Alexandros Gouvatsos, CDE, Friday, 04 April 2013

TAINT: Text Adventure and Interactive Novel Toolkit

1 INTRODUCTION

This report breaks down the structure and initial design of an open source text adventure and interactive novel engine with the codename TAINT (Text Adventure and Interactive Novel Toolkit).

The first section of the report explains the overall concept of a text adventure and interactive novel and the motivation behind creating an engine. Then it presents and analyses previous approaches to tackling this problem, which have served as inspiration and as a guide.

The second section, which is the main emphasis of this report, goes into the initial design of TAINT, including the considerations and assumptions that have been gathered until now to form functional and non-functional requirements for the system. It also discusses the scope of the project in order to realistically account for the time-frame within which a prototype of the system must be completed. After the main building blocks of the design are underlined, they are formalised through the use of design tools such as use cases, flow charts and class breakdowns. Apart from internal, back-end design the second section also includes some considerations for the graphical user interface (GUI) which is an important part of the project.

Finally, the third section summarises the goals and briefly discusses potential issues and difficulties that may arise.

2 1. BACKGROUND

2.1 MOTIVATION

Text adventures and interactive novels are another way to serve written fiction. It is similar to reading a book, however instead of having a completely passive experience, the reader can make choices that affect the course of the story. In this report, the term 'text adventure' is used for such interactive fiction where the reader interfaces with the system through commands, which are hidden to him. On the other hand, the term 'interactive novel' is used for interactive fiction where all the possible actions the player can take are always visible, like a multiple-choice list.

Text adventures and interactive novels were extremely popular back when computers were not as powerful as today and a good story was the focus of games, instead of impressive visuals. With the coming of more powerful computers, these games became overshadowed by modern games which could combine a good story together with graphics, more intense game-play and realistic sound.

However, with the coming of portable devices like smart phones and tablets, these games have become relevant again: not because today's devices lack the computational power to play more advanced games, but due to the environment in which people use these devices to play. More specifically, common experience shows that people read or play on their portable devices when in-between locations or tasks. Because of the short-term nature of these sessions, deep games with complex mechanics which require the user's full audio-visual attention are not as popular, compared to simple games which require little input from the user.

As such, text adventures and interactive novels have the potential to make a comeback. It is important to note that this does not mean that it is expected for these games to claim a greater market share than games with actual graphics, but rather to become more popular than they have been in the recent past.

The main motivation of the TAINT project is to allow non-technical people to bring their own stories to life, which they can then share with their friends and family or even release on a greater scale.

2.2 Previous work

In IFComp of 2006, an interactive fiction competition, most games were implemented in Inform (Graham Nelson, 2010), while a minority of games were also made using TADS (High Energy Software, 2006) and ADRIFT (Campbell Wild, 2012).

TADS is a programming language especially made for creating text adventures. While it was originally very popular, it was slowly overthrown by Inform. It is worth noting that no matter how simple a programming language, it might still be demoralising for someone who just wants to write a good interactive story.

ADRIFT also focuses specifically on text adventures but is on the opposite side from TADS: it relies on a heavy graphical interface which can help the user keep everything in sight. The interface includes a node map which keeps track of states and how they are connected to one another. Additionally it includes windows for grouping areas, items and characters together – something which might be intimidating at first but very useful for larger projects.

Inform takes a different approach, which is that of natural language processing. Although there is an interface element which can display a node map as well as a branching editor, Inform's power lies in the fact that it focuses on the strength of story writers: writing. Instead of having to learn a programming language (no matter how simple) or to tackle complex interfaces, Inform allows the user to intuitively define variables or descriptions through the actual use of the English language. For example to define a bidirectional connection between two rooms one can simply write 'The Cloakroom is west of the Foyer'.

It is worth mentioning that most engines work through the concept of a virtual machine (VM). The person who wants to play a game made in these engines only needs to have the corresponding VM installed on their system, allowing for a multi-platform experience.

