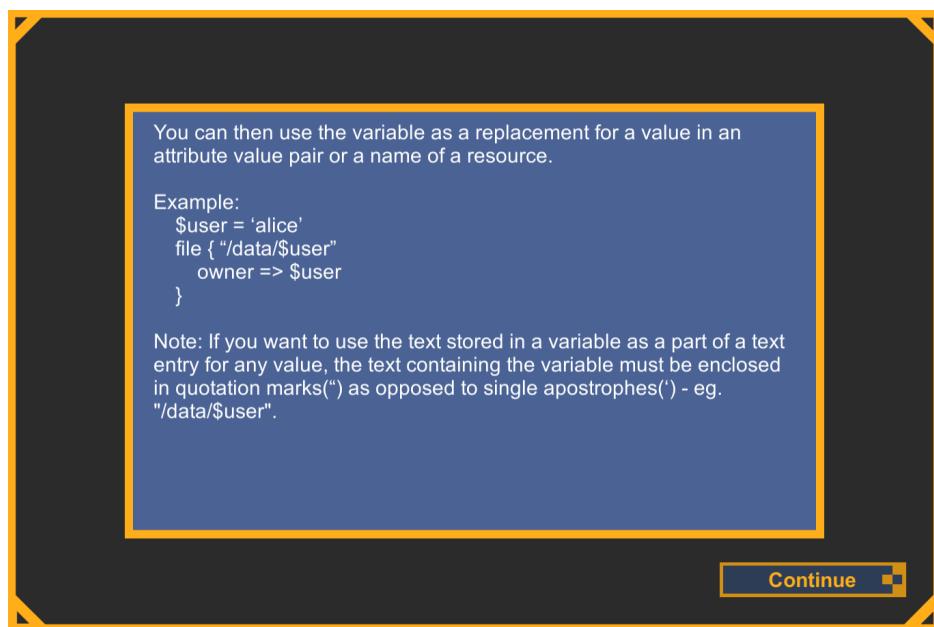
Figure 5.1: Screenshot of a level introduction screen of Level 3, explaining the use of variables.

5.3 Level Content

5.3.1 Level Introduction

In order to introduce new knowledge to the user, each level includes an introduction section where explanations of Puppet concepts taught in the level are present (Figure 5.1). The narrative of the texts refers to users as *Puppet apprentices* and *students of system administration* in order build a connection with the knowledge gained and the area of usage of Puppet, while also establishing the character of the user as a system administrator.

All Puppet language concepts explained to the users are presented alongside code samples featuring the use of said concepts. The introductory texts present in the level introductions are brief, as the game aims to teach its users through examples. In addition, due to the brief introductions, users are given access to additional explanations within the knowledge base described in Section 5.3.3, as well as sample code located within each level.

5.3.2 UI Tutorial

As an introduction to the user interface, my system features a tutorial describing each section of the main gameplay interface by highlighting the UI aspects being described at each part of the tutorial (see Figure 5.2). The users must proceed through the tutorial before starting the first level in order to ensure that they are familiar with the game
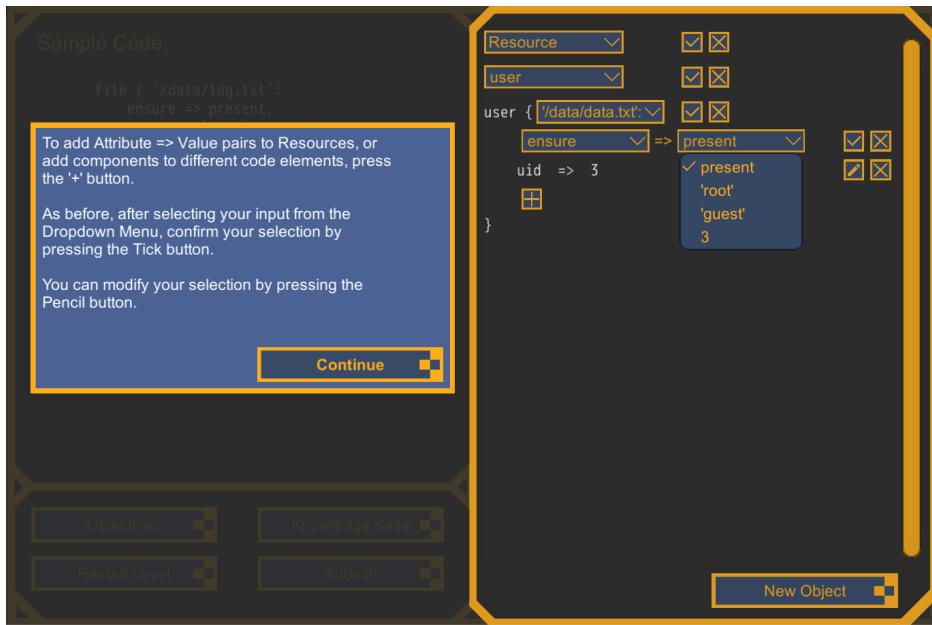
Figure 5.2: Screenshot of a part of the UI tutorial presented to the user in level 1.

interface and the functions of all buttons.

5.3.3 Main Gameplay

After navigating through the level introductions and tutorials, the users are presented with the main game screen where they must write Puppet code using the interface provided and submit a solution that enforces a system state specified in the level objectives.

As seen in Figure 5.3, the main game screen is composed of a section featuring level-specific sample code, the code input section and four buttons giving access to the knowledge base, objectives, solution submission and a restart level button allowing users to replay the level introduction.

Additional screenshots of the main gameplay elements described below can be found in Appendix B.

5.3.3.1 Objectives

After navigating through each level introduction, users are presented with the objectives of the current level. The objectives are presented as paragraphs in text form. Users can access the objectives by default at the start of each level and by pressing the "Objectives" button.

Each objective describes a certain aspect of how the system managed within the game must be configured, hinting to the users which resources and attributes to use when crafting their solution.
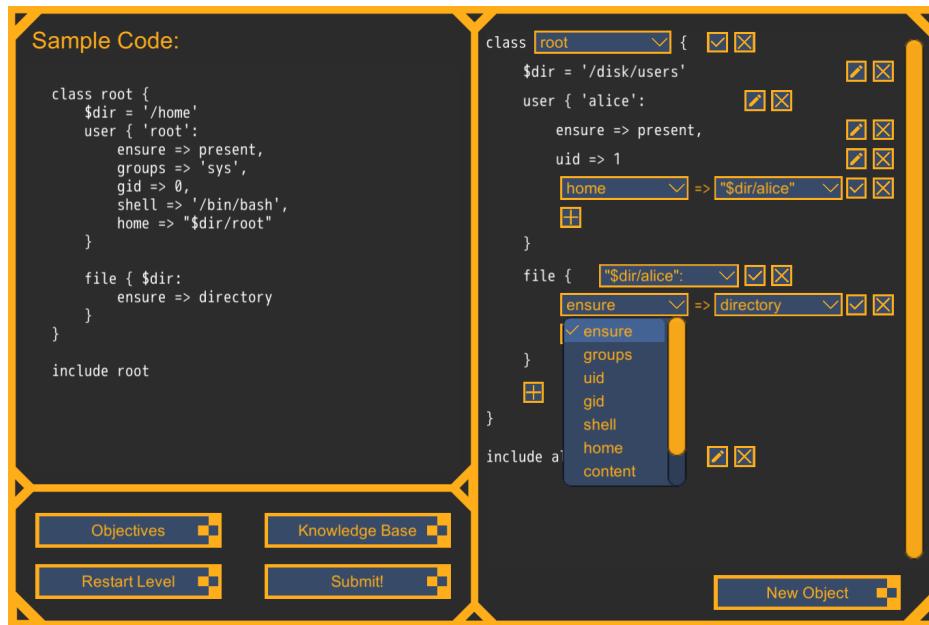
Figure 5.3: Screenshot of the main gameplay screen of level 3. The right side of the screenshot contains the code featuring the 'New Object' button, dropdown menus with input options, confirm ('tick'), modify ('pencil') buttons and delete ('x'), add ('+') buttons. Top right contains the sample code. Bottom left contains navigation buttons.

