Configurable User Interfaces with XML

Final Report

Author: James Briggs (1029537)
Project Supervisor: Dr Frank Langbein
Project Moderator: Dr Xianfang Sun
BSc (Hons) Computer Science
School of Computer Science and Informatics, Cardiff University
Date of Completion: 3rd May 2013

Abstract

Hard coding user interfaces is a tedious and time consuming task which usually ties an application to a specific platform or programming language. Many attempts have been made to define a cross-platform markup language for creating user interfaces however these are often tailored to a certain framework or designed for use within a limited environment. For this project an XML based file format capable of abstractly defining user interfaces and elements of application interactivity is proposed. This format allows user interfaces to be rendered on any device or platform with a compatible API library. The evaluation shows that this is a viable solution to the problem and that with future development a system such as this could replace many traditional methods of creating user interfaces.

Acknowledgements

I would like to thank my supervisor, Dr Frank Langbein, for his continued support and guidance throughout the course of the project and my moderator, Dr Xianfang Sun, for his informative feedback on the initial project plan and interim report.

Table of Contents

Table of Figures

1. Introduction

This section provides an introduction and overview of the contents of the report. The project is reintroduced and amendments to the original aims and objectives are discussed along with any extensions or new developments.

1.1 Project Aim and Objectives

The overall objective of the project is to address the problem of developing cross platform, flexible user interfaces. As stated in the initial plan the XML markup language is used to develop a standardised, flexible file format for encoding a user interface and associated event handling. An approach analogous to HTML or XUL is adopted although the project has a more concentrated focus on the layout and functionality of elements rather than their appearance or theme. Platform independence and diversity are key considerations and as such the system must both support a wide range of platforms and operating systems and facilitate future expansion to them. The following list presents a condensed set of aims and objectives derived from those in the Initial Plan and Interim Report:

- Create XML schemas describing the interface file format.
- Create lightweight yet practical API libraries for constructing and managing interfaces within client applications.
- Develop a visual interface editor to allow the production of interfaces in a high level manner.
- Develop a number of exemplar applications demonstrating the core functionality of the system.

The majority of aims, objectives and requirements remain unchanged from the Initial Plan and Interim Report. The most significant difference is the shift of focus from developing schemas and multiple API libraries to developing a single yet flexible informal standard and API library to set the standard for future implementations. More attention is devoted to the overall architecture of the system and abstraction of the file format than low level technical aspects.

1.2 Beneficiaries and Scope

As mentioned in the Interim Report this project has the potential to benefit a wide variety of users with varying abilities. The foremost target audience is software developers with fairly substantial background knowledge of programming and user interfaces. Nonetheless users without a high level of technical expertise or user interface design skills can still benefit from the project. Whilst developing full applications is the primary use case for the project it can also be applied to a range of other scenarios. For example it could be used by researchers requiring a quick and simple interface for a command line tool or algorithm. It could also be used by designers as a prototyping tool without implementing any actual functionality.

Within the current implementation the scope is limited to desktop applications. Whilst the file format is not restricted to any particular platform the Python API library will only function on desktop computers. The abstract nature of the file format allows it to be used on any platform for which an API library exists. Platform compatibility is simply an implementation issue rather than an issue with the overall architecture. The current implementation does not include the full range of available user interface elements however the architecture provides a large scope for expansion.

1.3 Approach

The agile development methodology is adopted throughout the course of the project. This approach complements the exploratory nature of the project by embracing requirement changes and differing circumstances at any stage [1]. It also suits the limited time scale well as working software is delivered on a weekly basis rather than monthly. Agile emphasises the importance of good design and technical excellence which are of key concern within the project.

The various components of the system were implemented in a compartmentalised fashion. The first stage of development involved outlining an informal draft XML specification and creating the visual interface editor. The second stage involved refining the XML specification, amending the interface editor as necessary and implementing the API library. Within each stage development tasks were further subdivided. This helped improve the overall quality of the system by ensuring that each component was fit for purpose before commencing further development.

Incremental integration testing is used as it fits well with the agile methodology and the accelerated development cycle required by the project. New features and functionality are tested continually during implementation to ensure that they are fit for purpose and function as required.

1.4 Outcomes

This project has produced three significant outcomes:

- A file format and architecture capable of encoding a user interface and event bindings in an abstract, cross-platform method.
- A visual interface editor allowing full interfaces to be designed in a high level way. Additionally the majority of the event scheme can be prescribed within the editor.
- An API library demonstrating the system functionality within client applications and setting the standard for future API library implementations.

2. Design

This section will outline the design of the chosen solution to the previously described problem. The proposed solution is described at a high level and for the purposes of clarity is split into three distinct parts: the interface editor, the API library and the file format. These three components of the system could each be considered separate products and for the purposes of maintainability are described separately.

2.1 Interface Editor

The Interface Editor is designed to simplify the process of creating a GUI and associated interactivity by providing a clean and simple method of arranging elements, describing slots/slot mappings and connecting slot mappings to signals. At a basic level the application allows a user to describe the fundamental components of an interface in an abstract, visual manner without the added complexity of functionality specific to a certain GUI framework. This description can be stored and retrieved in a standard XML format without requiring the user to write XML themselves.

2.1.1 Visualisation

Visualising components of an interface can greatly reduce the time spent on development as well as shifting user focus from the intricacies of the interface or framework to the usability and functionality of the interface in general. Various methods are used within the editor to visualise the arrangement of elements and the interactivity. A visualisation of a standard window is used to represent what the final interface will look like when rendered. This window and elements within it do not react to events as they would normally. Elements will only react to click-and-drag events which allow their position to be modified. Similarly the window itself can only be resized. A simple and intuitive drag-and-drop style is used for arranging elements. Dragging an element from the toolbox onto the window visualiser surface places the element at the desired position. Further adjustments can be made to the element by altering properties in the property grid. All changes made to elements via the property grid are reflected in the window visualiser. Similarly the signal grid can be used to assign slot mappings to element signals. An overall view of the interface is given by the element tree which details each element within the interface, highlighting the hierarchy created through component nesting.

2.1.2 Abstraction and Automation

Hard coding user interfaces is a notoriously laborious and time consuming process with the added inconvenience of constricting the application to a specific programming language and GUI framework. By adding a layer of abstraction above the GUI framework a versatile, cross platform solution can be derived whereby no part of the system is coupled with any specific platform or framework. The data types used to represent the visual parts of the interface

are generalisations of elements which can be found in most common GUI frameworks such as frames, text boxes and buttons. The properties associated with elements are also in a generic form in order to avoid platform specificity. Similarly the methods for describing slots and slot mappings are somewhat high level and implemented in a generalised manner, abstract from any particular programming language. Many of the processes involved in hard coding a GUI are automated within the editor. Each element has a certain set of defaults potentially allowing certain simple interfaces to be created simply through the drag-and-drop mechanism without changing any properties. Serialisation and deserialisation to XML (loading and saving) is a completely automated process which at no stage requires the user to manually edit the outputted file or produce their own XML. This removes the complexity of learning the file syntax and manually writing XML.

2.1.3 Data Types

In order to implement the required functionality in a clean, maintainable, object oriented manner a number of custom data types are implemented. All visual elements within the application indirectly inherit from a superclass representing an element at the most generic level possible. This class includes methods for controlling properties and signals as well as the visual rendering of the element. From this superclass two additional classes are derived, leaf element and branch element. A leaf element is one which cannot have children. This subclass does not extend the superclass with additional member functions, instance variables or properties but instead provides a method of distinguishing between element types. The distinction between a leaf element and a branch element is that the latter may have child elements contained within it. The branch element class extends the superclass by including additional instance variables and member functions for handling the addition, removal and retrieval of child elements. Declaring an instance of a branch element is synonymous with creating the root node of an element tree. This form of composition helps to provide a clear element hierarchy whilst gracefully accommodating varying types of elements.