Other modern examples of such ventures are Playfic (Andy Baio, Cooper McHatton, 2013), Twine (Sutton, 2012) and Quest (Warren, 2013). It is worth noting that allowing the games to be played independent of platform is also important for those three projects but it is achieved by making the stories web-based. By making everything web-based, a much greater sense of community is achieved which means that apart from the software, these projects provide a place to submit and to find new stories.

Playfic simply takes the game source and passes it to the Inform compiler, so it is entirely driven by natural language just like Inform. Therefore its contribution is the online hub.

Twine is an engine focused on interactive novels. It provides a very light graphical interface, which mostly includes a node map, in order for the user to see the interconnections between locations, objects and actions. However, the main development takes place in a text editor, which again means that the user would need to learn some light programming to take full advantage (it is based on the TiddlyWiki web tool). Twine also relies on the fact that it's web-based to allow for more than text to be displayed e.g. images.

Quest -the third web alternative- aims in providing a user-friendly experience through a detailed graphical interface. Focusing on text adventures rather than interactive novels, the idea is for the user to avoid any form of programming, such as variable definitions. It is worth noting, however, that the GUI can get crowded as the complexity of the game increases.

3 2. Design and methodology

3.1 Project scope

Creating an engine for interactive fiction can include many different aspects, for example text adventures and interactive novels purely from text, as well as their equivalents which include images or even short animations and sound. Moreover, it can be an engine which combines a UI with the scripted components architectural pattern (Robert C. Martin, 2011), to strike the perfect balance between ease of use and customisability. Apart from the development system, which is the tool with which the user creates the piece of interactive fiction, another point of interest is the interpreter. In this report, the interpreter is defined as the system which runs the final game, e.g. a VM, a website or a stand-alone application.

The scope for this project is to write a development system for both text adventures and interactive novels. Additionally, it should allow the user to do so without the need to do any programming. Instead the aim is for an easy to use GUI, which can provide sufficient depth and power without becoming too cluttered or intimidating. Nonetheless, some elements like e.g. the possible conditionals for certain actions to take place, might be extensible.

Since the engine is planned to support the creation of both text adventures and interactive novels, it must be noted that the distinction between the two adds another layer of complexity. More specifically, since the user interfaces with a text adventure game by typing commands, there is a command parsing element to be included. Parsing commands is where the text adventure complexity lies and it can be a project by itself. In this case the initial goal will be simple parsing of specific commands managed through keywords during the creation of the game; more complexity will be added only if there is enough time, without leaving other important elements behind.

The overall idea is that all the data of the stories will be externally stored in some format such as XML or JSON. Then the other two building blocks will be the development system and the interpreter. This approach presents several advantages.

Firstly, the separation between development system and interpreter allows for better modularity. This means that individual changes can occur in one of the two, e.g. an entirely new editor can be created (even by someone else), but as long as it adheres to the structure of the data of the interpreter, it will still be compatible.

Secondly, the interpreter can be re-written (in the worst case) for new platforms, allowing the stories to be enjoyed irrespective of the device without the need to port the entire engine.

Finally, this separation between system and data helps in the development itself. If the data representation of the story is simple and logical then the process for the user creating the stories is more likely to be so too.

3.2 Tools

The system is to be implemented entirely in Python 2.7.3 (Guido van Rossum, 2013). Python can run on Linux, Mac OSX, Windows and Android, though it is not necessary for the development system itself to be multi-platform. For the front-end, two libraries are currently being further inspected: PyQT (Riverbank Computing, 2013) and wxPython (Robin Dunn, Harri Pasanen, 2012). The initial interpreter will also be implemented in Python, although alternatives to prove the concept of multi-platform capabilities might be examined if the strict time frame allows it.

Documentation of the code is going to be handled via docstrings, using the standard for python documentation as explained in (David Goodger, Guido van Rossum, 2009).

Version control is handled through Git and the source code will be available on Github.