5.3.3.2 Sample Code

Within the gameplay screen of a level, as can be seen in Figure 5.3, sample code is given to the user showcasing Puppet concepts explained in the level introduction. The user is able to consult the sample code for hints and use it as a reference to assist them in composing their solution.

The sample code in early levels features examples very similar to the solution, where the user is only required to change a few resources and their attributes to complete a level. However, as the user progresses through the game, while code samples still feature concepts taught in a level, their content no longer contains all information necessary to craft a solution. The users must use knowledge gained through all sources available within the game to craft a complete solution to each level.

5.3.3.3 Knowledge Base

All Puppet language concepts explained in level introductions, with extra information on each topic, can be found within the knowledge base of each level. Figure 5.4 shows a screenshot of the knowledge base in level 6. The knowledge base contains explanations of all topics users have learned in the current and previous levels. It can be accessed by pressing the "Knowledge Base" button.

With the level introductions only offering a brief overview of each Puppet language
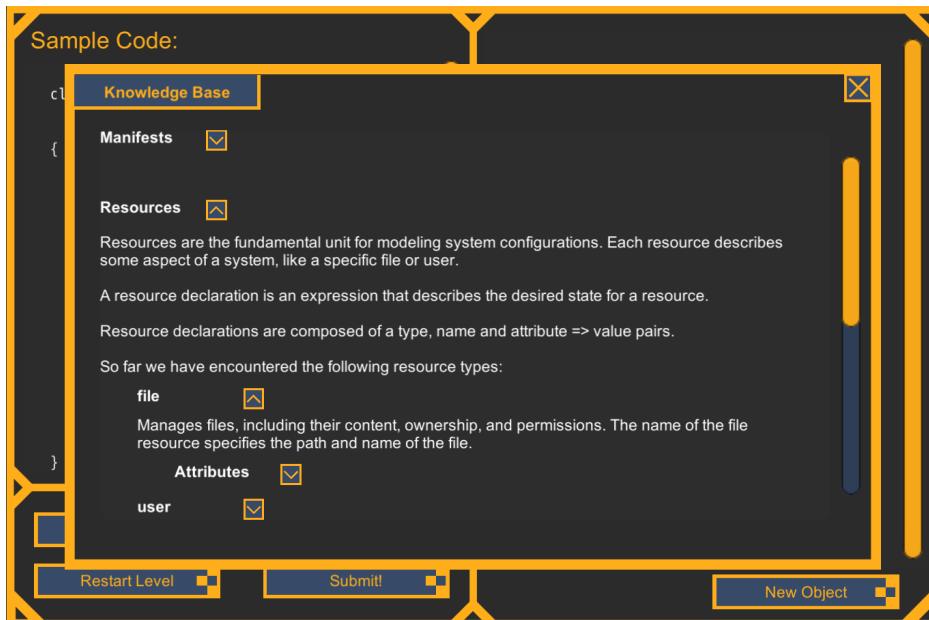
Figure 5.4: Screenshot of the knowledge base of level 6 with the Resources entry expanded.

concept, the knowledge base acts as a companion to the level, holding additional explanations and examples to aid the user when they encounter obstacles in the game.

5.3.3.4   Code Input

As can be seen in Figure 5.3, the user constructs their code in the right section of the UI. By pressing the "New Object" button, the user can create a new code block, choosing among elements such as Class or Resource, with the number of options increasing as they progress through levels and gain new knowledge.

After creating a code block, users can add components inside the block by using the "Plus" button. Any input can be removed by pressing the "Cross" button associated with it.

Whenever the user adds a new element, they are presented with a choice of input via dropdown menus. Each level features its own selection of possible inputs displaying a set of options required to complete a level, as well as additional options which serve as a medium to obfuscate the correct solution and increase the puzzle difficulty. Once the user selects the input option they want, they can confirm their selection by pressing the "Tick" button, and later modify it with the "Pencil" button.

When implementing the buttons associated with each code line or block in the user code input section of the screen, the buttons were placed at a constant position along the horizontal axis of the game for each level of code indentation (three levels of indentations can be found in the system code input screen), matching the indentation level of the code line the buttons affect. This was done to provide users with an intuitive design where the outcome of each button press is predictable.

The user can submit their solution by pressing the "Submit" button. If they submit a correct solution, they are rewarded with a console display showcasing all changes that were made on the system when their code is applied. A screenshot of the 'victory screen' can be found in Appendix B.

5.3.3.5 Hints

If the user submits an incorrect solution, they are given an option to view a hint which directs them towards the code section where the error was found. The hints do not give direct solutions, but instead inform the users about missing or incorrect elements and their location, such as within which class or resource the mistake was made. After reviewing the hint, users are able to continue solving their current level from the point they were at before submitting their solution.

Some examples of possible hints displayed to users are:

- "A variable declaration is missing"

- "There is a missing resource declaration, or a resource is incorrectly named"

- "The content of a class parameter is incorrect."

- "Class *'class name'* has too many variable definitions."

- "Resource *'resource name'* in class *'class name'* has too many attributes."

5.4 Levels

Within my system, six levels were implemented. Each level focuses on introducing a new Puppet language concept while making use of knowledge the user learned in preceding levels. The levels aim to teach the concepts explained in Section 2.2. As such the topics I aim to cover are:

- Resources of type `file`, `user`, `group`, `service`, `package` and `exec`

- Variables

- Classes with include-like and resource-like behaviour

- Ordering using the `before` function

Using these concepts as guidelines, I created six levels.

Level 1 - Manifests, resources and resource types `file` and `user`

With resources being the main building blocks of manifests, the first level introduces the user to their structure and use. The sample code and level introduction provide examples of `file` and `user` resource types, enabling the users to complete the objective of creating a `user` type resource declaration with two attributes.

This level serves as a medium for introducing the user to the system interface, while also explaining fundamental topics of the Puppet language.

Learning outcomes:

- user is familiar with the system interface and is comfortable when using it

- user recognizes and understands what "manifests" and "resources" are

- user understands the structure of a resource declaration

- user understands the usage of `file` and `user` resource types and some of the attributes that can be used with these types

Level 2 - Resource type `group` and several resource attributes

The second level is aimed at consolidating the fundamental knowledge explained in level 1, as well as introducing a new resource type and additional resource attributes.

The user is tasked with designing a manifest to configure a system state with no user 'guest', a user 'alice' and a 'log' file present in the system, with all components having several configuration attributes specified by the level objectives.

As this level aims to reinforce the knowledge gained in the first level, I intentionally introduce only a few new Puppet language concepts. After completing level two, the user should be able to navigate the user interface with ease and have a firm grasp on the concept of 'resources'.

Learning outcomes:

- user is able to swiftly and with ease navigate the system UI

- user has a firm grasp on the concept of 'resources' and their components

- user understands the usage of the `group` resource type

Level 3 - Classes using include-like behaviour and Variables

This level introduces code elements different from 'resource declarations'. The user is now able to incorporate classes and variables as part of their code. After an explanation of how classes are used and invoked, and where variables are declared, the user is tasked with specifying a system state using the concepts they have been taught.