In order to simplify the serialisation/deserialisation of elements as well as many other internal processes all properties are stored in a list structure defined within the element superclass. Several different types of property are defined to represent the available data types, such as string, integer and double. These property types all inherit from a property superclass which defines abstract accessor and mutator methods as well as a method for defining how the property type should be rendered within the property grid. Additional metadata is stored with each property such as a “friendly” name which helps to explain the purpose of a property in a more user friendly manner than the internal representation of the name provides.

Each element can have a number of associated signals. A signal is equivalent to an event and is represented by a simple data structure essentially associating a signal name with the

name of a slot mapping. The slot mapping class is another simple container class storing the mapping name, associated slot name and a list of parameter mappings along with appropriate accessor and mutator methods. A slot mapping details which slot should be called and the parameters which should be passed to it. The slot class is also a simple container class storing the slot name and a list of parameters with their associated data type. It allows functions within an application to be described in the most generic form possible maintaining platform agnosticism. A slot is tantamount to an event consumer, the entity which receives the signal from an element.

2.1.4 Static Architecture

Throughout the design of the editor, and indeed the entire system, an object-oriented approach has been adopted. Object-oriented development has proven to be a viable method of software development with many established conventions, practices and patterns. A mixture of different object-oriented techniques has been applied where appropriate. For the purposes of intelligibility and maintainability the editor is partitioned into a number of logical classes and subsystems. Each class is stored in a separate file with the appropriate name and extension in line with recommended conventions [1].

The application entry point `Editor` instantiates an instance of the `EditorWindow` class which represents the main portion of the editor interface. The `Editor` class provides easy access to the element, slot and slot mapping lists through a number of member functions and accessor and mutator methods. This helps to control access and mutation of these resources in addition to providing a central point at which they can be reached by subsystems. From within the `EditorWindow` class dialog windows for controlling slots and slot mappings are invoked. Each of these dialogs follows a similar, uniform pattern. `SlotsDialog` provides a list of all the slots within the client application along with their associated parameters. From this dialog new slots can be added and existing ones, along with their parameters, can be amended or deleted. `AddSlotDialog` presents an interface for creating a new slot by simply providing the name. Once a slot is created the main slots list is updated and parameters can be added via the `AddParamDialog`. This dialog presents two options for adding parameters: the name and data type. The data type is limited to the available property types within the system as discussed in 2.1.3 Data Types.

Two helper classes are introduced to handle element IDs and application preferences. The classes facilitate the delegation of trivial responsibilities and help to adhere to the separation of concerns principle. The `IDManager` class stores all element IDs in use within the application. Methods are provided for adding and removing IDs, resetting the class state, checking whether an ID is available, and retrieving the next logical ID for a given element type. All elements within the interface require an ID attribute from creation. The most logical method of assigning default IDs is to append a number to the element type in a superincreasing fashion, such as “Button1”, “Button2”. The use of such a class increases the

computational efficiency and simplicity of the ID management process. The `PreferenceManager` class allows the storage and retrieval of user customisable application preferences relating to the editor. These preferences include font and window visualiser grid size.

2.2 API Library

Arguably the most fundamental part of the overall system is the API library. This allows XML interface definition files to be loaded and rendered within the client application. It also handles the connection of user defined functions to events within the slot and signal subsystem. The library has been designed with simplicity in mind and the ideal that the developer should be able to focus on the core functionality of their application rather than the trivialities and technical issues of the GUI and event handling. Figure H shows a sequence diagram explaining how the client application and various components of the API communicate.

2.2.1 Loading and Rendering

In order to load and display an interface an instance of the `GUI` class must be instantiated. The load method must then be called passing the path to the interface file. The library handles the parsing of all elements within the interface file including the element tree and slot mappings. Similarly to the editor, elements are stored in a more abstract representation implemented by the library as opposed to their native GUI toolkit forms. In accordance with the adapter pattern the underlying components are stored as references within the container class. This abstraction permits the use of any GUI toolkit. If the current toolkit were to become unsupported it could simply be swapped out in favour of a different toolkit without requiring modification of client applications. The choice of toolkit is determined by the API library implementation and cannot be changed by the user.

Element access and manipulation methods are exposed through the API library. A Document Object Model style is used to retrieve elements. The `GUI` class contains a `GetElementByID` method which allows elements to be retrieved based on their ID. The underlying component of each element can be retrieved through the API. Whilst this breaks encapsulation it also facilitates lower level access to the toolkit component allowing more advanced users to implement custom functionality or non-standard behaviour. This is not advised however as functionality which works on one specific platform may not work universally. This practice should only be exercised when the use of a single, specific platform is guaranteed.

Rendering of the interface is initiated by calling the display method. This method shows the main window and invokes the main application loop allowing components to be rendered and events to be handled.

2.2.2 Connecting Signals

Whilst the interface editor handles the mapping of signals to slots the body of slot methods must be written by the developer within the client application. This allows a greater degree of freedom and flexibility as slot methods are not constrained in any way by the editor or interface file. The method body can perform any of the functions available to the client programming language and platform, the only condition being that the name and parameters must match those described in the interface file.

The connection of the events of underlying components to slot methods is not handled automatically by the library during interface loading. Instead the connect method must be called passing an array containing all slot methods as a parameter. The library will then match each slot mapping to a provided slot method, dynamically binding any parameters where appropriate. Slot methods must be declared before calling the connect method.

2.2.3 Data Types

The API library includes many data types similar to those found within the editor (see Figure G for a class diagram). The same set of elements is implemented following the same naming pattern for properties. This ensures consistency throughout different subsections of the system. A comparable pattern of inheritance is implemented within the library for elements. Each element can inherit from either `XElementLeaf` or `XElementBranch` which in turn inherit from the superclass `XElement`. The single distinction between `XElementLeaf` and `XElementBranch` is that the latter must implement a `GetContentComponent` method to allow for shortcomings within the GUI toolkit (see 3.2.3 Element Classes). `XElementLeaf` does not add any additional functionality and is implemented solely for the purposes of consistency and semantics. The element hierarchy has much less of a weighting within the library as it is only used during the initial loading and processing of an interface file. As the element tree is traversed the positions of nested elements are calculated relatively to the position of their parent. Once this process is complete elements are stored in a flattened, non-hierarchical manner. This permits efficient access to elements, which are indexed by their ID, with a constant time complexity.

In order to maintain the most dynamic approach possible and facilitate run-time binding of function parameters within the signals and slots subsystem element properties are stored within a list rather than member variables. In order to ensure that the property list contains data which is current internal responses to events must be implemented to handle external state changes. For example when a user changes text in a text field this state change must be reflected within the property list. Member functions unique to each element are used to handle such scenarios. These functions are bound to the appropriate event of the underlying component upon creation.

An additional data type `GUI` is implemented to represent the entire interface. It contains all of the elements, connections, slot mappings and connected methods needed to render the interface and provide interactivity. Methods are provided for loading an interface file (`Load`), connecting slot methods (`Connect`) and displaying an interface (`Display`). An additional method `GetElementByID` is implemented to allow references to individual elements to be retrieved based on their ID in a similar manner to the HTML DOM counterpart [2]. This is useful in situations where a slot needs to mutate an element or multiple elements. The `GUI` data type is the only data type within the library that must be specifically instantiated in the client application. All other data type instances are held internally within the library and, in the case of elements, are exposed via API methods.