3.3 Development methodology

Even though pre-production design tools such as requirements gathering, use cases, flow charts and class breakdowns are used to help create an overall picture of what is to be developed, the methodology that will be followed will contain some agile elements.

The development is going to consist of short iterations, where each iteration is going to be focusing at the most important requirements. This means that at the end of every iteration there will be a re-evaluation of the requirements and the remaining time until the deadline.

Normally, this might create unwanted overhead due to the fact that the team has to agree on the next step. In this case the team consists of only one person, so the overhead is going to be very small, negating this disadvantage. It is important to note that this iterative approach (most development is iterative, anyway) means that the contents of this report –including the project scope- might differ greatly by the end.

It is not clear yet whether this development project can benefit from the test-driven aspect that many agile environments require. In case that it is deemed necessary, the default unit testing framework of Python 2.7 is most likely to be used.

3.4 System design

Initially, in order to provide a compass for the design process, a compilation of functional and non-functional requirements was generated as shown below. As mentioned earlier, these requirements are going to be re-evaluated on every iteration in order to rank them in terms of importance. Additionally, new requirements might be included in subsequent iterations if deemed necessary. The term user is used to identify the entity which uses the engine to create a piece of interactive fiction. The term player is used to identify the entity which received the final output, the finished piece of interactive fiction.

3.4.1 Functional requirements

1. The user must be able to create text adventures.
2. The user must be able to create interactive novels.

3. The user must be able to create scenes. A scene can represent a location or state in the story.
4. The user must be able to link scenes together.
5. The user must be able to control global information. Global information can be time or weather in the fictional world, the current health condition of the player or other data worth tracking.
6. The user must be able to create and control an inventory for the player. The inventory is used to store elements that the player carries within the fictional world.
7. The user must be able to create and control elements within scenes. These elements can be objects or containers.
8. The user must be able to create actions through which the player will interact with the fictional world. The resolution of actions taken by the player includes output narrative in textual form and manipulation of elements of the fictional world, global information or the player inventory.
9. The user must be able to set conditions for an action to be resolved. These conditions include a value of global information being within a certain threshold, or an item existing within the inventory.

3.4.2 Non-functional requirements

I. The system should allow for a piece of interactive fiction to be created entirely through the GUI.

II. The system should store the data of the output to be stored externally.

III. The *development system module* of the system should be responsible for the creation of output data representing a piece of interactive fiction.

IV. The *interpreter system module* of the system should be responsible for the playback of the data outputted by the *development system module*.

V. The *interpreter system module* of the system should be able to store the state of the player's progress within the fictional world. This includes the reached scene, global information and inventory data.

VI. The *interpreter system module* of the system should be able to load the state of the player's progress within the fictional world.

VII. The system's actions should be extensible through additional libraries which can be written separately.

VIII. The system should include a basic *command parsing module*. The *command parsing module* is used to parse commands input by the player and translate them into actions in the fictional world.

3.4.3 Use cases

The above requirements can be used to show the interaction between the users and the individual subsystems in the form of a use case diagram, as seen in Figure 1 and Figure 2. The requirements of allowing the user to create text adventures and interactive novels, is represented within the structure of the interactions that the user can create and is hidden from the use case diagram; however, it does appear in the class design and class breakdown.

Another point about the design process chosen is that, while perhaps there are advantages into treating the different subsystems as completely separate systems, due to the fact that many of the assumptions carry over from one to the other, they are still treated as part of one whole system. The centrepiece of this system is the development system module. Having said that, certain requirements, use cases etc. can be analysed separately in order to help with the implementation.

In this case the term database does not necessarily mean a web interface. It might simply be data structures stored on the disk.

