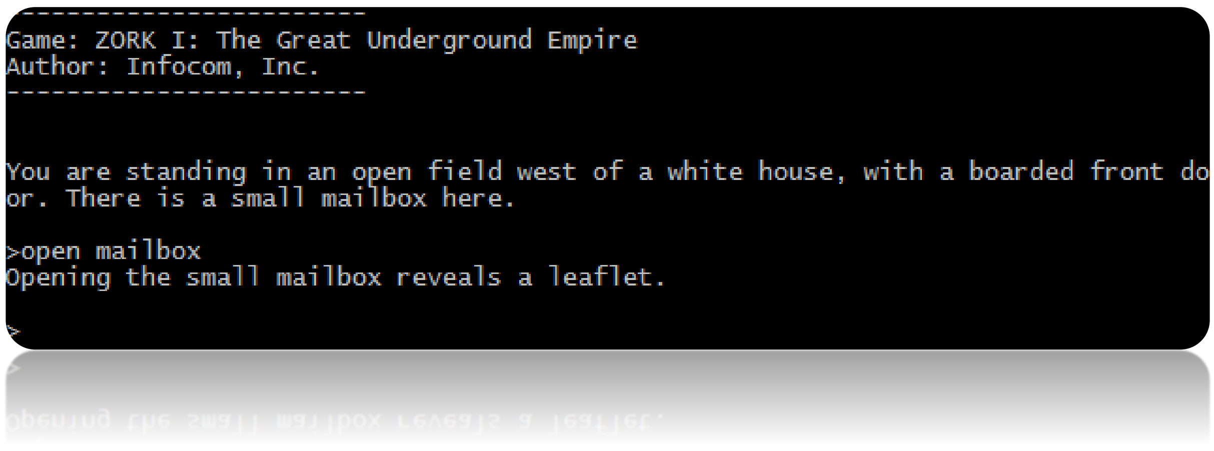
Alexandros Gouvatsos, Hibbert Ralph Animation, CDE - 16/05/2013

System report

TAINT: Text Adventure and Interactive Novel Toolkit



```
------------------------
Game: ZORK I: The Great Underground Empire
Author: Infocom, Inc.
------------------------


You are standing in an open field west of a white house, with a boarded front do
or. There is a small mailbox here.

>open mailbox
Opening the small mailbox reveals a leaflet.

>
```

1 Introduction

The Text Adventure and Interactive Novel Toolkit (TAINT) is a framework and collection of tools to facilitate the creation of text adventure and interactive novel games for non-technical users who have no programming knowledge as well as advanced users who enjoy a certain degree of power and customisability from their tools.

The motivation behind creating TAINT as well as an explanation of the terms text adventure and interactive novel is stated in the corresponding design document [1]. The project is broken down into the following sub-projects:

- The **core** of the system.
- A **data** representation for games.
- A Graphical User Interface (GUI) **editor** for creating games.
- An **interpreter** for playing back games.

The **core** captures the main game logic and is the back-end for the rest. The **data** representation of games is more of a finalised convention but also includes ways to store and load games. An **editor** is a tool for allowing end-users to create their own games without much technical knowledge. An **interpreter** allows end-users to load and play games created with WAINT.

The whole project can be found in a repository online [2]. The root project directory contains six directories and three files, as can be seen in Figure 1. The file *README.md* is a document which contains an introductory text for visitors to the repository. The *report* directory contains the system report document (the current document) and the design document for TAINT.

The files *make.bat* and *Makefile* correspond to the documentation generation system which is discussed in more detail in section 3.1. In the *source* directory are all the source files of the documentation and in the *build* directory are the generated results after compiling the documentation.

The *dev_setups* directory contains scripts and programs which might be useful to developers who want to setup their environment in order to contribute to the TAINT project.

Finally, the *src* directory contains the actual source code for the TAINT framework and its related tools as seen in Figure 2. More specifically, the *core* directory contains the source code for the core or "heart" of the framework. The *dataParser* directory contains the source code of a sub-system for loading game data. The *editor* directory contains the source code for a GUI tool for creating TAINT games. The *interpreter.py* file and the *webapp* directory contain the source code for a local disk and a web application game interpreter, respectively. Each of these modules is explained in more detail in section 3.

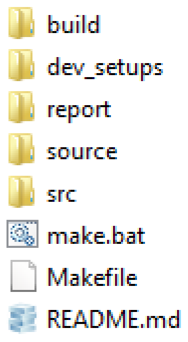


Figure 1 - Root project directory as seen in the Windows Explorer file manager.

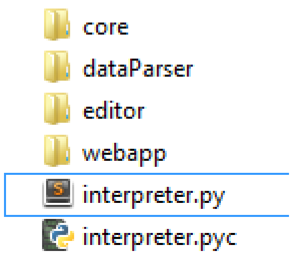


Figure 2 - The "src" directory of the project as seen in the Windows Explorer file manager.

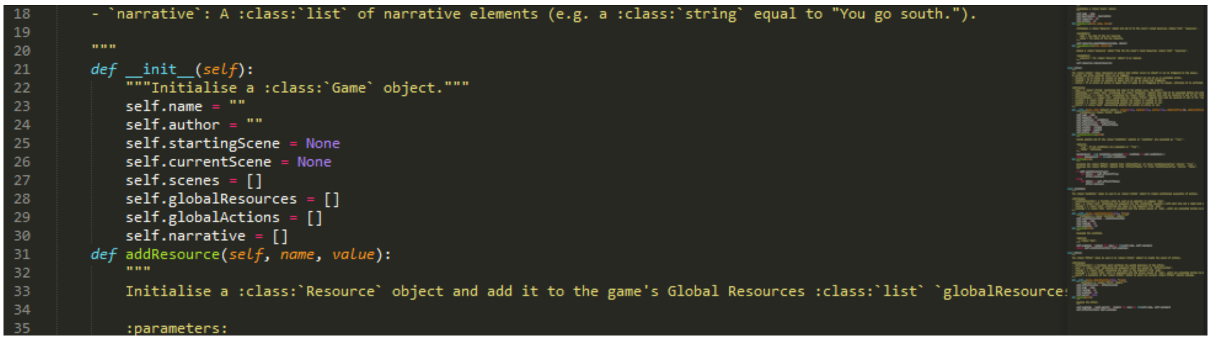
2 Produced System

2.1 Results

In summary, the project consists of a finalised back-end framework called the **core**, an **editor** which allows for the creation of games through a GUI, and two **interpreters**, one of which is web-based. The web-based interpreter is in the form of a website, meaning that games created with TAINT are platform independent – as long as the system has a web browser.

More specifically, the project so far has resulted in a prototype yet completely functional product. Apart from a system (in the sense of an assortment of software), we have also developed an approach to designing text adventures and interactive novels. This approach is called the TAINT game model and is explained in section 3.2.1. The conventions for a game's data representation are described in section 3.2.2.

The project consists of a finalised **core** framework, which captures the main logic and the game model. A user comfortable with programming could already use this framework as an API to directly create a game. This core of the system is further explained in section 3.3.1.



```
    - `narrative`: A :class:`list` of narrative elements (e.g. a :class:`string` equal to "You go south.").

    """
    def __init__(self):
        """Initialise a :class:`Game` object."""
        self.name = ""
        self.author = ""
        self.startingScene = None
        self.currentScene = None
        self.scenes = []
        self.globalResources = []
        self.globalActions = []
        self.narrative = []
    def addResource(self, name, value):
        """
        Initialise a :class:`Resource` object and add it to the game's Global Resources :class:`list` `globalResource

        :parameters:
```

Figure 3 - Screenshot of core module source code in a text editor.