In order to store the mappings of parameters to slots a `SlotMapping` data type is implemented. This data type stores the associated slot name and a list of parameter bindings. Each parameter binding is of the type `SlotMappingBinding`. This data type stores the name of the parameter, a reference to the element and the name of property. A third data type, `Connection`, allows slot mappings to be linked to the signals of an element. It stores a reference to the element, a reference to the underlying event synonymous with the signal and a reference to the slot mapping. Using this pattern slot mappings can be reused as any number of elements can be connected to a single slot mapping. Using references to elements and slot mappings rather than their string IDs helps to minimise the memory footprint of the library whilst making the aforementioned resources quicker and easier to access. All data types relating to the signals and slots subsystem are immutable from the library user's perspective and as such are not exposed via any API methods. Although defined within the editor a data type representing a slot is not necessary as all the necessary information is provided by the three above-mentioned data types and the slot methods specified by the user.

2.2.4 Static Architecture

The static architecture of the API library is simpler than that of the editor. This is due in part to the fact that it has fewer classes and that it does not have to allow for the same form of interactivity as the editor. Each class is stored in a separate file for the purposes of organisation and code maintainability. To facilitate portability and ease of use the library is compiled into a module which can then be loaded into the client application using standard module import techniques. This had the added benefit of disallowing permanent modification of the library code ensuring that user's stick to the standards and conventions of the system. The API is architected using a black box approach whereby the internal workings of the library are not exposed to the user. Instead the system can be communicated with via API calls to perform actions such as loading an interface or connecting events. The entire process of creating, rendering and receiving events from the GUI is transparent to the client application.

2.3 File Format

The file format used for storing interface definitions has been designed to be as generic and flexible as possible. XML has been chosen as the most appropriate markup language for implementing the file format due to its simplicity, flexibility and generality [3]. The hierarchical nature of XML also makes it an ideal candidate for storing data as a tree structure. The XML specification provides the basic syntax and structural elements of a document upon which advanced data representations can be built. Many high performance libraries exist for parsing XML documents. This enables elements within an XML document to be deserialised within a client application into a native data structure. Similarly data held within class instances in an application can be serialised into XML form. Any file loaded using the editor or API library must adhere to this format otherwise successful loading and rendering of the interface is not guaranteed. The file contains a root element `xgui` with a further 4 mandatory elements within it: `elements`, `slots`, `slot_mappings` and `connections`.

2.3.1 Elements

The `elements` component of an interface file contains a hierarchical tree of elements within an interface. Each element has a unique identifier (ID) associated with it which is used to connect signals, map properties to parameters and perform the aforementioned functions within the editor and library. Although no specific distinction is made between branch and leaf elements, as is the case within the editor and library, it is implied. Any element may be nested inside a branch element, such as a window, panel or group box, by locating it within the `elements` element of the parent. This form of nesting is valid up to a depth of 10 to avoid issues such as slow processing times and unnecessarily complex interface structures. The consistency of element names is maintained throughout the system. Using canonical names for elements makes it easier for the developer to become familiar with the system particularly when switching between using the editor and editing interface files manually. It also provides a clear abstraction from any GUI toolkit as well as simplifying the implementation of the factory method pattern within the editor and library.

2.3.2 Slots

The `slots` element is only parsed within the editor and as such is not strictly required to create a valid interface. Each slot represents an abstract view of a slot method within the client application. A slot simply contains a name attribute and optionally a number of nested parameters. The name of a slot is synonymous with the slot method within the client application. Similarly each parameter is synonymous with an input parameter of the same name.

2.3.3 Slot Mappings

The `slot_mappings` element can contain a number of slot mappings describing which element properties should be bound to particular slot parameters and to which slot the mapping relates. Each slot mapping is referred to by a uniquely identifying name. Multiple slot mappings can be defined for a single slot allowing slot methods to be reused whilst still permitting different input parameters.

2.3.4 Connections

The `connections` element stores any connections between element signals and slot mappings. Each connection contains an element ID, signal and slot mapping name. The element ID refers to any element within the `elements` section. The signal is simply an abstract text representation of the signal event such as click or text changed. These are mapped to the events of underlying components within the library. The slot mapping name refers to a slot mapping defined in `slot_mappings` which will be called when the signal is emitted.

3. Implementation

This section details the implementation process of the system based on the aforementioned design. A low level, technical insight is given into how the system was developed in a compartmentalised fashion. Splitting the system in this way makes initial development simpler and makes future development more feasible. An important consideration regarding the choice of tools and technologies throughout the implementation was portability across different platforms.

3.1 Implementing the Interface Editor

The Interface Editor is one of the principle components of the system. It allows a user to create an interface in a visual manner removing the intricacy of hand writing XML. The editor hides much of the complexity behind developing an interface and associated interactivity.

3.1.1. Development Tools

Java was chosen as the programming language and software platform for developing the editor. Java offers significant benefits over other potential languages considered. In speed benchmarks Java was shown to outperform Python, Ruby and Pascal in almost all tests performed [4]. Java runs on almost all major hardware and software platforms through the Java Virtual Machine without requiring modification or specifically targeted builds [5]. Less tangible benefits of Java include the large user and support base, abundance of documentation and rapidity of development. Much of the lower level and boilerplate functionality of an application is handled natively by Java whereas this is not generally the case when using languages such a C and C++. The Notepad++ text editor was used as the principal tool for writing the editor code. The official Oracle implementation of the Java platform and compiler (including the *java* and *javac* commands) was used.

3.1.2. Creating an Interface

3.1.2.1 Window Visualiser

The `WindowVisualiser` class extends the `JInternalFrame` class found within the Swing GUI toolkit. `JInternalFrame` is a lightweight implementation of a standard `JFrame` implementing much of the same functionality. The key difference is that a `JInternalFrame` is generally added to a `JDesktopPane` rather than rendered as a standalone frame. A `JDesktopPane` allows a virtual desktop to be created within an application. A limited subset of standard frame behaviour is required within the editor. For example the user should not be allowed to close, minimise or maximise the frame as this would disrupt the editor functionality. `WindowVisualiser` imposes limits such as these whilst still maintaining the core functionality of a `JInternalFrame`. Additional responsibilities such as performing window size calculations and responding to drag-and-drop actions are also delegated to the class. The use of the

native `TransferHandler` class allows components to be dragged from the component toolbox onto the `WindowVisualiser` surface. A new element is instantiated via the `ElementFactory` and positioned at the drop point.

3.1.2.2 Property Inspector

The property inspector is a highly customised implementation of a `JTable`. It presents properties of the selected element in a tabular, key-value style allowing manipulation by the user. Appropriate input components are used to manipulate the value based on the property data type. For example a `JComboBox` with possible values of *true* or *false* is presented for a Boolean property type. The JTable does not store any data; it simply presents and facilitates editing of it. The underlying data is stored within an instance of the `PropertyTableModel` which extends the `AbstractTableModel`. This data model links directly to the properties of elements and is responsible for orchestrating retrieval and modification of the underlying data.