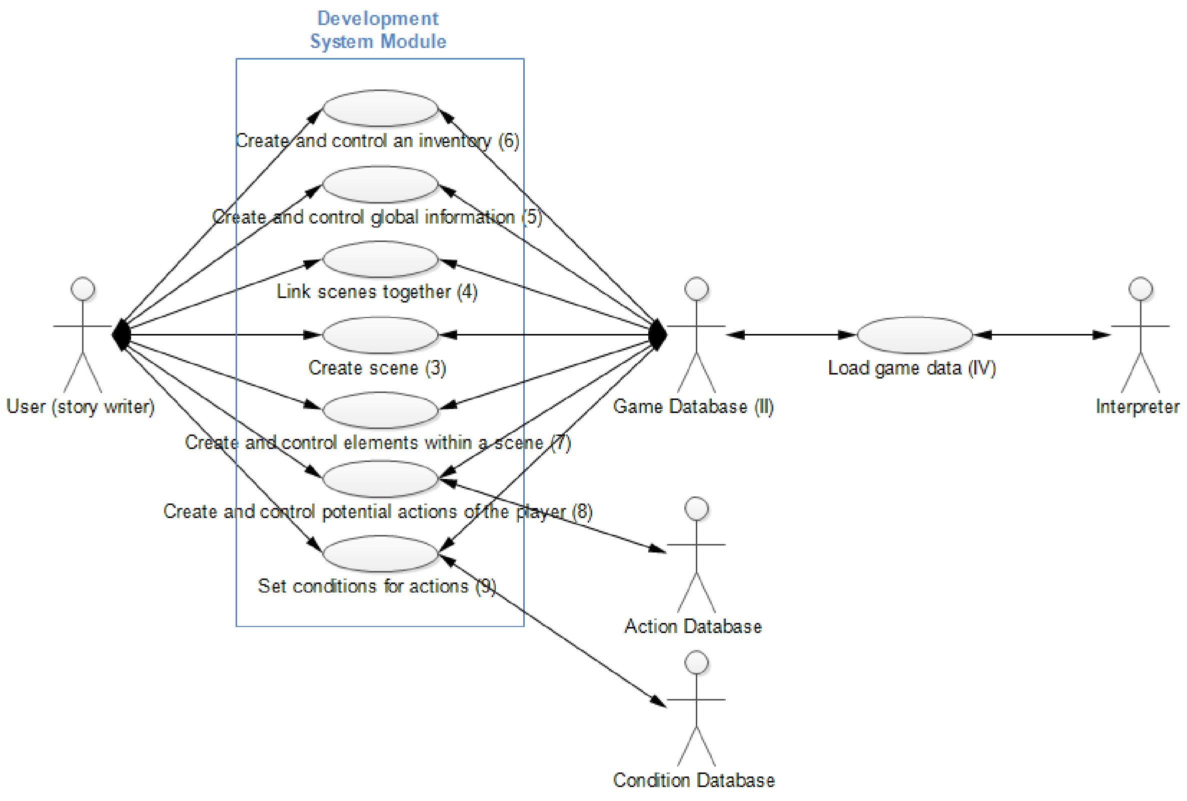
Development System Module
Create and control an inventory (6)
Create and control global information (5)
Link scenes together (4)
Create scene (3)
Create and control elements within a scene (7)
Create and control potential actions of the player (8)
Set conditions for actions (9)
User (story writer)
Game Database (II)
Load game data (IV)
Interpreter
Action Database
Condition Database