However, since the goal is for a user friendly solution which does not require technical knowledge such as programming, there is also a GUI **editor** (Figure 4) which has been developed. The scope of a user friendly front-end is inherently huge. Therefore, it is especially this part of the system that could benefit the most from continued development. Nonetheless, the current **editor** is completely functional and includes all the basic needs in creating a game graphically. This aspect of the system is further analysed in section 3.3.2.

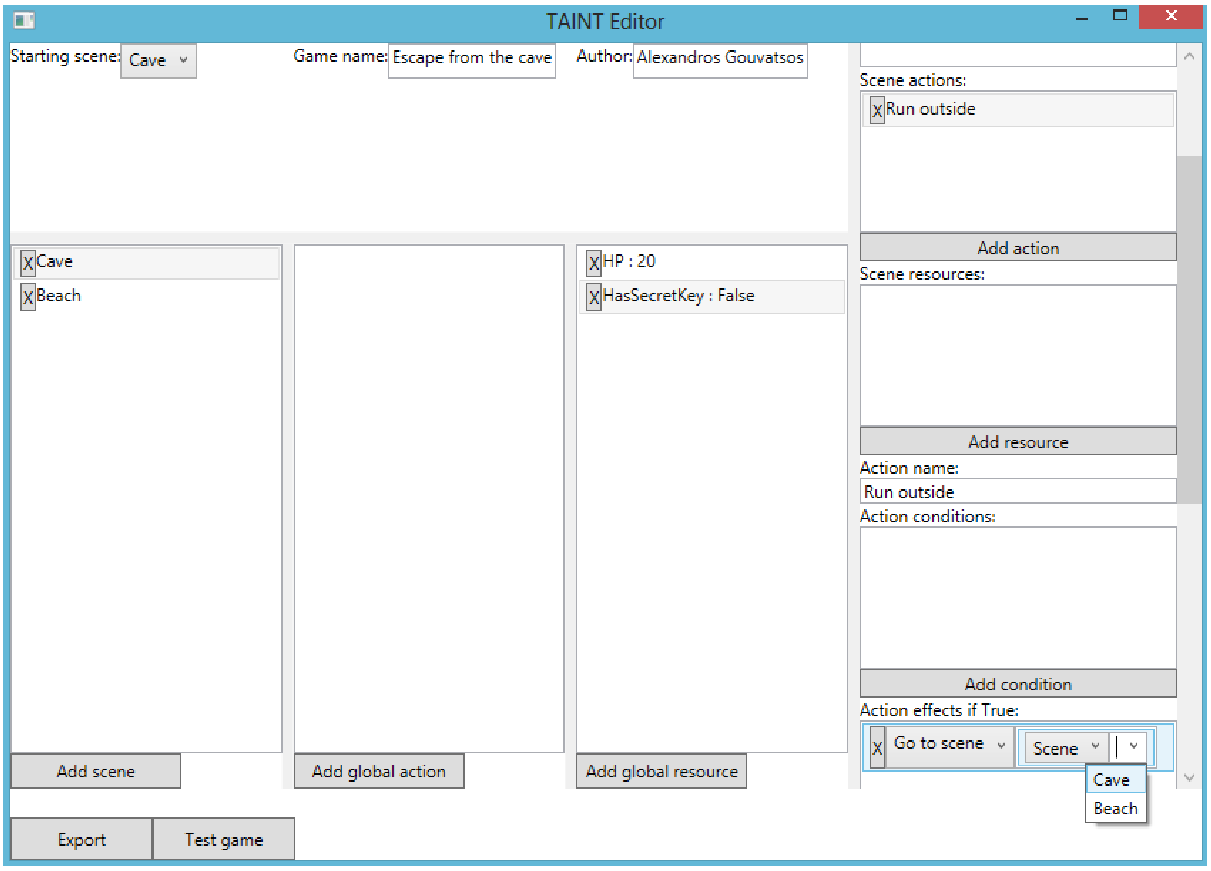
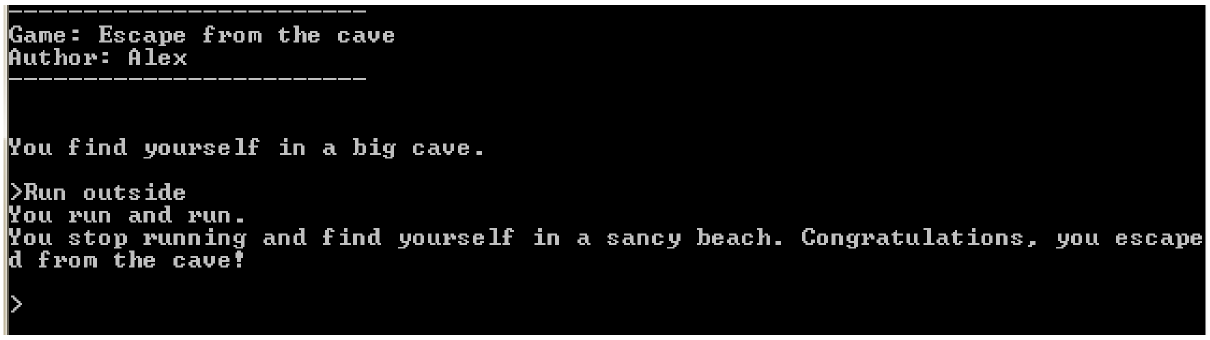


Figure 4 - Screenshot of GUI editor for creating TAINT games.

Finally, two **interpreters** are included in this project. The first (Figure 5) is a simple interpreter which can be run locally (and is in fact used within the **editor** for game testing). The second (Figure 6) is a web application which is published online and runs TAINT games in a web browser. Moreover, it is a good way of showing the potential of having an online hub of games, where an active community can share and play text adventures and interactive novels. The interpreter implementation is broken down in section 3.3.3.



```
------------------------
Game: Escape from the cave
Author: Alex
------------------------


You find yourself in a big cave.

>Run outside
You run and run.
You stop running and find yourself in a sancy beach. Congratulations, you escape
d from the cave!

>
```

Figure 5 - Screenshot of a game as seen in the interpreter which runs locally in a terminal.

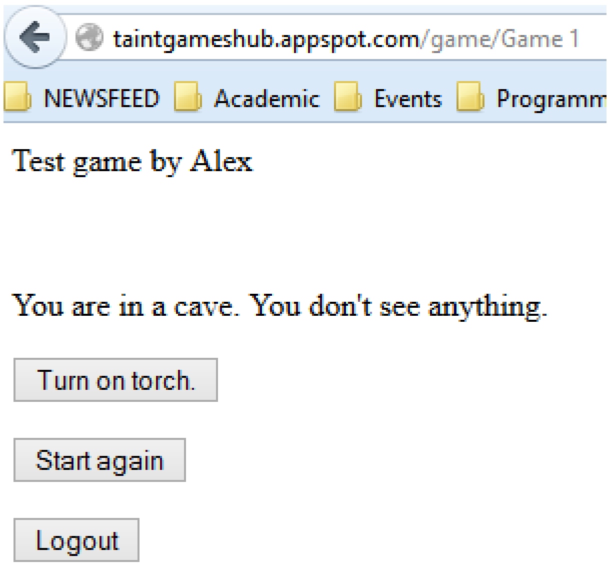

taintgameshub.appspot.com/game/Game 1
NEWSFEED
Academic
Events
Programm
Test game by Alex
You are in a cave. You don't see anything.
Turn on torch.
Start again
Logout

Figure 6 - Screenshot of a game run in the web interpreter as seen in a web browser.

2.2 System Use

As mentioned in section 1, the whole project is open source and is available online [2]. The code base is under the MIT license.