They are to ensure that a user 'alice' and a directory at her home folder titled 'alice' are created when the class is invoked while making use of a variable to specify the home directory of 'alice'.

Learning outcomes:

- user understands the concept of a 'class' and can create their own within the system interface

- user understands and can make use of the resource-like behaviour of classes by using `include` functions

- user understands what 'variables' are, how they are used within a manifest, where they can be declared and what restrictions they have within the Puppet language

Level 4 - Resource types `exec`, `service` and `package`

Within the fourth level, the resource types `exec`, `service` and `package` are introduced to the user. The `exec` type is used to execute external command on a computer, whilst the `service` and `package` resource types are used to manage packages and services such as Apache or OpenSSH.

The user is tasked with creating a configuration file which installs and runs the Apache HTTP Server on a system after being given an explanation of new concepts and sample code which uses a similar code base to install OpenSSH.

This level is meant to provide the users with additional insight into different use cases for Puppet, aside from managing users and files. Through creating a working configuration file for configuring and running Apache and OpenSSH on a server, the user is given insight into some intended uses of the Puppet configuration platform. Exposing the user to configuration files managing commonly used packages, the level aims to showcase cases where the users could use Puppet within their own systems, encouraging them to further explore the Puppet framework.

Learning outcomes:

- user understands and is able to make use of the `exec`, `service` and `package` resource types

Level 5 - Parametrized classes using resource-like behaviour

As the classes using *include-like behaviour* cannot have their arguments modified on invocation, we introduce the users to *resource-like behaviour* (see Section 2.2.3). Using resource-like declarations coupled with parametrized classes, the users are taught a method of creating more generalized classes of which parameters can be adjusted when invoking a class.

Within the fifth level, the users are tasked with creating a class which deletes a user and their home directory from the system, with the username and directory path specified by the class parameters.

Learning outcomes:

- user understands the concept of resource-like behaviour and the use of said behaviour with parametrized classes

Level 6 - Ordering using the `before` **function**

Finally, the users are taught how to use *execution-order control* (see Section 2.2.4) by using the `require` keyword. I focus on the use of only one of the order control functions, as the level aims at teaching the high-level concept of ordering within the Puppet language, to not overload the users with new knowledge.

Users are now tasked with creating two classes, one using *include-like behaviour* and the other *resource-like behaviour*, which are used to install and run the Nginx and OpenSSH packages.

Learning outcomes:

- user understands the concept of execution-order control and is able to use it using the `require` function

5.5 Summary

In this chapter I described how the final design described in Section 4.2.3 was implemented.

I highlight my choice of Unity as a development environment due to its widespread use and documentation and C# scripting support for the back-end. In addition, I describe how user input analysis and feedback was implemented.

An overview of the front-end design is presented, introducing all components of the UI. All features found within the game are described in detail and accompanied by figures showing their implementation.

At the end of the chapter, I provided an overview of the contents of each level, highlighting their intended learning outcomes. The levels are implemented with the following topics:

1. Manifests, resources and resource types `file` and `user`
2. Resource type `group` and several resource attributes
3. Classes using include-like behaviour and Variables
4. Resource types `exec`, `service` and `package`
5. Parametrized classes using resource-like behaviour
6. Ordering using the `before` function

Chapter 6

Evaluation and Discussion

In this chapter, I discuss the evaluation of my system and its results. I focused my evaluation on the usability of the system, making sure that its UI is intuitive and usable by beginners, while also evaluating whether my system is successful at teaching its users. Therefore, I conducted a think-aloud study testing for usability issues, accompanied with pre- and post-questionnaires which evaluated the pre-existing knowledge of users, user satisfaction and learning outcomes.

6.1 Evaluation

6.1.1 Think-Aloud

Think-alouds are one of the most common evaluative methods, asking participants to articulate what they are thinking, doing, or feeling as they complete a set of tasks. It grants insight into the task completion process, while also allowing for identification of system aspects that delight, confuse, and frustrate, which can then be improved or corrected (Martin et. al, 2012) [23].

I chose to use the think-aloud method for evaluation to gain insight into potential issues and obstacles users may face as they are playing the game. It allowed me to observe and record the thoughts and actions that the game instigates, allowing for identification of potentially frustrating or unclear features of the system.

Each of the think-alouds was conducted on the same version of the system, with the only differences within the system between studies being the correction of typos and back-end bugs not associated with the user interface design.

6.1.1.1 Protocol

In each think-aloud session, participants were presented with a consent form. After gaining their consent, a script was read aloud to each participant, presenting the

purpose of the study and asking them to fill in a pre-game questionnaire testing their pre-existing knowledge and experience. Once done, the participants were trained by solving a simple math problem and thinking aloud as they were doing so. After the training, participants were asked to begin playing the game, being given the task of completing all six game levels. Following the completion of the task, each participant was asked to fill in a post-game questionnaire aimed at measuring their satisfaction with the system and the learning outcomes of the game.

The consent form, script and questionnaires can be found in Appendix C. The researcher script was adapted from the script presented in the Human-Computer Interaction course taught at The University of Edinburgh.

When observing the session, most attention was attributed towards any game elements which caused visible user confusion and obstructed their progress towards completing a level.

6.1.1.2   Experiment Details

In total 6 participants took part in the study. All the participants were UG4 computer science students with no previous experience in using Puppet. While each participant was able to complete the game, the completion time varied between sessions. Table 6.1 shows the time taken for each participant to finish all levels of the game.

During the think-aloud sessions, in addition to taking notes on how each participant progressed through the game, their voice was recorded for later examination.

Participant	Duration (mins)
P1	86
P2	79
P3	43
P4	56
P5	72
P6	51

Table 6.1: Table showing the time needed for each participant to complete all game levels in minutes.

6.2   Results and Discussion

6.2.1   Level Progression

In Section 4.2.2, the introduction of Puppet concepts in order of difficulty and the use of previously taught topics to reinforce knowledge gained was identified as one of the requirements (Req. 2) for my system. While, as shown in Table 6.1, the time spent

completing the game varied quite substantially between participants, by observing the time spent on each level we can evaluate whether the requirement was met.

With the aims of levels 1 through 3 being to teach about the use of the UI and basic Puppet concepts such as `classes`, `resources` and `variables`, it was expected of users to progress through those levels at a fast pace. As evidenced by Table 6.2, most participants slowed down once they encountered level 4, where advanced resource types with less intuitive behaviour were introduced. With the participants getting stuck and not being able to progress even after they examined the sample code and hints given to them, level 4 was often the first time where participants had to consult the knowledge base in order to obtain the information they were missing to complete a level.

In contrast to level 4, the level following it posed little problems to participants. Observation of the participants' actions during their progress through level 5 suggests that their comparatively fast progress was due to the level not making use of more advanced concepts introduced in level 4. Additionally, the low amount of code input needed to solve its puzzle contributed to the fast solving times.

While level 6 took the longest time to solve on average, most participants identified level 4 as the hardest, as most of the solving time of level 6 can be attributed to the length of the code input required. However, the usage of the newly introduced concept of `ordering` still caused confusion with all participants except **P3**, with the use of `ordering` being clarified only upon consulting the knowledge base, or through use of hints.

The level completion times shown in Table 6.2 suggest that the level difficulty progression objective was met in most cases, with levels 4 and 5 deviating from the trend. To balance the difficulty of levels in future iterations of the system, some concepts taught in level 4 could be instead introduced in level 5, contributing towards a smoother difficulty progression.