Figure 1 - Use case diagram for Development System Module

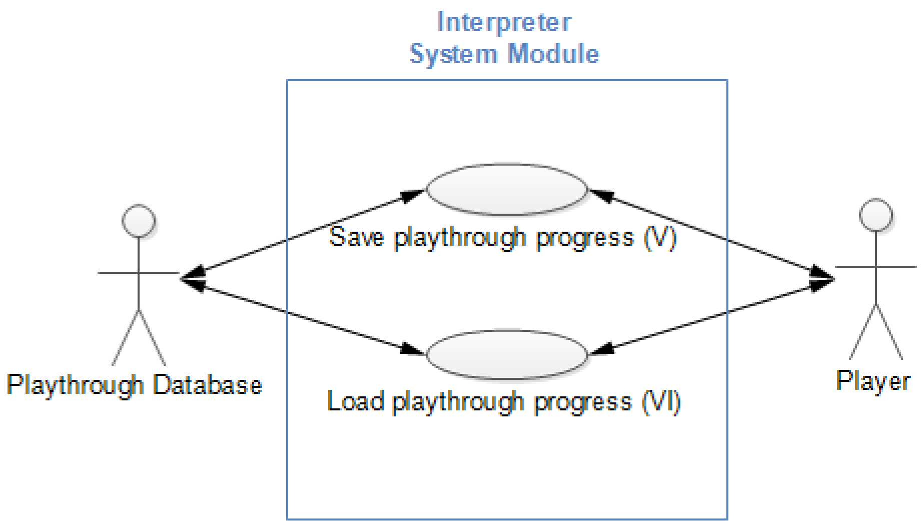
Interpreter System Module
Save playthrough progress (V)
Load playthrough progress (VI)
Playthrough Database
Player

Figure 2 - Use case diagram for Interpreter System Module

3.4.4 Class breakdown

At this stage in design, the overall logic is broken down into the classes as shown in Table 1. These classes are a concept of how the above requirements could be implemented using object-oriented methodology.

Class	**Description**
Scene	The Scene class represents a location or a state in the fictional world. Apart from a description which is used in the narrative it can contain element groups and actions that the player can trigger.
Resource	The Resource class represents global information, such as the date or the weather in the fictional world, or values like the player's health state.
Inventory	The Inventory class represents the player's inventory. It is still not certain if it will be a separate class or incorporated into the logic of Resource.
Action	The Action class represents an action that can be triggered by the player. An Action consists of a Condition and an Effect. The Effect is only resolved if the condition is true. An action can be visible (in the case of an interactive novel) or invisible (in the case of a text adventure). In fact, within the same game there might be cases when Actions are visible and cases when actions are invisible.
Condition	The Condition class represents a logical evaluation which can be part of an Action. When evaluated it returns a value true or false.
Effect	The Effect class represents the outcome of a given Action that the player performs. It includes changing the scene, interacting with the world or even losing the game.
EffectList	The EffectList class represents a collection of all potential Effects available for the user to add to Actions of their piece of interactive fiction. It should be externally extensible.
ConditionList	The ConditionList class represents a collection of all potential Conditions available for the user to add to Actions of their piece of interactive fiction. It should be externally extensible, although it is likely to be able to include all necessary conditions.
Interpreter	The Interpreter class takes game data (e.g. in the form of a file) as input and it outputs the actual game. In effect, it is the game client.
ElementGroup	The ElementGroup represents objects, characters or other things the player might encounter in the fictional world. It consists of at least one Element.
Element	The Element class is used as a building block for the ElementGroup. It allows the ElementGroup to act both as a single Element (e.g. a tree) and as a container of Elements (e.g. a chest which contains a precious gem).

Table 1 - Class breakdown

The current design has the strength that it can be very flexible. The flexibility of the separation between the development system and the interpreter is already explained in the 'Project scope' section.

Scenes can represent locations (e.g. a room), which contain elements with which the user interacts before moving to the next room. On the other hand, they can also be states; a room before an action takes place can be a different scene from the same room after an action takes place.

The flexibility of the design is also shown in the case of the Action class. Apart from the fact that it caters to both the text adventure and the interactive novel themes (through toggling of the visibility state), it also allows for more complex behaviour: actions can serve as simple state transitions, they can represent dialog choices when interacting with a character or they can be global actions whose conditions are evaluated in each frame (e.g. check if the player's hit points fall below 0).

The Resources can act as storage for literally any form of data the user wishes to keep track of.

Of course, being able to extend the ConditionList and the EffectList externally (this would be through actual programming) creates a cleaner and more modular design. Another reason for having these separate, is to help users edit them without fear of breaking the rest of the system, putting even more meaning in the fact that the project is open source.

3.4.5 Interpreter

The main loop of the interpreter from a high-level and platform-independent algorithmic point of view can be seen in Figure 3.