3.1.2.3 Element Classes

The element superclass `XElement` defines the standard member variables and methods required by all inheriting elements. This includes the component, a list of properties and a list of signals. The component is a standard Swing component used within the window visualiser. The list of properties and list of signals are both stored as a generic `Map` allowing interoperability between concrete implementations such as `HashMap` or `LinkedHashMap`. In this case `LinkedHashMap` is used as it preserves the insert order of list items maintaining consistency across elements and allowing persistent, logical groupings of properties [6]. This facilitates quick retrieval of properties and signals as they are indexed by their String ID. The use of lists over member variables within subclasses allows for a higher level of dynamicity. The problem of XML serialisation and deserialisation is mitigated as the same generic code can be used regardless of the element subtype. All member variables are declared with protected visibility allowing subclasses to access the data in full without the overhead of invoking accessor or mutator methods. Additionally three abstract methods are declared which subclasses are required to implement. The `generateComponent` method facilitates the initial generation of the underlying Swing component. This is not done within the constructor as properties are not provided until after object construction in certain scenarios such as when creating an element instance from XML. The `updateComponent` method updates the underlying component to reflect external property changes. This method is essentially used to notify the object of a property change, usually originating from the property inspector.

3.1.2.4 Element Factory

The `ElementFactory` is a simple implementation of the factory method pattern. The implementation differs slightly from the standard factory method pattern as all variables and member functions are declared static. This avoids unnecessary initialisation logic and provides a quick, simple and global method of accessing the factory. Two member functions

are defined for creating elements. The `create` method accepts an XML element as input and returns a fully initialised instance of the appropriate subclass of `XElement`. Instead of providing the element constructor with arguments this method calls the empty constructor. Each property within the XML element is then pushed into the element instance. This allows for a higher level of dynamicity as properties do not necessarily need to be known to the editor beforehand. It also enables the reuse of identical initialisation logic across all element types. The `createDefault` method creates an element instance based on the type provided by a string argument. The element is initialised with hard-coded default values. This method is used within `WindowVisualiser` for generating an element. To avoid the use of multiple conditional statements two `Map` instances are declared. The first maps a string value representing an element type to each class of element, the second maps a string value representing a property name to each property type class. A static initialisation block is used to instantiate and populate both of these maps removing the need for specific, external initialisation. Class instances are dynamically instantiated based on a single lookup in the element class map or property class map.

3.1.2 Signals, Slots and Slot Mappings

Each `XElement` can have a number of `XSignal` instances stored within a map. The `XSignal` class is a simple container holding its own name and the name of the associated slot mapping.

The `XSlotMapping` class essentially describes how properties of elements are mapped to slot parameters. Slot parameters are stored in a map containing `XParameterMapping` instances indexed by their name. Similarly to `XSignal`, `XParameterMapping` is a simple container holding string values representing the element name and the appropriate property name. All slot mappings within the system are stored in a static, globally accessible variable. This enables reuse of slot mappings and avoids the complication of passing object references to child dialogs. Besides standard accessor and mutator methods additional methods for adding and removing slot parameters are provided. A `toXML` method is provided for creating an XML representation of the class instance following a similar convention to the standard `toString` method. This reduces the amount of code required when serializing the interface into XML form as well as achieving further compartmentalisation.

Slots provide a generic, abstract way of representing a function within the client application. They simply describe the semantics of a function, such as the name and parameters, rather than what the function achieves. A language and platform agnostic view of functions is taken. For example a parameter type is required regardless of whether the client application is written in a loosely typed language like Perl or a strongly typed language like C#. The system is designed such that parameter types are generalised to their closest equivalent within the client application. For example if a C library were to be implemented the String type could be mapped to the String data type provided by the C standard library or a

character array. An enumeration is defined within the `XSlot` class canonically specifying the available parameter types. A slot stores the slot name as a standard string. Parameters are stored as a map associating the parameter name with a parameter data type. Similarly to the `XSlotMapping` class a method is defined to serialize an `XSlot` instance into XML form. Methods for adding and removing parameters are implemented to avoid breaking encapsulation by exposing the underlying map.

3.1.3 JDOM File Handling

Although the Java platform includes several native XML APIs the community driven JDOM library was chosen to handle XML documents. JDOM presents many benefits over the native APIs. JDOM is written specifically for Java and as such makes use of many language specific features. Where possible JDOM uses inbuilt data types such as String and standard collection classes such as `List` and `Iterator` [7]. Another benefit is the class driven implementation of the DOM. Creating new XML elements is a simple as instantiating an instance of `Element` [8].

3.1.3.1 Serialising

The elements within an interface are stored in a hierarchical manner in the form of a simple unordered tree. Instead of using a standard collection class a single root element, an instance of `XElementBranch`, contains all of the elements within an interface either directly or within child elements. When serializing elements into XML form a depth-first tree traversal is performed using recursive methods. The `generateXml` method is called initially providing the root element and a newly instantiated JDOM Element `elements` as parameters. The appropriate data is appended to elements. If the `hasChildren` method returns true a new JDOM Element `elements` is created to store child elements. The `generateXml` method is called providing each child element and the `elements` element as arguments. The `hasChildren` method is implemented by both `XElementBranch` and `XElementLeaf` despite the fact that `XElementLeaf` can never contain child elements. This simplifies the `generateXml` method by removing the need to check the superclass type via `getSuperclass` in addition to checking whether the element has children.

```
generateXml(element, parent)
  new_element = new xml_element(element.type)
  for each property in element
    append property to new_element
  if new_element has children
    for each child in element
      elements = new xml_element(“elements”)
      generateXml(child, elements)
  append new_element to parent
```

Figure A – Pseudo code for the serialisation tree traversal algorithm.

The traversal algorithm (Figure A) has the desirable quality of preserving the insertion order of elements. This helps to maintain consistency particularly when the interface editor and manual editing of the XML file are used interchangeably.

Slots and slot mappings are stored in a simple, linear map data type. Serialising these elements is a simple case of iterating over each one and appending the results of the `toXML` methods to the containing XML element. Serialising connections is a somewhat more complicated process as connections do not exist as concrete or detached classes. Each signal of each element is iterated over. If a slot mapping is defined for the signal, as determined by the signals `hasMapping` method, a new connection element is generated.

The JDOM `XMLOutputter` class is used in conjunction with the standard `java.io.FileWriter` to output the resulting byte stream to an XML file on disk. Despite the penalty on file size the multi-byte UTF-8 encoding scheme is used. UTF-8 offers the benefits of an expanded character set allowing any of the 110,000 Unicode characters to be represented as opposed to the 128 offered by ASCII. The Unicode standard is designed to allow documents to be used seamlessly across multiple platforms and programming languages [10]. XML output is formatted in pretty-print style whereby elements are preceded by an amount of whitespace, as decided by JDOM's `getPrettyFormat` method, dependant on their depth within the tree hierarchy [11]. This improves the readability of XML documents should manual editing be desired.

3.1.3.2 Deserialising

Deserialisation works in a similar way to serialising. The `SAXBuilder` class is used to construct a `Document` instance. `SAXBuilder` makes use of third party SAX parsers which offer a smaller memory footprint and faster document construction times than other XML parsing methods [12] [13]. The standard `java.util.Iterator` interface is used to iterate over all of the sub elements within the document (elements, slots, slot mappings and connections). The same tree traversal procedure that is used for serialisation is applied to deserialisation to maintain consistency. The `handleNode` method is called recursively providing the XML element to process, its parent node within the visual interface tree and its `XElement` parent.

3.2 Implementing the API Library

3.2.1 Development Tools

The Python programming language was used to implement an exemplar API library however the structure could easily be applied to any programming language. Version 2.7.4 (the most recent version at the time of development) of the official Python implementation is used. In addition to this the ElementTree XML API (`xml.etree.ElementTree`), wxPython (`wx`) wrapper and Python Imaging Library (`PIL`) are also used.