Participant	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
P1	12	11	8	16	11	26
P2	8	6	9	24	9	23
P3	6	4	6	8	5	12
P4	5	5	8	14	8	16
P5	10	11	13	14	9	15
P6	5	7	10	9	5	15
Average:	8	7	9	14	8	17

Table 6.2: Table showing the time spent on each level per participant in minutes, measuring the time from when each participant started the level introduction and until they finished the level.

In addition, the observation of participants suggests that the reinforcement learning component of the game was successful. When playing, participants spent most of their time creating code which made use of newly learned Puppet concepts, while quickly devising solutions for objectives requiring knowledge taught in previous levels. This

was most clearly shown in Level 4, where all participants except **P3** and **P6** had to repeatedly consult the knowledge base and sample code to identify mistakes made when using the newly introduced Puppet concepts while being able to identify the solution for all objectives using previously taught Puppet concepts.

6.2.2 Puppet Knowledge Explanation

6.2.2.1 Level Introductions

When presented with level introductions, participants took different approaches when reading the explanations of Puppet concepts present within them. While Participants **P1**, **P2** and **P5** thoroughly read the explanations, participants **P3**, **P4** and **P6** only skimmed through them, stating they would rather learn through playing the game instead. Due to the Puppet language often being self-descriptive, this approach worked for them, however, participants **P4** and **P6** experienced issues when solving later levels of the game, as they did not understand some of the more advanced concepts, having to resort to the knowledge base for explanations.

Participants **P1**, **P2** and **P5** also commented on there being a lack of in-depth explanations of some Puppet language concepts, such as details on the workings of each resource attribute and their possible values within the introduction. While the introductions are crafted to be short and feature the minimal amount information necessary, with additional explanations found in the knowledge base, the level introductions fail to convey that fact. Thus causing participants to be left confused, not knowing where to find further explanations.

Additionally, participants **P1** and **P4** commented on wanting a 'back' button in introductions, as they wanted to reread previous text pages after obtaining new information. Participants **P3** and **P4** also expressed concerns about the lack of formatting within the text, saying that the use of different or bold fonts for certain parts like the example code or important information would add to the readability of the explanations. With the level introductions featuring gradually appearing text, the scrolling speed of it was too slow for participants, with only **P3** and **P4** finding out that if the continue button is pressed before the text fully appears, the animation is skipped, and the full text is displayed. This suggests that the 'display full text' feature of the continue button should be presented to the users at the start of the game, allowing them to read the text at a faster pace than the scrolling animations allow for.

Overall, participants understood the explanations present within the level introduction. While some participants expressed a wish for further clarification within the introductions, their questions were answered by either the sample code provided in each level or the knowledge base.

6.2.2.2 Knowledge Base

The aim of the knowledge base was to be a companion for the level introductions, containing additional explanations on Puppet language concepts not present in the introductory texts. All users were expected to review the contents of the knowledge base when they would require further clarification of concepts the introductions fail to explain. However, except for participants **P1** and **P6** who explored the knowledge base in level 1, most participants ignored the knowledge base, with **P2**, **P4** and **P5** only reading through parts of it once they got stuck in level 4, and **P3** never opening the knowledge base at all.

Once participants found and read through the knowledge base, they commented on it being a good addition to the game, as it answered some of the questions they had. All participants except **P3** and **P6** used the knowledge base to resolve an error in their input at some point during their play session.

As evidenced by the think-alouds, the knowledge base served its purpose of clarifying concepts which the level introductions do not fully explain. However, as the knowledge base is never described as a mandatory component of the game and the users are never explicitly told that reviewing it is fundamental to their understanding of certain Puppet language concepts, most of them ignored the knowledge base until they exhausted all other options when seeking solutions to errors. When asked about why they did not visit the knowledge base earlier in the game, **P2**, **P3**, **P4** and **P5** all answered by saying they that thought the exploration of the knowledge base was optional in addition to being unsure about its contents. As such, I believe more emphasis should be put on describing the knowledge base to the users and highlighting its importance, possibly introducing the knowledge base in the UI tutorial present in level 1.

6.2.3 User Code Input

After participants completed the UI tutorial and started playing through the first level they were able to navigate the system without any issues, being able to identify what certain parts of the UI are used for. All participants commented that the UI layout and button functions were intuitive and easy to use.

While the function of each button was clear to all participants, **P1**, **P2**, **P4** and **P6** often forgot that their selection of input must be confirmed by pressing the 'tick' button, only to be reminded about it by the hints given upon submission. However, since the hints do not point out the exact error location, identifying the unconfirmed input selection to be the source of their errors, caused **P1** and **P2** to spend several minutes trying to pinpoint which part of their solution is wrong before realizing what their mistake was.

Overall, navigating the code input section posed little problems to participants. In particular, all participants enjoyed the feature of being able to modify all code entries instead of having to delete and remake them, which in turn saved great amounts of time when attempting to fix errors.

6.2.4   Hints

The feedback obtained on the hints generated when submitting an incorrect solution was mostly positive. Participants found the hints helpful in allowing them to determine which parts of their input they should focus on when identifying errors.

However, as the hints are general and meant to point users to the error location, they do not specify exactly what the error is and how to mend it. Therefore, when **P1**, **P2** and **P5** encountered situations where they, upon revision of all sources of help available to them (knowledge base, sample code and hints), still were not able to determine what is causing the error, the general pointers of the hints were of little help. While participants were able to resolve their issues through trial and error, the hints were aimed to be a sufficient unit of help for users when they get stuck. To address this issue, when users are unable to progress, after a certain amount of unsuccessful solution submissions, a more focused hint should be given, indicating exactly what the error is in order to allow them to progress.

With the hints incorporated in the system, Requirement 5, described in Section 4.2.2, was partially met. While the system does provide hints based on user input errors, these by themselves might sometimes not be sufficient to allow users to progress.

6.2.5   Level Rewards

All participants showed visible relief and satisfaction when completing a level and encountering the 'victory screen'. Participants **P1** and **P2** commented on the console display of their actions affecting the system, stating that seeing the impact of their submitted code was enjoyable and rewarding.

With one of the requirements of the system being the incorporation of rewards for completing levels to facilitate the 'eureka factor' of puzzle-based learning described in Section 3.2, the display of the user input in use was successful and was identified to be a stimulating reward for level completion.

6.2.6   Questionnaires

Having recruited participants with no experience in the use of Puppet, the participants were given a pre-game questionnaire aimed at obtaining an overview of their background knowledge applicable to the learning objectives of the game. After the participants finished playing the game, they were given a post-questionnaire aimed at testing whether the game was successful at teaching some of the concepts indicated as learning outcomes in Section 5.4, as well as obtaining their opinion on the system.

This section discusses the results obtained from the questionnaires which can be found in Appendix C.3.

6.2.6.1 Pre-questionnaire

All participants indicated a similar understanding of what configuration languages are, describing them roughly as "languages used to configure a system". In the pre-questionnaire participants were asked four questions about their experience in topics related to Puppet and my game system:

- **Q2**: Have you ever worked with any configuration language, eg.configuration files written in TypeScript, JSON?

- **Q3**: Do you know what a 'declarative programming language' is?

- **Q4**: Do you have any previous experience with system administration, such as managing servers, or setting up multiple Linux environments?

- **Q6**: Have you ever installed a package like Apache or OpenSSH using a Linux terminal?

By analyzing the results shown in Table 6.3, we can observe that the participant group taking part in my study had very varied experience in the concepts used within my game.