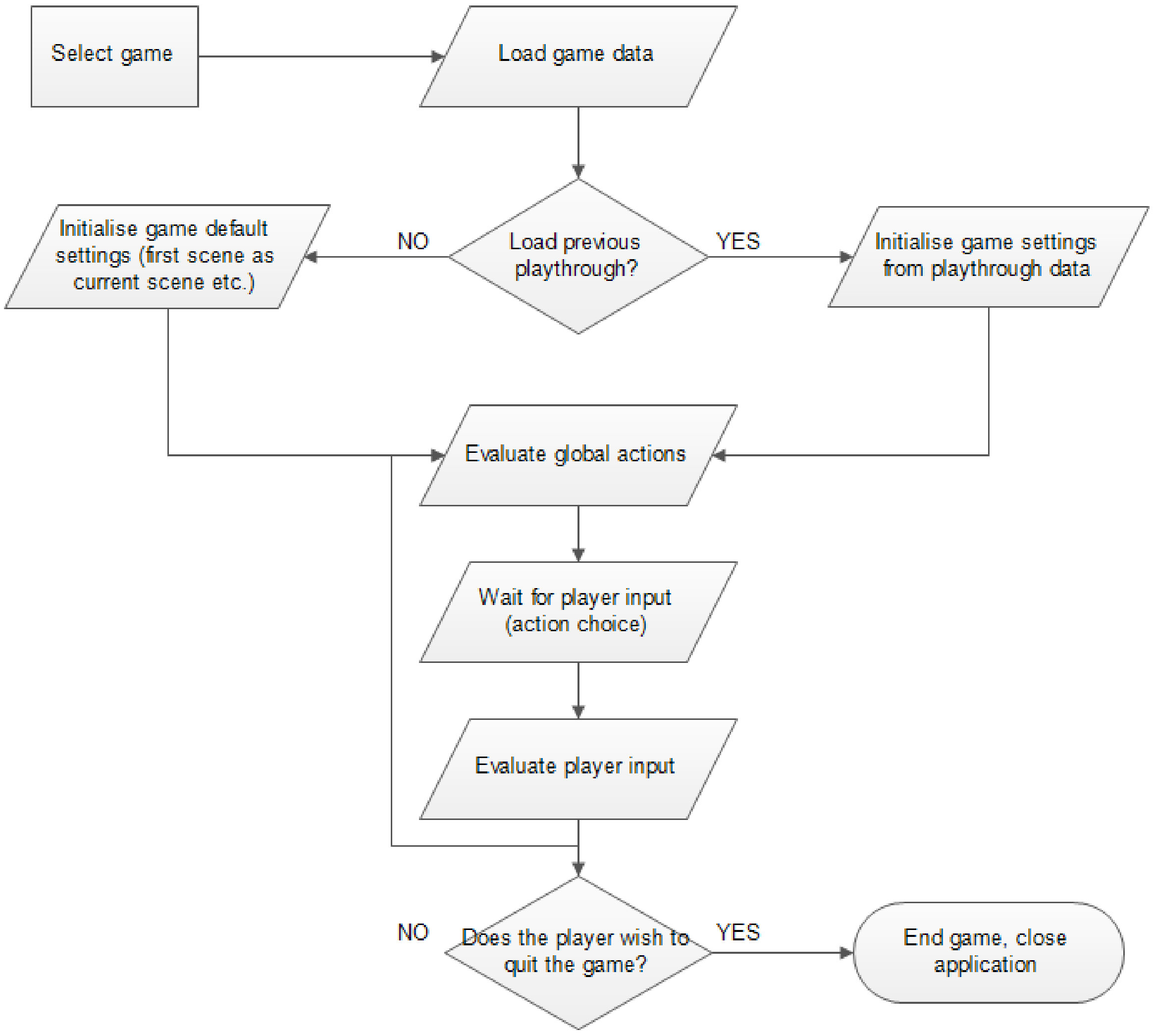
Select game
Load game data
Load previous playthrough?
NO
YES
Initialise game default settings (first scene as current scene etc.)
Initialise game settings from playthrough data
Evaluate global actions
Wait for player input (action choice)
Evaluate player input
NO
Does the player wish to quit the game?
YES
End game, close application