3.2.2 Deserialising

The API library only facilitates loading of an interface not permanent alternation. As such interface files only need to be deserialised. The `Element` class within the `ElementTree` API allows an XML files to be parsed in a similar way to JDOM making use of the native data

structures within Python. The element hierarchy is traversed using the same tree traversal procedure described in 3.1.3.2 Deserialising. The most significant difference is that the library flattens the hierarchical structure by indexing elements by ID in a `dict` mapping type rather than nesting them hierarchically. As a result elements cannot be dynamically added or removed from the interface as this would require the hierarchy to be retained. As the `dict` type is an implementation of a hash table API methods such as `getElementByID` are generally more efficient as the average time complexity is O(1) with a worst case of O(n) [14]. In the majority of cases hash tables are also more efficient on memory as recursive tree traversal can often require large amounts of memory on the stack. When the `GenerateComponent` method of an element is called it must be passed the containing branch element as a parameter. When instantiating a wxPython widget a parent should be provided unless it is the top level widget within the interface. Within the library this also helps to determine the Z-order of components as well as their relative position within the interface.

Slots do not need to be deserialised as the functions they abstractly describe exist within the client application as standard functions. Sufficient information exists within the slot mappings and connections elements to connect events of underlying components to the signals and slots subsystem.

3.2.3 Element Classes

A superclass `XElement` contains a `dict` object for storing properties which simplifies the deserialising process. Appropriate accessor and mutator methods are provided for changing property values (`SetPropertyValue`), retrieving a property value (`GetPropertyValue`) and checking whether and element has a property (`HasProperty`). The subclasses `XElementLeaf` and `XElementBranch` inherit from `XElement`. All subclasses of `XElementBranch` must implement a `GetContentComponent` method. This allows each subclass to return the component which will contain child elements as this may not be the principal underlying component. For example the underlying component of `XWindow` is an instance of `wx.Frame` however child elements are stored within a `wx.Panel` instance. This overcomes problems encountered in the initial prototype whereby certain components, such as `wx.Frame`, would not render correctly unless child components were added to a `wx.Panel` instance (see Figure B).

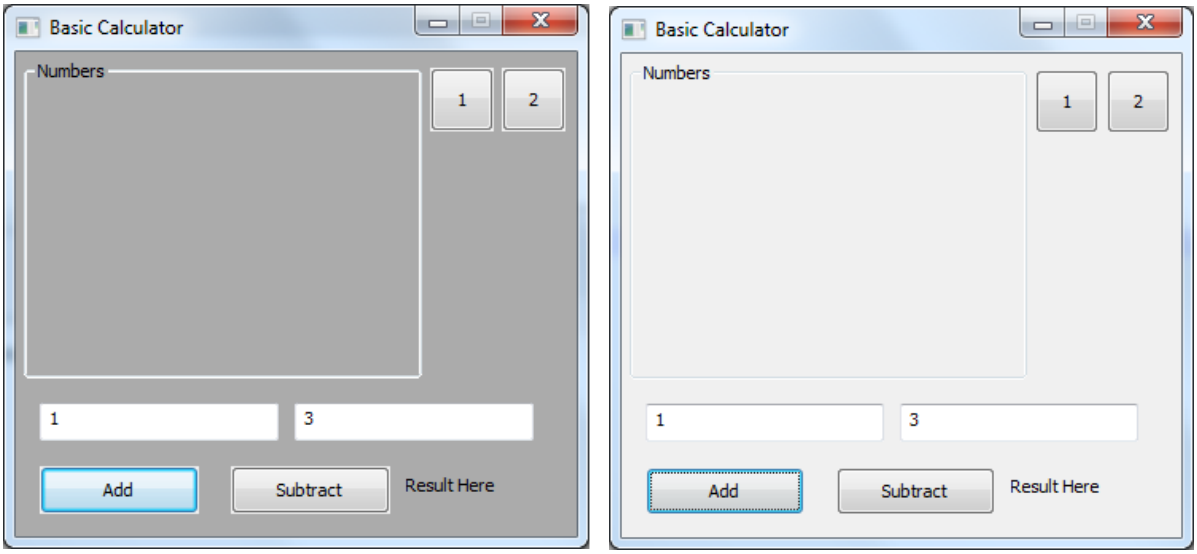

Figure B - Error encountered on Windows 7 and subsequent fix.

The `XWindow` element is one of the most complex elements to construct within the system. This is due to the fact that the underlying `wx.Frame` component uses integer constants to represent different window styles. For example `wxCLOSE_BOX` renders a close button on the window and `wxSTAY_ON_TOP` places the window on top of other windows [16]. Multiple styles can be combined using bitwise *and*/*or* operators [17]. The complexity arises when the decomposed window styles within the interface file need to be represented using wxWidgets constants. As attributes such as `closable` and `resizable` are stored in Boolean form this requires an initial window style to be declared upon which subsequent attributes are appended by performing a bitwise *and* operation on the initial style and the desired style.

To simplify API use subclasses of `XElementBranch` and `XElementLeaf` are extended with member functions. These functions allow manipulation of the attributes unique to each element. For example `XImageBox` includes `GetImage` and `SetImage` methods whereas `XTextBox` includes `GetText` and `SetText` methods. Python's use of duck-typing greatly simplifies this style of implementation as casting or type checking is not necessary [18]. This approach may not be as appropriate in languages with stricter type checking.

3.2.4 API Entry Point

3.2.4.1 Encapsulation

Although Python does not explicitly enforce class member visibility constraints, such as *public*, *private* or *protected*, certain naming conventions can be used to convey the intended visibility [19]. Name mangling is used to declare class members as private to preserve encapsulation and avoid inadvertent errors introduced by the client application. Class members prefixed with two underscores cannot be modified accidentally as the member name is textually changed by the Python interpreter [20].

3.2.4.2 GUI Class

A `GUI` class is introduced encapsulating all of the required class instances and API methods within the library. This is essentially the single entry point of the API as all interactions must be directed towards an instance of this class.

3.2.4.3 Load method

A `Load` method is defined which parses the XML file passed to it and generates all necessary class and component instances. File parsing is not accomplished in the constructor in order to facilitate future development. For example future versions could allow interfaces to be programmatically constructed without requiring file parsing.

3.2.4.4 Connect method

The `Connect` method allows slot functions defined within the client application to be linked to the signals and slots subsystem. This method stores the functions passed to it in a `dict` object then iterates over each connection attaching the appropriate event. The `Bind` function within wxPython is used to link the events of an underlying component to a private `DispatchEvent` method within the library. A lambda expression is used to pass the wxPython event object and matching slot mapping to the method. `DispatchEvent` effectively routes an element signal to the appropriate slot with the desired parameters.

3.2.4.5 Display method

The `Display` method performs the final functionality required to render the interface and initiate the wxPython events system. Displaying the interface is a simple case of calling the `Show` method of the top level element. In order to respond to events the application event loop must be started. This is achieved by calling the `MainLoop` method of the underlying application. The event loop operates asynchronously for the duration of program execution. The underlying wxPython application follows the default blocking behaviour running on a single thread [21]. As a result any code included after calling the `Display` method will not be executed until after the event loop has exited and interface rendering has ceased. Code within slots is unaffected as it is executed from within the event loop. The decision to use a single threaded approach rather than introducing multithreading stems from the requirement for system simplicity. Multithreading requires additional code to handle concurrency and synchronisation issues increasing the complexity of the codebase.

The use of a single threaded model introduces a fairly significant problem which was not foreseen during the initial system design. Code which is particularly CPU intensive or incurs significant time or resource overheads can interfere with the execution of the client application. This results in poor performance or in extreme cases suspends the interactivity of the interface indefinitely. See **Error! Reference source not found.** for further explanation.