Documentation for users and for developers is also available online [3]. There is a prototype of a hub with a selection of test games created with TAINT, which can also be found online [4].

Alternatively, to test a game one can navigate to the */src/* directory and run the *interpreter.py*, which requires Python to be installed on the system. There is a default test game called *game.xml* which will be loaded by this interpreter.

The prototype editor for creating games is also published online [5] and it includes an installation suite which ensures all the prerequisites are also downloaded and installed, as well as automatic updates.

The editor, unlike the rest of the system, only runs on Windows. The rest of the system requires an installation of Python, while the web interpreter only requires a web browser.

To facilitate access to the above online sources, they are presented again as URLs in the following list:

- Online repository: https://github.com/glorion13/TextEngine
- User and developer documentation: http://gouvatsos.com/Media/Default/Apps/TextEngineEditor/docs/build/html/index.html
- Games hub: http://taintgameshub.appspot.com/
- Editor installation: http://www.gouvatsos.com/Media/Default/Apps/TextEngineEditor/publish.htm

It is recommended that in order to assess the system one visits these online sources to view the source code, the documentation and to play some games. It is also recommended that instead of inspecting the code directly in a text editor, one inspects it on the documentation website.

3 IMPLEMENTATION

An overview of the original design and background research at the beginning of the project is available in the design document [1].

When comparing the final product to the initial design, it becomes apparent that there are some deviations. This is due to the iterative nature of software development and more specifically to the semi-agile approach that was followed throughout the project. Having said that, many aspects of the design were kept all the way to the current state of the project, especially in terms of the overall logic and the project scope.

Due to the fact that many different system aspects were visited through the development cycle, it was very important to go back to the original design document and revisit the scope, before attempting to implement some requirement. These re-evaluations were done quite often as the development process consisted of short iterations. This ensured that only requirements with a high value were implemented.

Moreover, with each new requirement which had to be satisfied the code was often refactored to maintain a clean codebase and a logical breakdown. After the initial iterations, the main modules were already set which made this refactoring process even easier, as modules can be changed without fear of breaking other functionality.

3.1 TOOLS, TECHNOLOGIES AND CODING STANDARDS

For version control Git was the preferred system and Github [6] the chosen provider. Most parts of the system are programmed in Python, apart from the editor which is created using C# and WPF under the .NET 4.5 framework. Moreover, the code is tested to work under all versions of Python above 2.5.

The web interpreter is created using Google's AppEngine [7] and Python 2.7. The reason for choosing this is that the **core** of the system (which is in Python) remains untouched and only the **interpreter** had to be modified to fit a web format. Moreover, it provides other functionality such as databases and user login out of the box.

In general, it is important for the games created with TAINT to be accessible in as many platforms as possible and therefore Python and a web application allow for exactly that. On the other hand, the editor does not necessarily need to be multi-platform (although it is not something undesirable) and hence C# and WPF were chosen as acceptable technologies.

For the creation of some of the class diagrams, PyNSource [8] and the default class diagram editor of Microsoft's Visual Studio are used for the Python and C# components respectively. Both of these tools allow for parsing of the source code directly, which is very useful when comparing class diagrams at implementation time with class diagrams at design time.

There are no specific coding standards that were followed, however the code base is constant in style. There is a different style used for the Python aspects and a different style used for the C# aspects, in order to remain consistent with popular conventions. More specifically, an outline of the coding style followed in Python and in C# is presented in Table 1.

	Python	**C#**
Functions / Methods	camelCase	CamelCase
Attributes / Fields-Properties	camelCase	- Private: camelCase - Public: CamelCase

Classes	CamelCase	CamelCase
Code clarity and comments	- Descriptive naming. - Documentation of each class and function within itself using reStructuredText syntax.	- Descriptive naming. - Rigorous consistency with chosen design and architectural patterns. - Code separated using #region tags.

Table 1 - Coding styles for Python and C# components of the code base.

The documentation is created using Sphinx [9]. Sphinx uses the reStructuredText [10] markup syntax to generate code documentation directly from the source code. Since the Python aspects of the system did not follow a specific documented design pattern, it is useful to include the functionality of each class and function in itself. Since this is done anyway for its own merit, Sphinx has the advantage that it can directly then create documentation.

3.2 TAINT GAMES

Traditional text adventure games, where the player inputs commands by typing (e.g. pick up leaflet) follow an object-centric approach. What this means is that the game model consists of objects in the game environment and each object responds to specific actions such as “take” or “pick up”. A game designer then thinks in terms of what objects exist in a scene and how a player will interact with them.

This model is suitable for creating a command-driven game since a command parser only requires a specific syntax (e.g. action [objects]) and access to a list of objects. It is also more suitable for an experience of a “living”, more interactive world as the player essentially interacts with an environment that is laid out piece by piece by the designer. However, it is not suitable for an interactive novel where the game offers a multiple choice of possible actions and the player gets to choose one. Quite often the designer is required to create invisible objects just to define some additional action that they want their player to be able to perform.

The TAINT game model follows a different approach, which can be called action-centric. In an action-centric approach, the game designer does not need to construct a world and think about the objects in the environment so explicitly. The design process follows a more intuitive “what happens next?” approach, by requiring the designer to think about what actions are possible in each scene.

Especially in the genre of text adventure and interactive novel games, the game designer is more of a writer rather than designer. Therefore, an assumption is made that this action-centric approach would be more intuitive than that of a world-building approach. Furthermore, using an action-centric approach one can successfully generate interactive novels without much hassle. It has also been extended to support a list of keywords for each action, allowing interactions with a traditional command-driven interface. The downside is that in a command-driven game such as text adventures are, the different actions for each object are stored in the scene, which can result in bloating of a scene’s model if there are too many objects. More about how to tackle this issue is discussed in section 4.

Overall, the TAINT game model provides a very simple yet extremely flexible and powerful way to reason and design games. The TAINT user documentation [3] is referenced to provide a breakdown of the whole model in sections 3.2.1 and the game data representation in section 3.2.2.

3.2.1 Game model

3.2.1.1 Main building blocks

TAINT has the following core building blocks:

- **Scene**: Scenes represent a location or a state in the game world.
- **Resource**: Resources represent anything that needs to be tracked! The player's HP, Mana, the time or date in the game world, whether or not the player has an item etc. One example of a resource can have a name of "Hitpoints" and a value of 10.
- **Action**: Actions represent happenings in the game world, whether it is something that occurs by itself (passive) or something that is triggered by the player (active). An action consists of conditions and effects. One list of effects are triggered if the conditions are true and another if the conditions are false.
- **Condition**: Conditions are ways to give further branching to your game, by checking if certain things are true or not (e.g. does the player have the required item to open the door?). Conditions have a left-hand side and a right-hand side argument and then an operator which compares them. For example, the left-hand side argument might be a Resource called "HP" and the right-hand side argument might be the number 0. Using a Condition, it is possible to check whether the "HP" has dropped below 0 or not.
- **Effect**: Effects are the resolutions of an Action. Basically, effects can be things like going to a new Scene, changing the value of a Resource or simply giving some additional information to the player.
- **Narrative**: The narrative is the story which is presented to the player in the form of text.

These building blocks are then combined in the game structure in order to create interactive experiences. Figure 7 displays a high level visualisation of the game model.