However, when observing the answers to **Q5**, asking participants how many hours per week they spend playing video games, we can observe a possible correlation to their time taken to finish all six levels. With **P3** completing the game in the shortest time and most weekly time spent on video games, followed by **P4** and **P6**, we can observe that the participants with more video gaming experience were able to complete the game faster. This observation suggests that the system is more suitable for users with video gaming experience.

With all participants being able to complete the game while having no previous experience in using Puppet, Requirement 3 described in Section 4.2.2 was met. Additionally, while **P3** and **P4** indicated in **Q4** that they have no previous experiences with system administration, they were still able to complete the game with arguably fewer obstacles as other participants, as suggested by their completion time.

Participant	Q2	Q3	Q4	Q6	Q5	Time Taken
P1	No	Yes	Yes	Yes	1-3 hours	86
P2	No	No	Yes	No	1-3 hours	79
P3	Yes	Yes	No	No	>15 hours	43
P4	Yes	No	No	Yes	4-7 hours	56
P5	Yes	No	Yes	Yes	1-3 hours	72
P6	No	Yes	Yes	Yes	4-7 hours	51

Table 6.3: Table showing the answers to questions posed in the pre-questionnaire, as well as the time taken for each participant to complete the game. Questions 2, 3, 4 and 6 ask about previous Puppet related knowledge. Question 5 asks how many hours per week participants spend playing games.

Participant	Liked	Disliked	Suggestions
P1	Sample Code	Restart button, Knowledge base layout	
P2	Hints, Code input, UI layout	Objectives layout	
P3	Code input	Objectives layout	
P4	Hints, Code input		Highlighting of code areas containing errors
P5	Difficulty progression	Level introductions	Less solution examples in sample code
P6	Code input, UI layout	Restart button	

Table 6.4: Table showing the post-questionnaire feedback for each participant, stating aspects of the system they liked or disliked and suggestions for improvement given.

6.2.6.2   Post-questionnaire

Participant Feedback:
Questions 1-4 of the post-questionnaire were aimed at obtaining feedback on how engaging the game is, as well as to identify positive and negative aspects of the game design.

With four participants marking the game was 'very engaging' and two participants choosing 'extremely engaging', the results indicate that the puzzle-based approach using elements of game-based learning was successful in motivating and engaging the participants. After completing the post-questionnaire, participants **P1** and **P5** commented that, while they at first were not interested in Puppet, the game appealed to them, sparking their interest in the topic of Puppet, as well as making them want to use the tool in their personal projects.

While participants listed different features of the system which they enjoyed most, the most prevalent ones were the code input, hints and UI layout, with the most disliked features being the objectives layout and the restart button position, which **P1** and **P6** accidentally clicked when trying to open the objectives window.

In future iterations of the game, the restart button would prompt a confirmation form and not immediately restart the game, to avoid possible input errors. The objectives layout issue would be addressed by listing tasks in numerical order and allowing users to collapse and expand their content, thus ensuring the users are not overwhelmed with text when they encounter a level with a long list of objectives.

Learning Outcomes:
Questions 5-9 were aimed at evaluating the learning outcomes of the game. They

asked the users to answer questions related to *resource declarations*, *classes*, *class declarations* and *execution order*. All six participants were able to correctly answer all questions testing the learning outcomes, indicating that the system was successful in teaching its users.

While all participants were able to answer every question, when asked about the attribute `enable` present in resources of type `service`, users were unsure on what exactly the outcome of setting the attribute value to 'true' was. With the attribute defining whether a service should be run on system startup, its name is not necessarily indicative of its behaviour. Since the system explains the working of the attribute only in the knowledge base, the information was not accessed by most users, highlighting an issue of having information present exclusively in the knowledge base. This could be addressed either by making the knowledge base a compulsory element of the game, as is outlined in Section 6.2.2.2, or by moving some descriptions exclusive to the knowledge base the the level introductions.

6.3   Study Outcomes

Through the observation of the participants' actions while interacting with my system, as well as their feedback, the study showed that the game was overall successful in both teaching its users, as well as keeping them motivated throughout their learning process. Additionally, valuable feedback was obtained on some negative aspects of my system, allowing me to identify features which could confuse or obstruct users when playing.

As indicated by participants, even though they previously had no interest in learning about Puppet, my system encouraged them to explore the subject and presented the knowledge through interactive and engaging methods. In particular, all participants pointed out the user interface to be well designed and intuitive, with the code input methods allowing for quick creation and modification of code without being overly restrictive.
Game levels were found to be of sufficient difficulty, as they prompted users to make use of their critical thinking and problem solving skills instead of following patterns, making level completion feel rewarding. This was evidenced by participants showing visible relief on level completion and their positive reaction when encountering the reward presented when finishing a level.
The participants' comments on the teaching support in the form of level introductions, knowledge base, sample code and hints were largely positive, indicating that the explanations present within the game are clear and understandable by users.

Additionally, the results of the post-game questionnaire suggest that my game was successful in teaching its users about the knowledge specified as learning outcomes in Section 5.4, indicating that the goals of my system as a teaching tool were met.

While the feedback obtained from participants was overwhelmingly positive, the study highlighted several negative aspects of the system. Below I summarize what the negative aspects are and how they can be addressed.

Difficulty progression: As highlighted in Section 6.2.1, the difficulty progression of levels was inconsistent in levels 4 and 5. While level 4 featured a significant jump in difficulty, the difficulty level in level 5 was in comparison substantially lower. To address the issue I suggested to shift some Puppet concepts taught in the fourth level to level 5, smoothing out the difficulty curve of the game.

Hints: When participants got stuck while playing, they would consult the hints for advice. With the advice presented in the hints only pointing to the code block in which an error is made, that information by itself might not be sufficient for users to resolve the error. As such, in Section 6.2.4, I suggest the inclusion of additional, more focused hints indicating exactly what the error is, which would be presented to users after a certain amount of unsuccessful code submissions.

Knowledge Base: The participants often visited the knowledge base late during their playthrough, or ignored it completely, assuming it was not mandatory for their progression through the levels. As the contents of the knowledge base are not properly explained within the game, it is often not perceived as useful or necessary. In Section 6.2.2.2, I suggest the inclusion of the description of the knowledge base in the UI tutorial as a possible solution, as it would make users aware of its existence and contents earlier in the game.

Level Introductions: During the study, four issues were identified with the level introductions:

- Some participants were confused as to why the level introductions do not fully explain all Puppet concepts. To shorten the introductory texts, some of the explanations were moved to the knowledge base. However, the users are not explicitly made aware of that fact, causing confusion during gameplay. This could be addressed by fully explaining all Puppet concepts in the level introductions, or by explicitly stating that the explanations are available within the knowledge base.

- Participants indicated that level introductions could benefit from additional formatting features to increase readability. These could be included through the addition of bold fonts to highlight important knowledge, as well as the use of different fonts for text parts such as code samples.

- Although the game includes a feature of skipping text animations by pressing the 'continue' button while the level introduction text is still appearing, users are never made aware of it. Adding a short notice about the feature at the start of the level 1 introduction would make users aware of its presence.

- Some participants wished to reread introductory texts available on text screens they already navigated past. This feature could be added through the inclusion of a 'back' button allowing access to previous screens in level introductions.

Objectives: As highlighted in Section 6.2.6.2, participants expressed concern about the structure in which objectives are presented to users, stating that the lack of formatting makes them hard to read. To address this issue, I suggested to list individual tasks present in the objectives in numerical order, allowing users to expand and collapse their content.

Restart Button: Two participants accidentally clicked on the 'restart level' button when attempting to view the objectives, causing them to lose their progress within the level they were solving. In Section 6.2.6.2 I suggested adding a confirmation form asking users to confirm whether or not they would like to restart the level.