3.2.4.6 GetElementByID method

The `GetElementByID` method is introduced to allow programmatic access to the elements of an interface without breaking encapsulation. When called passing an element ID String it

returns a reference to the appropriate element from the internal dictionary or `None` if no such element could be found. It is analogous to the HTML DOM counterpart of the same name which allows elements within an HTML document to be retrieved in a similar way [22]. Initial iterations of the library stored elements in a hierarchical manner. This meant that in the worst case all tree nodes would have to be traversed to try and find the appropriate element leading to a time complexity of O(n). The latest iteration of the library uses a Python implementation of a hash table which offers a more efficient approach (see 3.2.2 Deserialising).

3.2.5 Signals, Slots and Slot Mappings

A list of connections between signals and slot mappings is generated when an interface file is deserialised. A connection stores references to the element it relates to, the slot mapping which will be called and the event within the wxPython framework which the signal represents. To simplify the association of signal names and underlying events the library contains a dictionary mapping signal names to wxPython events such as `click → wx.EVT_BUTTON`.

A list of slot mappings is stored relating functions passed to the library through the `Connect` method with a set of parameter bindings. This allows the `DispatchEvent` method to pass the appropriate parameter values to slot functions. When the `DispatchEvent` method is called a dictionary is constructed mapping a parameter name to a value. The `**` operator is used to unpack the dictionary into an argument list which is passed to the appropriate slot [23]. The use of argument list unpacking allows the parameter binding mechanism to remain as dynamic as possible whilst still allowing functions to be written in a standard way using parameters. The alternative would be to force users to write slot functions which accept an array of values as an argument however this would not feel as natural as using parameters for the majority of users in addition to increasing complexity.

3.3 File Format

The file format is designed with simplicity and extensibility in mind. All elements are clearly distinguishable as each element has a different tag to signify its type. This offers more clarity than using the same tag for each element and further specifying the tag as an attribute such as `<XElement type="XButton">` as found with certain elements, like `<input>`, in the HTML specification [24]. Attributes are used solely when specifying unique identifiers for elements within the document. For example `<XButton id="button1">` or `<slot name="subtract">`. All other data is stored within child elements, such as `<width>200</width>`, creating a clear distinction between properties and identifiers. Originally a set of XML schemas were going to be developed outlining the structural restrictions of an interface file however due to time constraints this became infeasible.

4. Results and Evaluation

This section of the report explains the various testing procedures employed to analyse how well the system functions and the extent to which the project aims and objectives have been met. The benchmarking and profiling procedures used to measure the performance and efficiency of the final solution will be discussed in addition to a critical appraisal of the extent to which the requirements outlined in the interim report have been met.

4.1 Evaluation Strategy

The evaluation aims to critically assess the system against all of the original functional and non-functional requirements through a variety of experiments. Due to the exploratory nature of the project a qualitative approach is adopted for the majority of experiments. This allows the actual functionality to be evaluated rather than the intricacies of performance and efficiency. The experiments each aim to evaluate a variety of aspects encompassing both the high level functionality of the system and lower level, technical issues.

4.2 System Experiments

A series of evaluation experiments were performed based on the functional and non-functional requirements specified in the interim report. These experiments provide both quantitative and qualitative measures of how well the requirements have been met as well as justifications for any requirements which have not been fully met or areas in which the system falls short. These tests were conducted using multiple testing approaches. For certain aspects a black-box approach was taken whereby only the system requirements were taken into consideration, not the internal design of the system. For other cases grey-box testing (a mixture of black-box and white-box testing) is used. This allows internal elements of the system to be evaluated whilst still discussing them at a high level.

4.2.1. Render an Interactive, Cross-Platform GUI

4.2.1.1 Setup

The aim of this experiment is to demonstrate that the system can successfully render an interactive user interface across multiple different platforms. Three approaches are taken to demonstrate this. The first approach involves evaluating the stated system compatibilities of the libraries and development platforms chosen to build the system. The second approach offers an empirical analysis across multiple platforms observing the behaviour of the system. The third approach discusses how the overall high level architecture facilitates cross platform development and potential future expansion.

4.2.1.2 Execution

In order to determine whether the system is theoretically cross platform the operating system compatibility of each of the external dependencies was researched. This involved

examining vendor websites and documentation. Additionally licensing restrictions were also researched to determine whether the system could potentially be released as open source software. The following table details the platforms, technologies and libraries used within each component of the system, the most common compatible operating systems and the software licence.

Platform/Technology/Library	Compatibility	Licence
Java (official Oracle implementation).	Windows XP and above, Mac OS X 10.7.3 (Lion) and above, Oracle Linux 5.5+, Red Hat Enterprise Linux 5.5+, Ubuntu Linux 10.04 and above, Suse Linux Enterprise Server 10 SP2/11.x [25].	Proprietary but unrestrictive licence [27].
Swing	Same as Java.	Same as Java.
JDOM	Same as Java.	Apache-style open source license, with the acknowledgment clause removed [28].
Python	Debian, OpenSuse, Fedora, Slackware, FreeBSD, OpenBSD, OpenSolaris, Windows XP and above, Mac OS 10 and above [26].	Python Software Foundation Licence [28].
wxPython	Same as Python.	Open source wxWidgets Licence [31].
XML	Any platform capable of interpreting text files in the given encoding.	Open standard.

This table demonstrates that the system is theoretically compatible with a sufficient cross section of operating systems to be deemed cross-platform. The list of operating systems may not be exhaustive as only the platforms with which they have been fully tested on are included. They may be compatible with other operating systems that are not listed however this would have to be determined on a case-by-case basis.

According to the above table the system should be compatible with most common operating systems. In practice however each operating system tends to have compatibility issues unique to either the software or the operating system itself. To evaluate this the system was tested on a number of different operating systems. This testing was conducted as a black-box end-to-end test whereby certain scenarios involving a large number of paths through the system were enacted and the results observed. The results were almost entirely positive showing only minor visual differences across platforms.

Another aspect of the system to consider is whether the overall, high level architecture facilitates cross-platform compatibility. Testing this is a non-trivial task which cannot be

truly validated without implementing and thoroughly evaluating multiple API libraries across a vast range of platforms. Hypothetically the system architecture facilitates a wide variety of GUI toolkits. The XML file has been designed to serve as a layer of abstraction between the interface layout and the GUI toolkit. As such the system is somewhat loosely coupled with the GUI toolkit.

The architecture of the system has many parallels with HTML and web browsers. The structure of the interface is described in a file which the API library parses within a client application in much the same way that a browser parses an HTML file. It is the API libraries responsibility to render the interface as accurately as possible on the user's device in a similar manner to how a web browser would render webpage.

4.2.1.3 Results

- The system follows many of the same patterns and conventions of HTML therefore in principle it should prove equally diverse and universally compatible.
- The system is compatible with a wide range of the most popular desktop operating systems.
- The architecture is flexible enough to support almost any device capable of processing XML documents.
- The architecture greatly facilitates future development and expansion.
- As long as the underlying GUI toolkit does not radically differ from standard conventions it will fit within the system architecture without requiring changes to dependant classes.

4.2.2. Visual Interface Designer with XML Serialisation

4.2.2.1 Setup

This experiment aims to evaluate whether the visual interface designer has been effectively implemented allowing interfaces to be created in a high level fashion. Additionally it evaluates whether interfaces can be accurately serialised into XML form.

4.2.2.2 Execution

This experiment can be validated fairly simply. The most effective way of testing whether the interface editor is capable of serialising visually designed interfaces into XML files is to perform end-to-end tests. This involves creating an entirely new interface within the editor including visual elements, slots and slot mappings. Additionally unit testing is used to ensure that the editor accurately serialises the required data structures into valid XML.