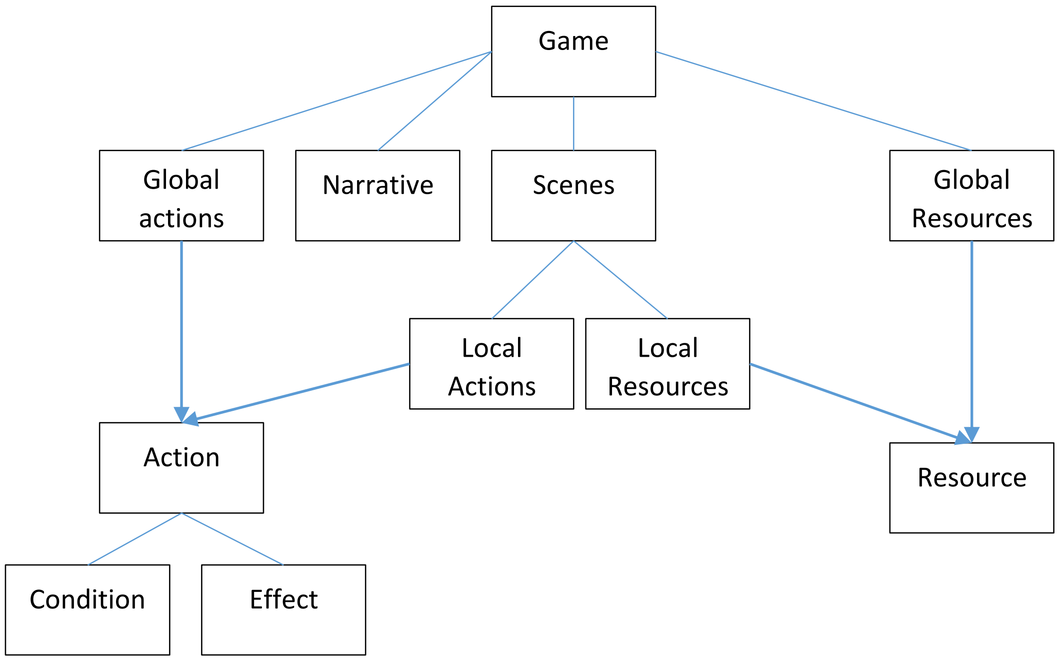


Figure 7 – A high-level view of the TAINT game model.

3.2.1.2 Game structure

A game contains a list of scenes, a list of global resources and a list of global actions. Global resources are resources which are accessible from all scenes of the game. Similarly, Global actions are actions which are accessible from all scenes. A game also has a starting scene which is the scene in which the game begins.

Each scene of the game, contains a list of actions and resources. Those are also called local actions and local resources, because unlike their global counterparts, they are only accessible within the scene to which they belong.

For example, a global action could be an action that tells the player what is in their inventory and it is global because the player should be able to check his inventory at all times, no matter where he is. On the other hand, a local action (an action within a scene) could be an action which opens a door in that specific scene. It is clear that there is no reason for this action to be available in any other scene of the game.

Actions have three switches which can be toggled between on and off:

- The **enabled** switch, sets an action as accessible or entirely not accessible (as if it doesn't exist).
- The **active** switch sets the action as passive or active. Active actions can only be triggered by the player (e.g. the player decides to open a door) while passive actions are evaluated in every turn (e.g. always check if the player's "HP" falls below 0 and if it does let them know they are dead).
- The **visible** switch makes the action visible to the player. The player can always see a list of all visible actions, like a multiple choice list. An example case where a scene should not be visible is in command-driven games. Another case would be to include Easter eggs.

3.2.1.3 Types

There are 4 types (programmatically speaking) in a TAINT game:

- **Number**: a Number can be an integer like 10 or a float like 0.75.
- **Text**: text is used for any data in textual form.
- **Boolean**: a Boolean can be either *True* or *False* and can be used for checks like whether or not something exists or whether or not the player has done something.
- **Resource**: a Resource is a type by itself as one can explicitly compare e.g. a Resource with a Boolean. A Resource has the interesting property that its value can be any of the three types Number, Text or Boolean.

3.2.2 Game data representation

Games built using the TAINT model are stored as XML. The fact that the data is open has several advantages. Firstly, one can manually inspect a TAINT game directly in XML which is a human-readable format. Secondly, open data means that one can develop their own tools such as an editor or an interpreter, as long as they adhere to the TAINT structure.

Table 2 contains an explanation of the different XML tags and how they are used to store a game. For information and examples are available in the user documentation [3].

XML	DESCRIPTION
<GAME>	The main tag which contains all the others is the <Game> tag. It contains the <GameName> the <Author> tags which contain the name of the game (e.g. Escape from the mines) and the name of the author(s). It also contains the <StartingScene> tag which sets the starting scene by name (e.g. Cave 1). It then contains the <GlobalResources>, <GlobalActions> and <Scenes> tags. The <GlobalResources> contains a number of <Resource> elements which represent

	the global resources of the game. The <GlobalActions> block contains a number of <Action> elements which represent the global actions of the game. Finally the <Scenes> block contains a block of <Scene> elements.
<RESOURCE>	Contains: • A <Name> element which contains the resource's name (e.g. HP). • A <Type> element which sets the type of the resource (e.g. Number). • A <Value> element which represents the value of said resource (e.g. 20).
<ACTION>	Contains: • A <Name> element which is the name of the action (e.g. pick up rock). • The <Visible>, <Enabled> and <Active> elements, each of which can be set to be either True or False. • A <Conditions> block which contains a number of <Condition> elements. • An <EffectsIfTrue> block which contains a number of <Effect> elements. • An <EffectsIfFalse> block which contains a number of <Effect> elements.
<CONDITION>	Contains: • The <ConditionFunction> element which needs to be set to one of the available condition names (e.g. equals). • The <LeftHandSide> and <RightHandSide> elements, which requires an attribute Type to set the type and then a value of that type to compare.
<EFFECT>	Contains: • The <EffectFunction> element which needs to be set to one of the available effect names (e.g. Tell player). • The <args> block which contains a number of <arg> elements, representing the arguments passed to the effect function.
<SCENE>	Contains: • A <Name> tag to set the scene's name (e.g. Cave). • A <Description> tag to set the scene's narrative when the player enters. • A <Resources> block which contains a number of <Resource> elements. • An <Actions> block which contains a number of <Action> elements.

Table 2 - Breakdown of individual XML elements for storing game data.

3.3 MODULES

As explained in section 1 of this report, TAINT consists of three modules. The first module, which is called **core**, is the module on which the others depend and contains the logic of the game model as interfaces for other components. The second module is called the **editor** and is a GUI program for creating games. The third module is called the **interpreter** and it represents a program which can playback game data from an XML file.

Linked to the core module and auxiliary to the rest (hence not included in the list of three modules) is the **dataParser** sub-module which merely deals with loading XML game data into memory. It is useful mostly for the **interpreter** but it would also be useful for loading project files into the **editor**.

Structuring the system in a modular way has several advantages. Perhaps the most important advantage is that once one module is in place, the others can be edited or even completely rewritten without affecting it. This decreases the risk of error production through refactoring. It also draws a clear line when it comes to separation of concerns.

Another advantage is that as the project increases in scale and more developers are added to the team, it is easy to assign workload. Furthermore, a new team member would not need to have explicit knowledge of the entire code base in order to contribute to an individual model, reducing time needed to get up to speed.

Especially since the power of the system is for someone to easily extend it or make use of the fact that the data is open to completely redesign their own editors and interpreters, the reasons for this design decision become even more apparent.

Finally, traditional reasons for a modular design are reinforced by the fact that the system is object-oriented. In the following sections 3.3.1, 3.3.2 and 3.3.3 the individual modules and their sub-modules are explained, showcasing the loose coupling and high cohesion achieved by the design.

3.3.1 Core

The core of the system is programmed entirely in Python and works on all versions later than 2.5. It captures the main game logic and allows evaluation of attributes such as effect functions and condition function in runtime. This means that an effect function can hold a reference to a Resource that does not exist yet but will exist when the function is evaluated, without raising an exception. This is performed through the use of *lambda* functions.