6.4 Summary

In this chapter, I discussed the method used to conduct my evaluation of the system. The method I chose for testing the usability of the system was think-alouds in conjunction with a pre-questionnaire testing the participants' background knowledge and a post-questionnaire aimed at obtaining user feedback on the system and testing on whether the system was successful at teaching the users.

I discussed the results of the six think-alouds conducted, identifying issues that arose during the sessions and proposing solutions. A list of the main positive and negative elements of the system can be found in Table 6.5.

Positive Elements	**Negative Elements**
• Introductions are clear and understandable • Knowledge base is easy to navigate • UI is intuitive and behaviour of buttons is predictable • Code input is intuitive • Hints provide intended information on the error type and location • Level completion is rewarding • Sample code is useful when crafting code solutions	• Level difficulty progression is inconsistent in levels 4 and 5 • Knowledge base is not portrayed as mandatory and its use not properly explained • Level introductions do not fully explain all Puppet concepts • Level introductions are missing a 'back' button • No indicator of level introduction text animations being skippable • The introduction texts lack meaningful formatting • Hints do not provide specific enough information when users get stuck • The restart button position makes it easy for users to miss-click on it • The objectives text layout makes the objectives hard to navigate

Table 6.5: Table summarizing the results of my evaluation, grouping them into positive and negative elements of my system.

Chapter 7

Conclusion

7.1 Overview

With configuration errors being one of the leading causes of system failures and a large shift of system administration population occurring, as the system administrator group expands to include semi- and non-professional administrators, there are not enough tools available to facilitate their learning process. To contribute towards addressing that issue, I developed a game system that teaches the basic concepts of the Puppet programming language to aspiring system administrators, answering the following question:

"How can we design a learning tool for teaching the Puppet language to new system administrators with lacking background knowledge?"

Through the use of sample code and explanations, my system teaches its users about the Puppet language concepts of *resources*, *variables*, *classes* and *execution order*. The learning process is facilitated with the use of puzzle- and game-based learning, with users being tasked to solve a puzzle in each game level featuring newly learned Puppet concepts, as well as concepts taught in previous levels.

Each game level contains a level introduction offering explanations of taught concepts, a set of objectives describing the desired system state, as well as sample code and additional explanations of taught topics. Users must craft their own code solution that meets the level objectives through the use of the game's input system.

I carried out an evaluation of my system aimed at testing my game's usability and suitability as a learning tool. The focus of the study was testing whether my system's UI is intuitive and usable by beginners, while also being successful at teaching. To evaluate my system I conducted six think-alouds accompanied with pre- and post-game questionnaires.

With all participants being able to complete every level featured in my game in addition to successfully answering all post-game questionnaire questions aimed at evaluating the learning outcomes of the game, I concluded that that my system is successful in teaching its users. While the participants' comments suggested that the UI features in

my system are intuitive and easy to use, as well as its design being engaging, as seen in Chapter 6, there are still several aspects of the game which can be improved.

7.2 Future Work

With several issues with my system identified in Chapter 6, as well as features which were not possible to implement within the scope of this project, I present a list of future work to be done on my system:

- Address usability and implementation issues identified in Chapter 6

- Implement additional Puppet code input options, allowing the use of other Puppet language concepts within my game

- Develop a level editor for my system, allowing for easy customization of its levels for anyone looking to use the game, wanting to design and implement their own levels

- Refactor and document the source code to prepare it for publishing as an open-source project allowing for modifications and additions to the system by parties interested in the project

Bibliography

[1] C. Burt, "Level 3 error behind widespread us internet outage," 2017. [Online]. Available: https://www.datacenterknowledge.com/uptime/level-3-error-behind-widespread-us-internet-outage/

[2] R. Johnson, "More details on today's outage," 2010. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/

[3] CircleID, "Misconfiguration brings down entire .se domain in sweden," 2009. [Online]. Available: http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden/

[4] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043572

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, ser. USITS'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251460.1251461

[6] W. Fu, J. Cheney, and P. Anderson, "An operational semantics for a fragment of the puppet configuration language," *CoRR*, vol. abs/1608.04999, 2016. [Online]. Available: http://arxiv.org/abs/1608.04999

[7] T. Xu, V. Pandey, and S. R. Klemmer, "An HCI view of configuration problems," *CoRR*, vol. abs/1601.01747, 2016. [Online]. Available: http://arxiv.org/abs/1601.01747

[8] Puppetlabs, "Quest guide for the puppet learning vm," 2019.

[9] "Puppet 5.3 reference manual," 2018. [Online]. Available: https://puppet.com/docs/puppet/5.3/index.html

[10] "Puppet 6.1 reference manual," 2018. [Online]. Available: https://puppet.com/docs/puppet/6.1/puppet_index.html

[11] S. Deterding, M. Sicart, L. Nacke, K. OHara, and D. Dixon, "Gamification: Using game design elements in non-gaming contexts," vol. 66, 01 2011, pp. 2425–2428.

[12] K. Begnum and S. S. Anderssen, "The uptime challenge: A learning environment for value-driven operations through gamification," *USENIX Journal of Election Technology and Systems (JETS)*, vol. 1, no. 1, Submitted. [Online]. Available: https://www.usenix.org/jesa/0201/begnum

[13] J. PARKER, "The use of puzzles in teaching mathematics," *The Mathematics Teacher*, vol. 48, no. 4, pp. 218–227, 1955. [Online]. Available: http://www.jstor.org/stable/27954863

[14] B. Parhami, "Puzzling problems in computer engineering," *Computer*, vol. 42, no. 3, pp. 26–29, March 2009.

[15] N. Falkner, R. Sooriamurthi, and Z. Michalewicz, "Puzzle-based learning for engineering and computer science," *Computer*, vol. 43, no. 4, pp. 20–28, April 2010.

[16] K. E. Merrick, "An empirical evaluation of puzzle-based learning as an interest approach for teaching introductory computer science," *IEEE Trans. on Educ.*, vol. 53, no. 4, pp. 677–680, Nov. 2010. [Online]. Available: https://doi.org/10.1109/TE.2009.2039217

[17] Z. Michalewicz and M. Michalewicz, *Puzzle based learning: An introduction to critical thinking, mathematics and problem solving*.   Hybrid Publishers, 2008.

[18] "Puppet learning vm." [Online]. Available: https://puppet.com/download-learning-vm?_ga=2.196239663.2041492960.1548267585-1340069746.1543357651

[19] "Puppet enterprise." [Online]. Available: https://puppet.com/products/puppet-enterprise

[20] K. Henner, "Puppet learning vm: a new lesson architecture," 2017. [Online]. Available: https://puppet.com/blog/puppet-learning-vm-new-lesson-architecture

[21] C. Schroeder, "Scaffolded assignments: Designing structure and support," 2012. [Online]. Available: http://www.hartnell.edu/sites/default/files/llark/designingscaffoldedassignmentsbooklet_1_.pdf

[22] U. Technologies, "Unity game development platform." [Online]. Available: https://unity.com/

[23] B. Martin, B. Hanington, and B. Hanington, *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*.   Rockport Publishers, 2012.

Appendix A

UI Assets

Figure A.1: Image used for the main gameplay scene background.

Figure A.2: Image used for the level introduction background.

Figure A.3: Image used for the objectives and knowledge base background.

Figure A.4: Image used for the text background in the level introduction scene.

Figure A.5: Image used for the navigation buttons and the "New Object" button.

Figure A.6: Image used for the '+' button used to add components to code blocks.