Figure 3 - Flow chart outlining the high-level logic of the interpreter

3.5 GRAPHICAL USER INTERFACE

At this stage there are no strict plans or design for the GUI. Having said that, the system will be implemented in such a way that the GUI can be easily changed without affecting the back-end. Especially since the requirements of the system include a user-friendly interface from which everything can be done, in the future (beyond the scope of the coursework deadline) prototyping and usability testing of different interfaces would be of top priority.

Currently, the general idea behind the GUI is that it must provide the entire functionality, allow for complex games and at the same time avoid getting cluttered. One way to evaluate the simplicity of the GUI is to use the Goals, Operators, Methods, Selection rules (GOMS) and State Transition Networks

(STN) (Alan Dix, Janet Finlay, Gregory D. Abowd, Russell Beale, 2003) which can help control state explosion and generally predict or automatically evaluate usability.

4 3. Discussion

The scope of the project, while consciously restricted compared to some very ambitious existing projects, still means a considerable amount of work. It includes the design of a powerful yet abstract representation of interactive fiction (data), the implementation of a development system which is GUI based, as well as a basic python interpreter; all these in an abstraction layer which allows to have interchangeable interpreters.

The main issues expected to arise are two. The first issue is about being able to implement all three aspects of the system in time; many of the implementation specifics will be picked up in a learn-as-you-go fashion, e.g. creating a GUI in Python. The second issue is about managing abstraction.

Abstracting the general story building blocks and the potential for different outputs from the same data may prove to be complicated. More specifically, having the possible Effects and Conditions as abstractions loaded from individual scriptable components -especially in a multi-platform situation- is not trivial and the use case diagram shows how much it adds to the complexity of the system. Therefore these requirements might be removed from the scope altogether in subsequent development iterations.

5 References

Alan Dix, Janet Finlay, Gregory D. Abowd, Russell Beale, 2003. *Human Computer Interaction.* 3rd ed. s.l.:Prentice Hall.

Andy Baio, Cooper McHatton, 2013. *Playfic Official Homepage.* [Online]
Available at: http://playfic.com/
[Accessed April 2013].

Campbell Wild, 2012. *ADRIFT Official Homepage.* [Online]
Available at: http://www.adrift.co/
[Accessed April 2013].

David Goodger, Guido van Rossum, 2009. *Python Docstring Conventions.* [Online]
Available at: http://www.python.org/dev/peps/pep-0257/
[Accessed April 2013].

Graham Nelson, 2010. *Inform 7 Official Homepage.* [Online]
Available at: http://inform7.com/
[Accessed April 2013].

Guido van Rossum, 2013. *Python Programming Language Official Website.* [Online]
Available at: http://www.python.org/
[Accessed April 2013].

High Energy Software, 2006. *TADS Official Homepage.* [Online]
Available at: http://www.tads.org/
[Accessed April 2013].

Riverbank Computing, 2013. *PyQT Introduction Page.* [Online]
Available at: http://www.riverbankcomputing.com/software/pyqt/intro
[Accessed April 2013].

Robert C. Martin, 2011. *Agile Software Development, Principles, Patterns and Practices: Principles, Patterns, and Practices.* 1 ed. s.l.:Perason.

Robin Dunn, Harri Pasanen, 2012. *wxPython Website.* [Online]
Available at: http://wxpython.org/index.php
[Accessed April 2013].

Sutton, P., 2012. *Twine Official Homepage.* [Online]
Available at: http://gimcrackd.com/etc/src/
[Accessed April 2013].

Warren, A., 2013. *Quest Official Homepage.* [Online]
Available at: http://www.textadventures.co.uk/quest/
[Accessed April 2013].