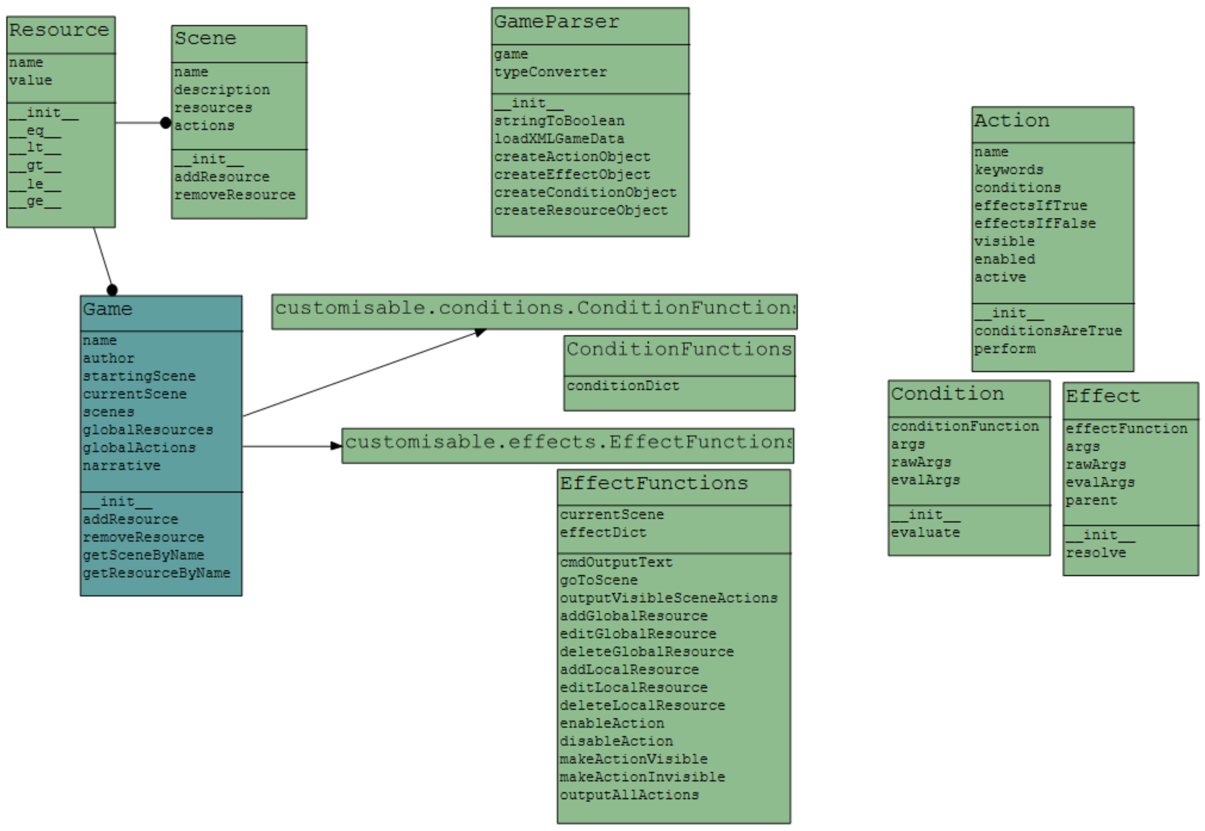


Figure 8 - Core module class diagram.

The class diagram in Figure 8 shows how the game model is captured through an object-oriented design and makes clear what the interdependencies are. Even though Python is not a traditionally object-oriented language, this design allowed for:

1. A direct translation of the game logic to code.
2. A direct interface between the core modules and the other modules (especially the editor which uses C#).

Moreover, TAINT makes use of the scripted components design pattern [11] in order to allow advanced users to extend the engine with their own condition functions and effect functions. These are captured under the *customisable* sub-module.

Having said that the design is object-oriented, these sub-modules which are directly exposed to the end user have a more functional layout, making it easier for these users to prototype and experiment without being exposed to any other parts of the system. More details on exactly how these scripted components work is available online [12].

Once the **core** is set up, the other modules (e.g. the **editor**) do not access it directly but through certain *interfaces*. The reason for this is again the fundamental concept of loose coupling. Finally, the **editor** and the **interpreter** use the core system without the need for the developer to rewrite or understand it.

3.3.2 Editor

The **editor** is developed in C# using the Windows Presentation Foundation (WPF) on top of the .NET 4.5 framework. The design pattern used is the Model View ViewModel (MVVM) which allows for a clear separation of concerns between the design model (Model), what the user sees (View) and the interactive components and controls of the GUI (ViewModel). Furthermore, the module is again built following object-oriented concepts. Figure 9 shows a high-level class diagram of how the individual components of the editor's ViewModel are linked to each other. More detailed class diagrams are available in the Appendix (section 6). The graphs and diagrams are also available in the */report/graphs/* directory of the root directory of the project.

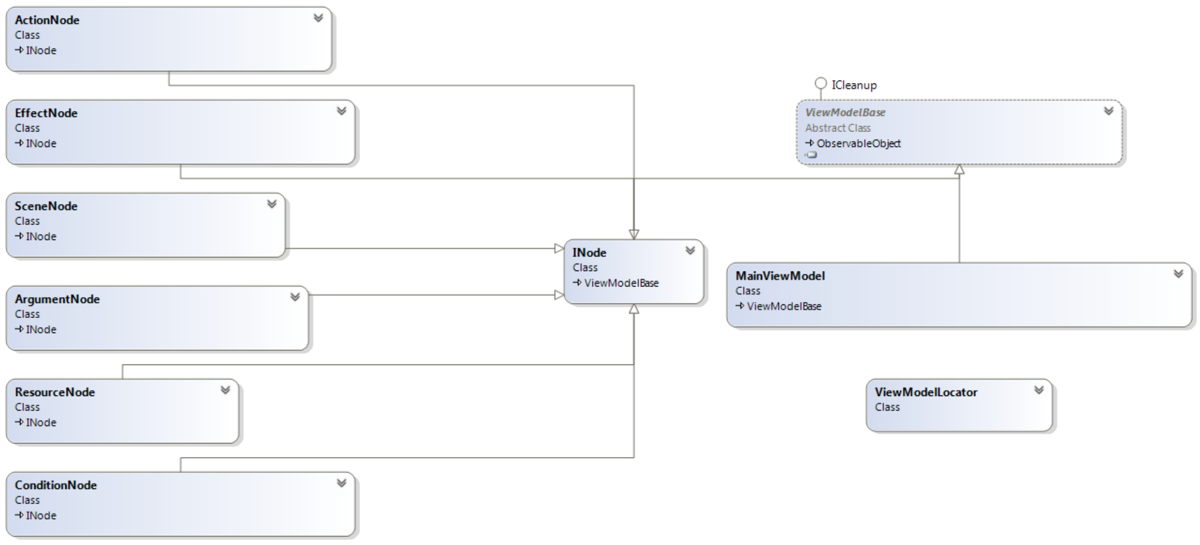


Figure 9 - Editor module high-level class diagram.

The direct interface with the **core** module means that the editor can parse the customisable scripted components in order to provide visual aid to the user creating a game. For example, when the user adds a new effect to an action, the editor can read through the Python *customisable* components and offer all the possibilities through a drop-down box (Figure 10).