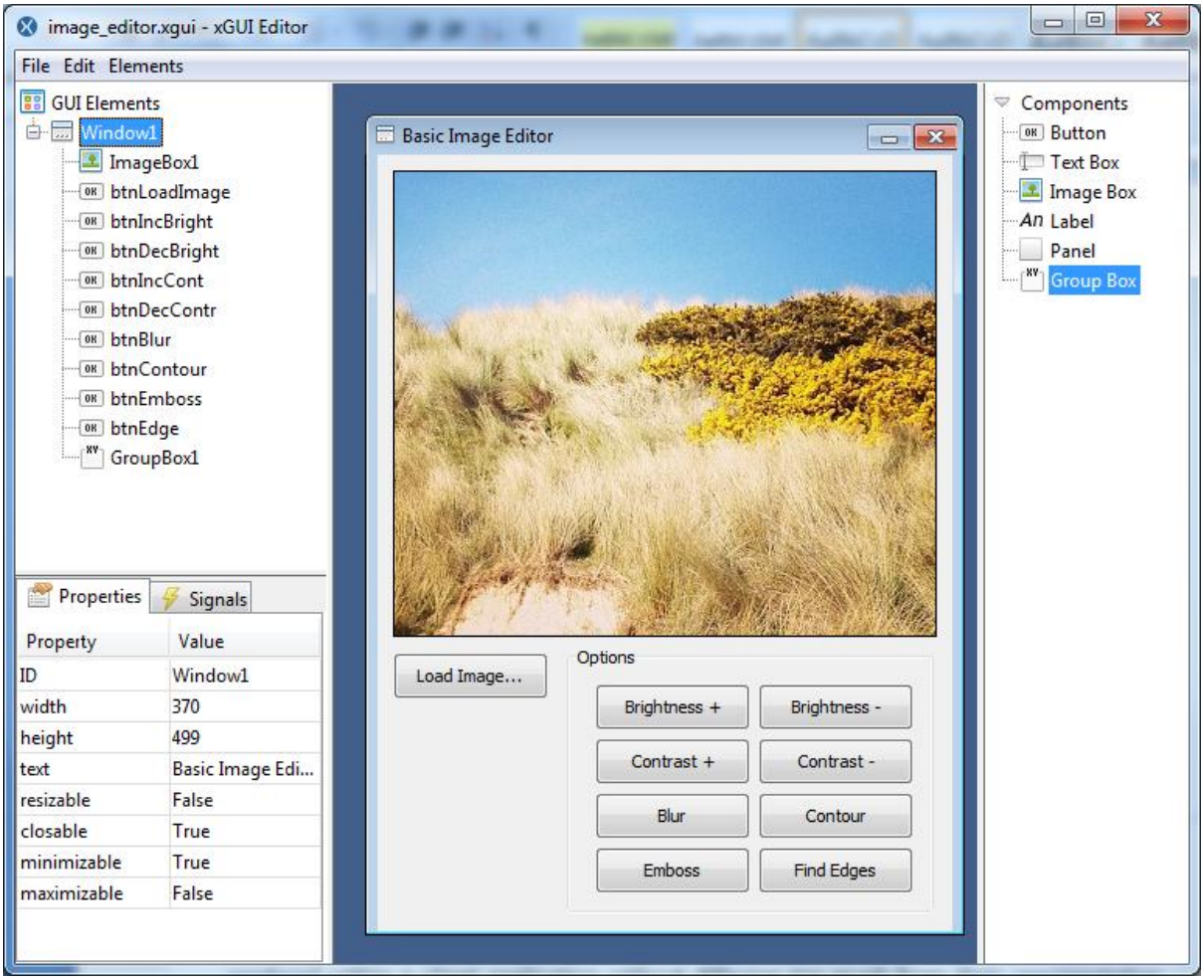

Figure C – An example application in the GUI Editor.

A simple image processing application (Figure B) was developed from start to finish using solely the interface editor and a basic text editor for writing the Python client application. The editor provided all of the required functionality to successfully create the interface. Formally measuring and comparing the time required to produce an interface in the editor against manually writing XML is a non-trivial task. As such the simplest approach was to conduct user tests whereby a small selection of users were asked to complete tasks whilst under observation and provide feedback. Due to time constraints only a small group of users could be feasibly tested. These users have varying degrees of technical ability and background knowledge relevant to the system. Users were asked to implement a simple application encompassing features of the editor including designing an interface, creating slots and assigning slot mappings to signals. The general consensus was that using the interface editor is far more productive than hard coding interfaces using a specific toolkit and programming language or manually writing XML. Users appreciated the interface visualisation and instant reflection of property changes as visualising an interface from a hard coded definition is often a laborious process. Advanced users in particular found the lack of inbuilt XML editing capabilities surprising however agreed that it did not pose a major problem.

Simple unit tests were performed as part of the incremental integration testing strategy to ensure that the editor functioned correctly. All of these tests succeeded showing that the editor fulfils its purpose on a technical level.

See 4.2.7. Produce Example Applications for further example applications and a more detailed explanation of the image processing application.

4.2.2.3 Results

- The interface editor provides a convenient method for quickly and simply defining an interface.
- XML interface files can be serialised/deserialised quickly, simply and accurately whilst still allowing manual editing.
- In most cases it would be quicker to use the interface editor over manually writing XML.
- The editor succeeds in instantly visualising an interface and reflecting property changes immediately.
- The interface editor suits all user groups but appeals more to beginner and novice users. Advanced users may be frustrated at the lack of built in XML editing capabilities.

4.2.3. Implement two API Libraries

4.2.3.1 Setup

This experiment evaluates whether two cross-platform API libraries have been successfully and efficiently implemented. The API library goals are tested to ensure that interfaces can be accurately rendered within a client application without differing too much from their representation within the editor. The library portability is briefly assessed to ensure it can be used on an acceptably diverse range of operating systems.

4.2.3.2 Execution

The original requirements specification states that two API libraries were to be developed. Due to time constraints however this proved infeasible and as a result only a single API library for the Python programming language was implemented. Unit testing and end-to-end testing ensure that this library fulfils the required role of exposing system functionality whilst maintaining a clear abstraction from the underlying GUI toolkit.

The API library clearly demonstrates all of the methods required to render an interface and handle events. The structure established by the library implementation should be applicable to any programming language or platform; any arbitrarily chosen programming language and GUI toolkit should prove to be equally viable as Python and wxWidgets. This implementation essentially serves as an example of the principles and standards that other library implementations should follow.

It was initially envisioned that timings and benchmarks would be required to verify whether a significant time penalty is incurred by using XML over hard coded interfaces. Naturally any system which requires additional I/O operations and processing, in this case retrieving and parsing an XML file, will incur a performance penalty. Through somewhat prolonged use of the system it was observed that only a minimal delay is encountered during the start-up of client applications. As such it was deemed unnecessary to conduct further timing and performance analysis. Once the interface has been constructed within the client application the responsibility of rendering is almost entirely passed to the GUI toolkit, platform and operating system. Quite often the processes used are finely tuned for optimal performance. Attempting to analyse and improve performance of the API library would be time consuming an unnecessary. Certain scenarios, such as constructing particularly large GUIs, may incur large time penalties however it is generally considered bad practice to create interfaces with a large number of elements [27].

Thorough testing across operating system was not feasible due to time constraints however the system was tested on 3 of the most common platforms: Windows 7, Linux Mint and Mac OS X. The result of the testing was positive showing that the API library performed its required functionality on all 3 platforms. Further, more extensive testing would be required to fully evaluate compatibility.