Figure A.7: Image used for the 'x' button used to remove code components and close pop-up style windows such as Objectives and the Knowledge base.

Figure A.8: Image used for the 'tick' button used to confirm input selection.

Figure A.9: Image used for the 'x' button used to modify confirmed input. Is available after pressing the 'tick' button.

Figure A.10: Image used for the 'expand' button used to expand text entries within the knowledge base or to show hints when submitting incorrect solutions.

Figure A.11: Image used for the 'collapse' button used to collapse text entries within the knowledge base or to hide hints when submitting incorrect solutions.

Appendix B

System Screenshots

Sample Code:

Objectives

Level 3:
- Create and invoke a class titled alice.

When the class alice is invoked:
- a user 'alice' with uid 1 and a home directory '/disk/users/alice' must be created.

- '/disk/users/alice' must be ensured to be a directory.

Note: Make use of the $dir variable when setting the home directories and resource names.

Restart Level   Submit!   New Object
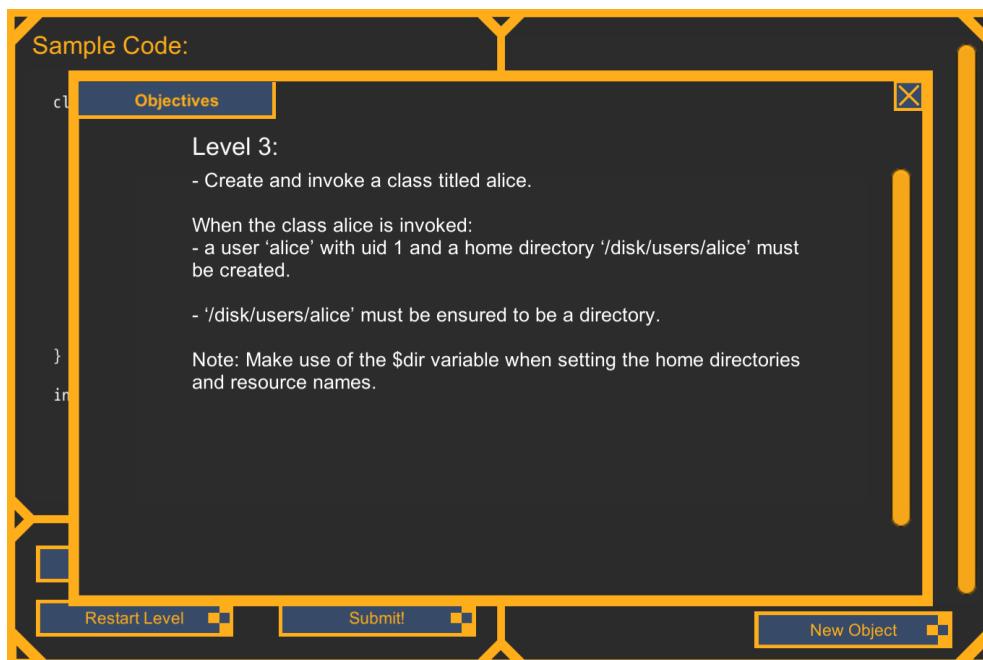
Figure B.1: Screenshot of the objectives of level 3.

Sample Code:

Knowledge Base

Manifests

Resources

Classes

Variables

Ordering

Restart Level   Submit!   New Object
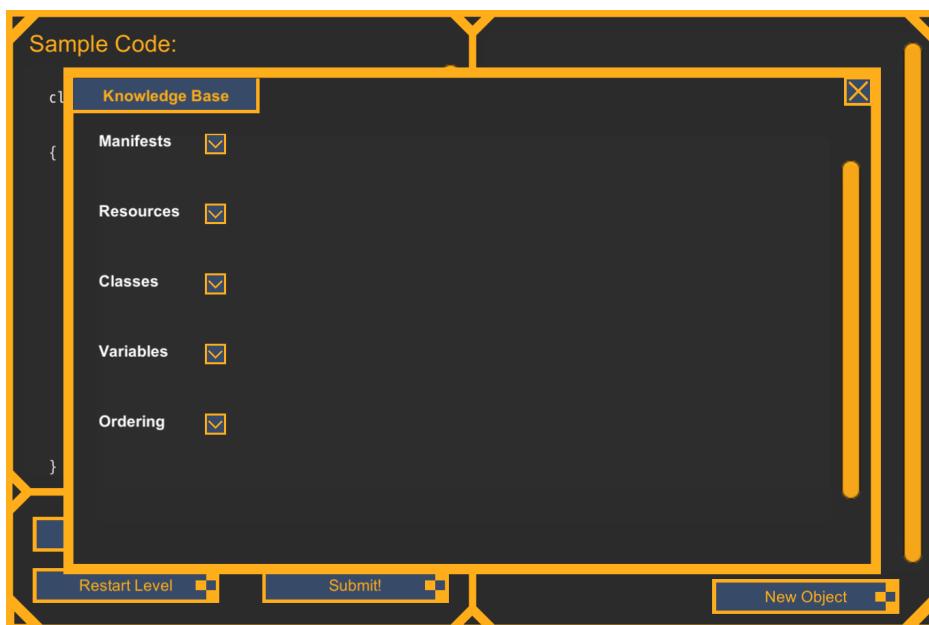
Figure B.2: Screenshot of the knowledge base of level 6.
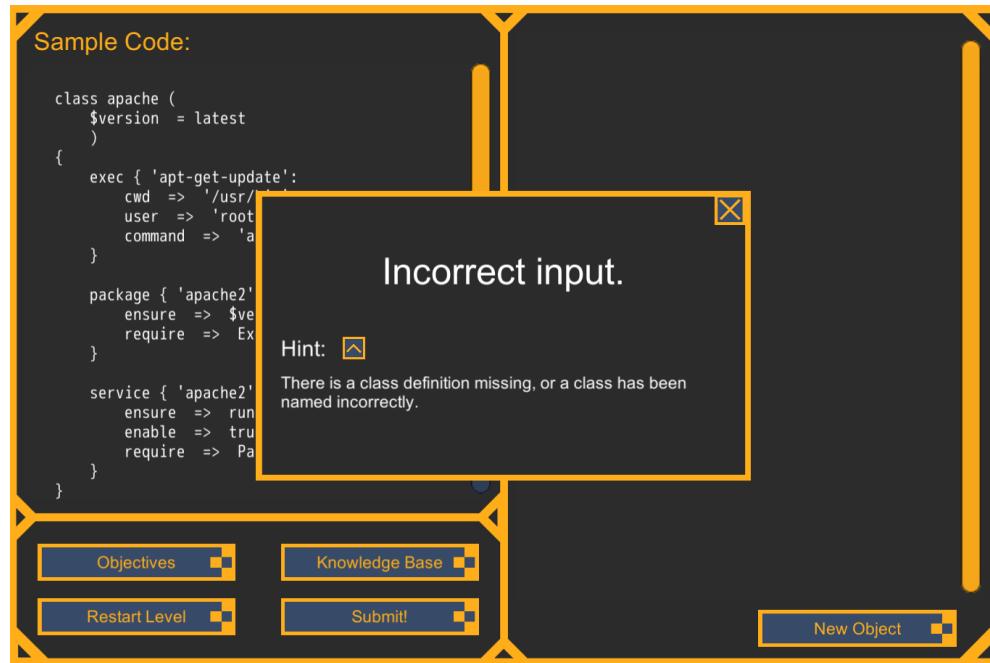
Figure B.3: Screenshot of a hint displayed to the user when submitting an incorrect solution in level 6.

Figure B.4: Screenshot of the console screen displayed to the user when successfully finishing level 3.