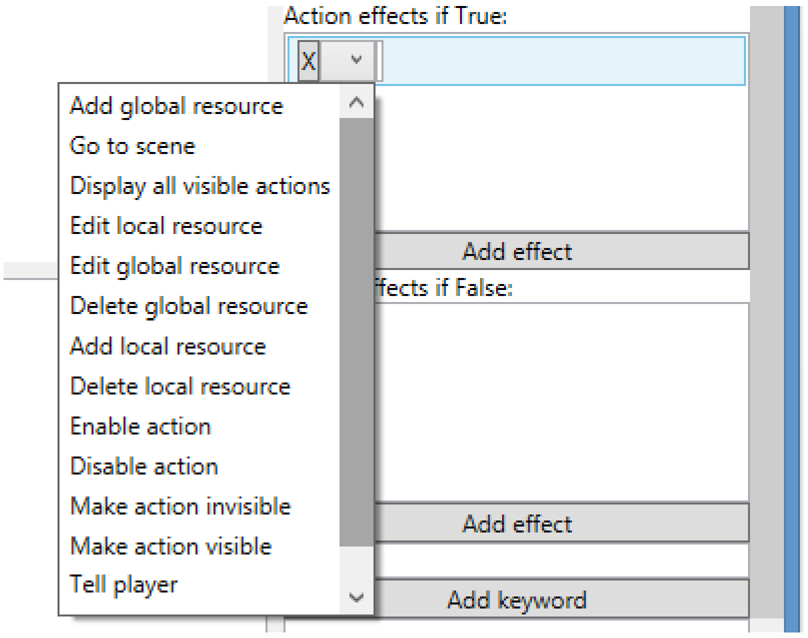


Figure 10 - Effect functions loaded directly from Python scripted components.

The visual aid that the editor can provide through this interface with the **core** module can also be extended for even more complex affordances. For example, once the user selects an effect function from the drop down list, the editor is able to read the potential parameters of that function (Figure 11). Subsequently, it can also offer specific options for the values of those parameters by combining the Python types with the types of the available ViewModel objects (Figure 12).

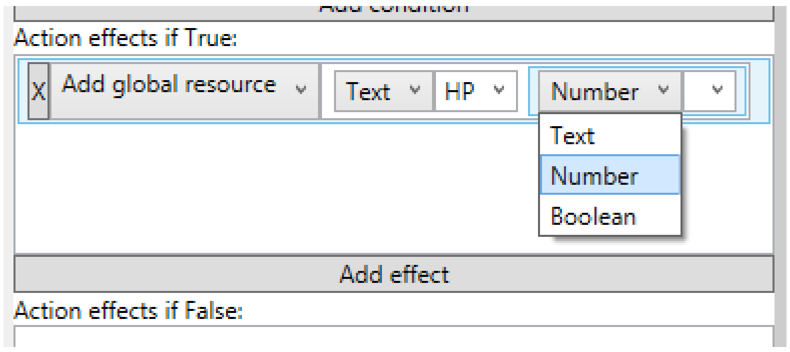


Figure 11 - "Add global resource" effect function.

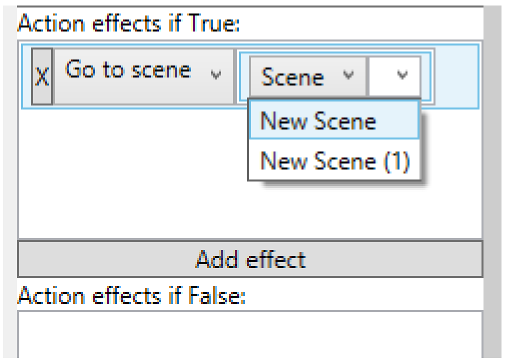


Figure 12 - "Go to scene" effect function.

Finally, the **editor** can provide direct game testing capabilities by calling the available Python interpreter, as long as Python is installed and referenced in the PATH environment variable of the user's operating system.

3.3.3 Interpreter

The interpreter module provides the functionality of playing back games. Because it is the very end-user (the gamer) who will be dealing with this, it is extremely important to be as accessible as possible. As such, apart from a simple implementation in Python (which is useful by itself as it can be used stand-alone and it is also used by the game testing functionality of the editor), the potential of TAINT games is shown through the implementation of a web application interpreter.

The idea is that an online hub can exist where users login with their details and access games from their game library and play them anywhere; as long as their device has a web browser. At the same time, in order to minimise code rewriting and allow for this clean separation in our design, using the **core** module is important. That is why Google's AppEngine is chosen as the underlying technology.

Overall, an interpreter needs to have access to the **core** module as well as the **dataParser** sub module. Even though for a web platform the workflow is somehow different, the general idea is as follows:

1. Select a game.
2. Load game data.
3. If loading previous playthrough go to step 3.a. If not, go to step 3.b.
    a. Initialise game settings from stored playthrough data. Set current scene to the stored current scene.
    b. Initialise default game settings. Set first scene as current scene.
4. Evaluate passive actions.
5. Wait for player action.
6. Perform player action (global or local). Go to step 5.

4 Discussion

Overall, TAINT is a finished engine which can be used to create and play text adventure and interactive novel games. Its components have been published and are available online. It provides both a back-end and a front-end for content generation and content consumption. Most aspects of the system are platform independent.

Apart from the end product, it is also a project which -through continued development- can become an actual commercial or 'mainstream' product. It facilitates outside development and contributions (which is the idea of open source) as its foundations are built from the beginning with that in mind.

During the initial design process, a list of functional and non-functional requirements was composed [1]. These requirements are now revisited in order to examine design changes through the development cycle as well as future work possibilities.

The action-centric approach that was followed is prominent even early on in the design process. The strong modular approach is evident as well. Most differences are in the actual implementation of the core system.

More specifically, in the original design the game model (and therefore the **core** module) included the idea of an inventory. While this idea does not explicitly exist in the current design, it is supported through the abstraction of Resources. Therefore, a game designer can create an inventory just like before, even though it is abstracted in the form of a list of Resources.

In general, all functional and non-functional requirements that were originally gathered are satisfied, except for the non-functional requirements V, VI and VIII. These are the following:

> 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.
>
> 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.

The requirements V and VI are still definitely within the scope of the project, however where moved on to later iterations in order to satisfy other requirements first. One way to achieve this functionality is by storing a list of the actions the player performs. Then when loading a game, the **interpreter** will repeat the list of actions, therefore reaching the exact same state as before.

The requirement VIII is potentially very complex and does not add any value to the current action-centric game model. On the other hand, if the current game model is extended to encompass some object-centric attributes, this requirement will be revisited and re-evaluated.

Apart from the above requirements, there are many paths future work can follow. Firstly, extending the game model to support object-centric ideas can help make TAINT more suitable for games which are designed as an interactive world rather than an interactive story. Secondly, the online hub can be improved from a prototype to an actual online portal, both in terms of the functionality as well as visually. This would include a way for users to upload their content to the server. Thirdly, user experience (UX) evaluations can be performed either on individual aspects of the system or to the entire project. This would help with the design and refactoring of the editor and it would also provide evidence as to whether or not the TAINT game model is useful and intuitive to a non-technical user. Finally, the editor itself is a direct interface between the user and the system and could benefit from future work.

Ideas for improving the editor include UX experiments as mentioned above, as well as making it multi-platform like the rest of the system. Adding Python debugging for the scripted components would help guide advanced users who are writing their own condition and effect functions. Being able to reference variables such as a Resource's name or value within a Text type would allow for more powerful and complex mechanics. Improving the GUI to include mind maps or other visual tools would help leverage the editor from a development tool to an actual game design tool, all in one.

5 References

[1] Alexandros Gouvatsos, “TAINT: Text Adventure and Interactive Novel Toolkit (Design Document),” 2013.

[2] Alexandros Gouvatsos, “TAINT Github repository,” 2013. [Online]. Available: https://www.github.com/glorion13/TextEngine. [Accessed 15 May 2013].

[3] Alexandros Gouvatsos, “TAINT documentation,” 2013. [Online]. Available: http://gouvatsos.com/Media/Default/Apps/TextEngineEditor/docs/build/html/index.html. [Accessed 15 May 2013].

[4] Alexandros Gouvatsos, “TAINT games hub,” 2013. [Online]. Available: http://taintgameshub.appspot.com/. [Accessed 15 May 2013].

[5] Alexandros Gouvatsos, “TAINT editor installation page,” 2013. [Online]. Available: http://www.gouvatsos.com/Media/Default/Apps/TextEngineEditor/publish.htm. [Accessed 15 May 2013].

[6] “Github main page,” Github.com, 2013. [Online]. Available: https://github.com/. [Accessed 15 May 2013].

[7] Google, “AppEngine main page,” 2013. [Online]. Available: https://developers.google.com/appengine/. [Accessed 15 May 2013].

[8] “PyNSource main page,” 2013. [Online]. Available: http://code.google.com/p/pynsource/downloads/list. [Accessed 15 May 2013].

[9] “Sphinx-doc main page,” 2013. [Online]. Available: http://www.sphinx-doc.org/. [Accessed 15 May 2013].

[10] “reStructuredText documentation,” 2013. [Online]. Available: http://docutils.sourceforge.net/rst.html. [Accessed 15 May 2013].

6 APPENDIX

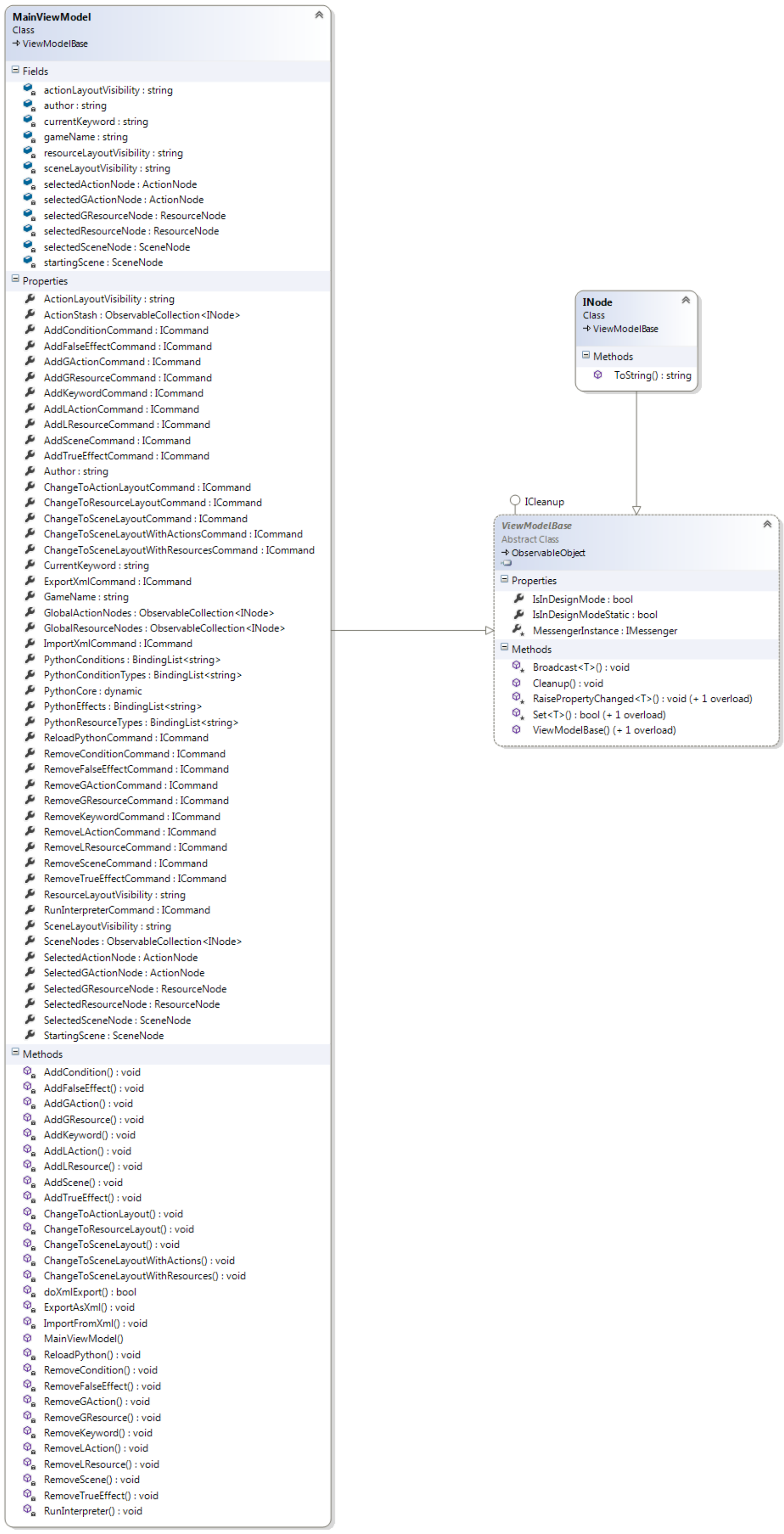

MainViewModel
Class
→ ViewModelBase
Fields
actionLayoutVisibility : string
author : string
currentKeyword : string
gameName : string
resourceLayoutVisibility : string
sceneLayoutVisibility : string
selectedActionNode : ActionNode
selectedGActionNode : ActionNode
selectedGResourceNode : ResourceNode
selectedResourceNode : ResourceNode
selectedSceneNode : SceneNode
startingScene : SceneNode
Properties
ActionLayoutVisibility : string
ActionStash : ObservableCollection<INode>
AddConditionCommand : ICommand
AddFalseEffectCommand : ICommand
AddGActionCommand : ICommand
AddGResourceCommand : ICommand
AddKeywordCommand : ICommand
AddLActionCommand : ICommand
AddLResourceCommand : ICommand
AddSceneCommand : ICommand
AddTrueEffectCommand : ICommand
Author : string
ChangeToActionLayoutCommand : ICommand
ChangeToResourceLayoutCommand : ICommand
ChangeToSceneLayoutCommand : ICommand
ChangeToSceneLayoutWithActionsCommand : ICommand
ChangeToSceneLayoutWithResourcesCommand : ICommand
CurrentKeyword : string
ExportXmlCommand : ICommand
GameName : string
GlobalActionNodes : ObservableCollection<INode>
GlobalResourceNodes : ObservableCollection<INode>
ImportXmlCommand : ICommand
PythonConditions : BindingList<string>
PythonConditionTypes : BindingList<string>
PythonCore : dynamic
PythonEffects : BindingList<string>
PythonResourceTypes : BindingList<string>
ReloadPythonCommand : ICommand
RemoveConditionCommand : ICommand
RemoveFalseEffectCommand : ICommand
RemoveGActionCommand : ICommand
RemoveGResourceCommand : ICommand
RemoveKeywordCommand : ICommand
RemoveLActionCommand : ICommand
RemoveLResourceCommand : ICommand
RemoveSceneCommand : ICommand
RemoveTrueEffectCommand : ICommand
ResourceLayoutVisibility : string
RunInterpreterCommand : ICommand
SceneLayoutVisibility : string
SceneNodes : ObservableCollection<INode>
SelectedActionNode : ActionNode
SelectedGActionNode : ActionNode
SelectedGResourceNode : ResourceNode
SelectedResourceNode : ResourceNode
SelectedSceneNode : SceneNode
StartingScene : SceneNode
Methods
AddCondition() : void
AddFalseEffect() : void
AddGAction() : void
AddGResource() : void
AddKeyword() : void
AddLAction() : void
AddLResource() : void
AddScene() : void
AddTrueEffect() : void
ChangeToActionLayout() : void
ChangeToResourceLayout() : void
ChangeToSceneLayout() : void
ChangeToSceneLayoutWithActions() : void
ChangeToSceneLayoutWithResources() : void
doXmlExport() : bool
ExportAsXml() : void
ImportFromXml() : void
MainViewModel()
ReloadPython() : void
RemoveCondition() : void
RemoveFalseEffect() : void
RemoveGAction() : void
RemoveGResource() : void
RemoveKeyword() : void
RemoveLAction() : void
RemoveLResource() : void
RemoveScene() : void
RemoveTrueEffect() : void
RunInterpreter() : void
INode
Class
→ ViewModelBase
Methods
ToString() : string
ICleanup
ViewModelBase
Abstract Class
→ ObservableObject
Properties
IsInDesignMode : bool
IsInDesignModeStatic : bool
MessengerInstance : IMessenger
Methods
Broadcast<T>() : void
Cleanup() : void
RaisePropertyChanged<T>() : void (+ 1 overload)
Set<T>() : bool (+ 1 overload)
ViewModelBase() (+ 1 overload)