Figure B.5: Screenshot of the final screen of the game, shown to the user once they complete the final level.

Appendix C

Think-Aloud Material

C.1   Consent Form

Learning the Puppet language using an educational game

I would like to invite you to take part in my research study about my Puppet programming language learning game. Before you decide if you would like to take part, please take the time to read the following information on why this research is being done and what it involves for you. Feel free to ask any questions at any point.

Purpose
I am conducting this study as a part of my Honours Project at the University of Edinburgh. My goal is to develop a useful learning game that teaches people how to accurately write code in the Puppet language.
The researcher is:
  • Jaka Mohorko, University of Edinburgh, s1552344@sms.ed.ac.uk
The project is supervised by:
  • Dr. Kami Vaniea, University of Edinburgh, kvaniea@inf.ed.ac.uk

Taking Part
Today I will be asking you to answer some questions about your prior experience and demographics. You will then play my game, followed by answering some more questions about the game design. I expect that the session will take no longer than 90 minutes.

Your participation is entirely voluntary, and you may withdraw at any point, even if you initially agree to complete the study. You do not need to give a reason in case you want to withdraw.

Your Data
If you decide to take part in the study, I will ask you to fill out a short survey before and after the game play to measure the impact of the game on your pre- and post-session knowledge and experience. The survey data will be aggregated with that of the rest of the participants and presented only in aggregate form (e.g.: "40% of our participants felt engaged when playing the game").

Additionally, your voice and actions on the screen will be recorded. These recordings will be stored locally and only ever be accessible by the researcher.
I will not videotape you, I will only capture what is happening on the screen. Furthermore, I will be asking you to speak outloud as you play the game and I may transcribe what you say and anonymise any identifying information. I may quote parts of this transcript in our report using anonymous identification (e.g.: "Participant 2: '..' ").

The report itself, containing analysed and anonymised data and quotes will be made publicly available. The collected raw data – voice, screen recordings, and survey sheets will however be deleted after my report has been turned in and marked

approximately at the end of May 2019. If you decide to withdraw from the study, all the raw information and data collected from you will be destroyed immediately.

Compensation
You will not receive any compensation for taking part in this study, other than the knowledge that you have benefited science.

Your Consent

☐ I understand that my session will be screen and audio recorded.

☐ I understand that my participation in this study is completely voluntary and that I can withdraw from the study at any point in time without giving a reason.

☐ I understand that I can ask for the information I provide to be deleted/destroyed at any time. I can also ask for a copy of my data at any time before it is destroyed.

I, _____ **[PRINT NAME]**
consent to participate in the study conducted by Jaka Mohorko.

Signed: _____

Date: _____

C.2   Researcher Script

1 Researcher Script

Hello my name is Jaka Mohorko.

Today we will be playing Puppet language learning game PuppEzy to learn the fundamentals of the Puppet programming language. You will not be entering any financial information and your participation today is purely voluntary, you may stop at any time.

The purpose of this exercise is identify issues with the Puppet learning game. Please remember we are testing the game, we are not testing you.

Before you start playing the game, I would like to ask you to fill in a questionnaire investigating your current experience. We are not expecting you to be a professional, so do not worry if you can not answer some of the questions. The purpose of this project and game is to teach the users. Therefore it is not necessary for you to have previous knowledge of the subject.

Hand them the questionnaire

After you are done playing the game, I will ask you to fill in a post-questionnaire. The aim of that questionnaire is to gauge whether or not the game was successful at teaching and to obtain your opinion on the game.

We will now start the talk-aloud session.

1.1 Think aloud training

In this observation, we are interested in what you think about as you play through the levels of the game. In order to do this, I am going to ask you to talk aloud as you work on the task. What I mean by "talk aloud" is that I want you to tell me everything you are thinking from when you start playing the game, and until you finish playing. I would like you to talk aloud constantly from when you start playing and until you have finished the game. I do not want you to try and plan out what you say or try to explain to me what you are saying. Just act as if you were alone, speaking to yourself. It is most important that you keep talking. If you are silent for a long period of time, I will ask you to talk.

Do you understand what I want you to do?

Good. Now we will begin with some practice problems. First, I will demonstrate by thinking aloud while I solve a simple problem: "How many windows are there in my flat?"

Demonstrate thinking aloud.

Now it is your turn. Please think aloud as you multiply 120 * 8 [Let them finish]

Good. Now, those problems were solved all in our heads. However, when you are working on the computer you will also be looking for things, and seeing things that catch your attention. These things that you are searching for and things that you see are as important for our observation as thoughts you are thinking from memory. So please verbalise these too.

As you are playing the game, I won't be able to answer any questions. But if you do have questions, go ahead and ask them anyway so I can learn more about what kinds of questions the game brings up. I will answer any questions after the session. Also, if you forget to think aloud, I'll say, "please keep talking."

Do you have any questions about the think aloud?

Here is the game that you will be playing. You may begin once you're ready. You may begin.

C.3 Questionnaires

Pre-questionnaire:

1. What do you think the term 'configuration language' refers to?

   _____

   _____

2. Have you ever worked with any configuration language, eg. configuration files written in TypeScript, JSON?
   a. Yes
   b. No

3. Do you know what a 'declarative programming language' is?
   a. Yes
   b. No

   If yes, briefly explain the term 'declarative programming language'.

   _____

   _____

4. Do you have any previous experience with system administration, such as managing servers, or setting up multiple Linux environments?
   a. Yes
   b. No

   If yes, briefly summarize your experiences.

   _____

   _____

   _____

5. How many hours per week do you spend playing video games?
   a. 1 - 3 hours
   b. 4 - 7 hours
   c. 8 - 15 hours
   d. More than 15 hours
   e. I don't play video games

6. Have you ever installed a package like Apache or OpenSSH using a Linux terminal?
   a. Yes
   b. No

Post questionnaire:

1. How engaging did you find the game?
   a. Not at all engaging
   b. Slightly engaging
   c. Moderately engaging
   d. Very engaging
   e. Extremely engaging

2. What features of the game did you like most?

   _____

   _____

3. What features of the game did you like the least?

   _____

   _____

4. Do you have any suggestions on how the game could be improved?

   _____

   _____

5. Briefly describe what are Resource Declarations in the Puppet language and how they are structured.

   _____

   _____

   _____

6. Briefly describe what are Classes in the Puppet language and how they are structured.

   _____

   _____

   _____

7. Given the following class definition, which way of invoking the class would you choose if you wanted to create a user named 'alice'?

```
class add_user (
    $user = 'default'
    )
{
    user { $user:
        ensure => present
    }
}
```

a) An include function
b) A resource-like class definition
c) Neither of above

8. Order the Resource Declarations below by their execution order. Mark the Resource that will be applied first as 1 and the one applied last as 4.

```
package { 'apache2':
    ensure => installed,
    require => Package['nginx']
}

package { 'nginx':
    ensure => absent,
}

file { '/backup':
    ensure => directory,
    require => Service['apache2']
}

service { 'apache2':
    ensure => running,
    require => Package['apache2']
}
```

_____   package { 'apache2'

_____   package { 'nginx'

_____   file { '/backup'

_____   service { 'apache2'

9. Briefly describe the system state parameters that are enforced when the following code is applied.

```
package { 'nginx':
      ensure => installed
}

service { 'nginx':
      ensure => running,
      enable => true,
      require => Package['nginx']
}

user { 'root':
      ensure => present,
      uid => 1,
      home => '/root'
}

file { '/root/backup':
      ensure => directory,
      require => User['root']
}
```

_____

_____

_____

_____