Figure 13 - Class diagram of the MainViewModel class of the editor module.

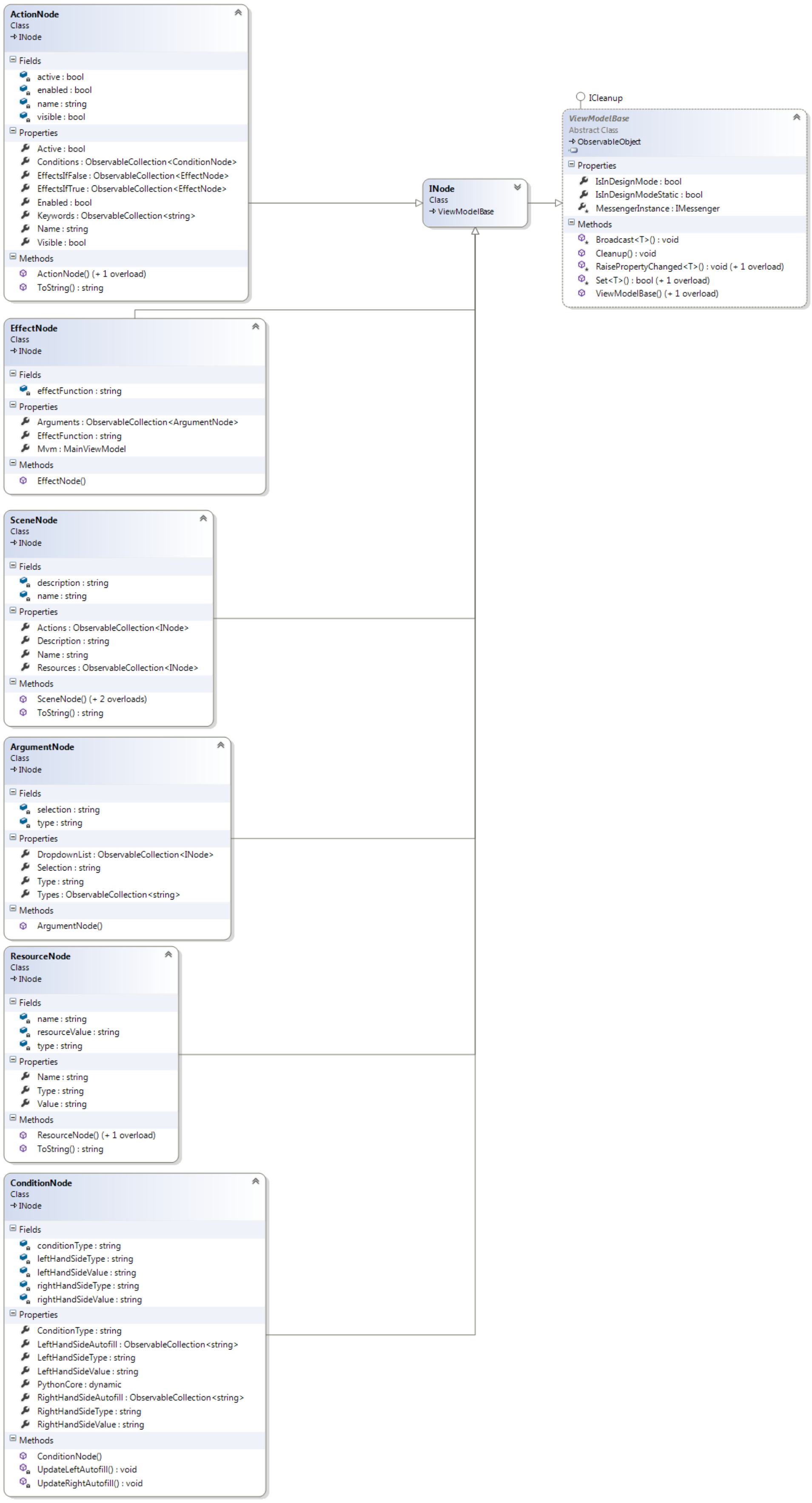

ActionNode
Class
INode
Fields
active : bool
enabled : bool
name : string
visible : bool
Properties
Active : bool
Conditions : ObservableCollection<ConditionNode>
EffectsIfFalse : ObservableCollection<EffectNode>
EffectsIfTrue : ObservableCollection<EffectNode>
Enabled : bool
Keywords : ObservableCollection<string>
Name : string
Visible : bool
Methods
ActionNode() (+ 1 overload)
ToString() : string
ICleanup
ViewModelBase
Abstract Class
ObservableObject
Properties
IsInDesignMode : bool
IsInDesignModeStatic : bool
MessengerInstance : IMessenger
Methods
Broadcast<T>() : void
Cleanup() : void
RaisePropertyChanged<T>() : void (+ 1 overload)
Set<T>() : bool (+ 1 overload)
ViewModelBase() (+ 1 overload)
INode
Class
ViewModelBase
EffectNode
Class
INode
Fields
effectFunction : string
Properties
Arguments : ObservableCollection<ArgumentNode>
EffectFunction : string
Mvm : MainViewModel
Methods
EffectNode()
SceneNode
Class
INode
Fields
description : string
name : string
Properties
Actions : ObservableCollection<INode>
Description : string
Name : string
Resources : ObservableCollection<INode>
Methods
SceneNode() (+ 2 overloads)
ToString() : string
ArgumentNode
Class
INode
Fields
selection : string
type : string
Properties
DropdownList : ObservableCollection<INode>
Selection : string
Type : string
Types : ObservableCollection<string>
Methods
ArgumentNode()
ResourceNode
Class
INode
Fields
name : string
resourceValue : string
type : string
Properties
Name : string
Type : string
Value : string
Methods
ResourceNode() (+ 1 overload)
ToString() : string
ConditionNode
Class
INode
Fields
conditionType : string
leftHandSideType : string
leftHandSideValue : string
rightHandSideType : string
rightHandSideValue : string
Properties
ConditionType : string
LeftHandSideAutofill : ObservableCollection<string>
LeftHandSideType : string
LeftHandSideValue : string
PythonCore : dynamic
RightHandSideAutofill : ObservableCollection<string>
RightHandSideType : string
RightHandSideValue : string
Methods
ConditionNode()
UpdateLeftAutofill() : void
UpdateRightAutofill() : void

Figure 14 - Class diagram showing how the game model is encompassed in the ViewModel of the MVVM pattern of the editor.