4.2.3.3 Results

- Although only one API library was developed it sufficiently demonstrates the functionality of the system.
- Development of further API libraries for a variety of programming languages/platforms would render the system a more viable solution.
- The library functions well on the limited platforms tested.
- Further testing is required to ensure the system is compatible with other operating systems. Theoretically the library should be compatible with many platforms as described in 4.2.1. Render an Interactive, Cross-Platform GUI.

4.2.4. Emulate the Functionality of a DOM

4.2.4.1 Setup

This experiment analyses how effectively Document Object Model functionality has been implemented within the API library and how well the overall architecture supports it. This involves comparing parts of the system to analogous components within the HTML DOM specification with which many similarities can be found.

4.2.4.2 Execution

Although it would be unwise to implement most of the features within the HTML DOM specification certain methods and practices have been adapted for use within the project. Most notably the `GetElementByID` method is used within the API library to retrieve an element based on its ID string in an identical manner to its DOM counterpart. This is the

only method within the API library which is directly based on the HTML DOM. As such the API library cannot truly be compared to the full DOM specification which includes a much richer set of methods and encompasses more aspects than simply the API definition. A comparable pattern of inheritance is employed whereby all elements inherit from a base class upon which additional functionality and responsibilities are added [28]. The approach taken is more simplistic than the DOM as there are only two base classes from which elements can inherit. The DOM specification describes a much larger number of element types, most of which are not necessary in the context of the system [29].

4.2.4.3 Results

- The `GetElementByID` method is an effective method of programmatically accessing elements within client applications.
- Implementation of additional DOM-based methods could enhance usability, flexibility and efficiency.
- A similar inheritance structure to the one described in the DOM specification is used. This further supports the inheritance choice as the DOM is a reliable convention.
- Full implementation of and compliance with the DOM specification are sacrificed in order to maintain simplicity.

4.2.5. XML Event Signalling Mechanism

4.2.5.1 Setup

This experiment tests how well the signals and slots subsystem has been implemented. This involves evaluating the flexibility and abstractness of the XML representation in addition to evaluating the effectiveness and performance within the API library.

4.2.5.2 Execution

The signals and slots mechanism allows abstract events to be mapped to functions within the client application. All of these mappings are stored within the XML file and can be created simply and easily within the editor as shown in Figure D and Figure E. Slots allow an abstract, language neutral definition of a function to be stored containing the function name and any parameters along with their data type. Within the editor slot mappings can be assigned to events allowing dynamic binding of element properties to parameters of slot functions. This proved a very effective method of event handling as it provides a clear overview of the functionality within the client application and allows slots to be reused.

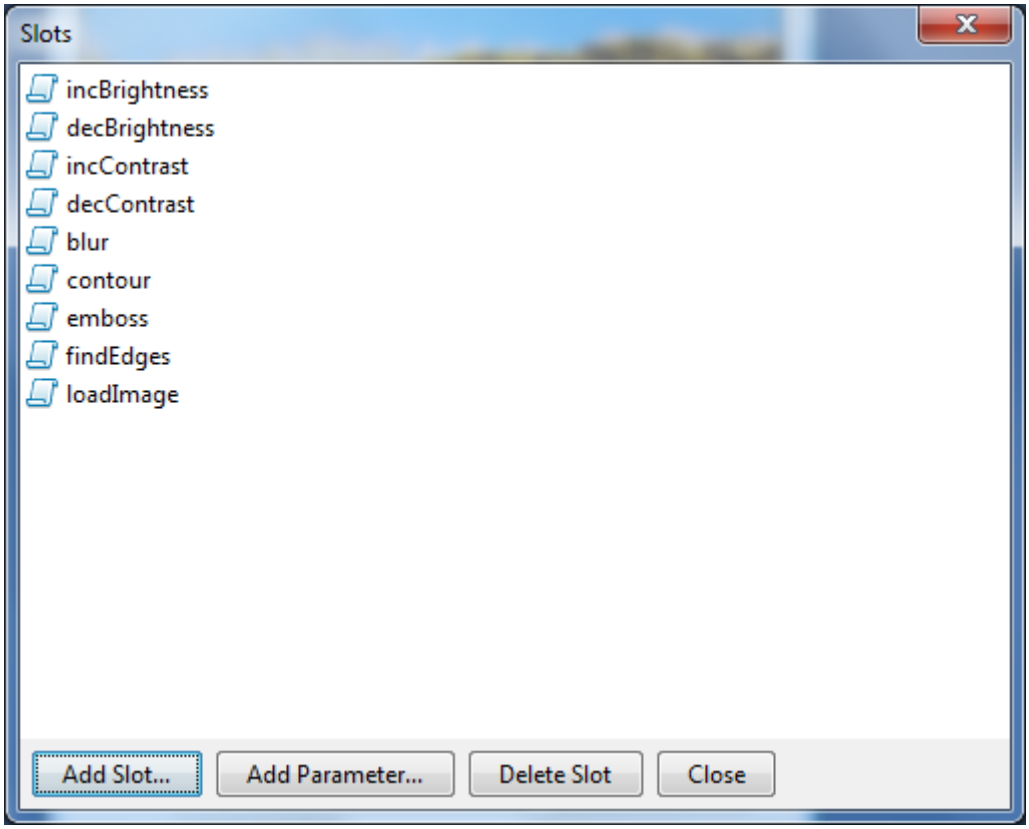

Figure D – The slots dialog showing 9 slots in the image processing application.

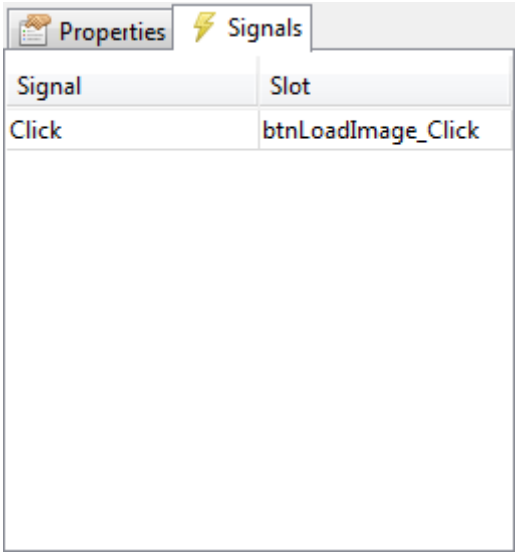

Figure E – The signal inspector showing a slot mapping assigned to a click signal.

Storing the entire event scheme in an XML file simplifies the client application as slots do not have to be programmatically hooked up to signals. Instead the developer simply needs to implement each of the slot functions and pass them to the API library in a single method call. Similarly parameter bindings are handled entirely within the XML file and API library meaning that no additional code is required to obtain the required values from elements. The major drawback is that only a predetermined list of data types is available for slot parameters. It is not possible to implement slots requiring custom data types as arguments.

Another downside is that only slot input is handled by the library, not output. Slot methods must handle all desired functionality including any manipulation of elements. Return values cannot be used as the API library simply ignores them.

4.2.5.3 Results

- The XML format enables a wide variety of slot methods to be encoded.
- Custom data types cannot be used which may prove restrictive for certain use cases.
- Slots and parameters can be mapped in a simple high level manner within the editor.
- Handling output would be advantageous as the entire event mechanism could be encoded in XML and handled within the API library.

4.2.6. Facilitate High Levels of Customisation, Flexibility and Expandability

4.2.6.1 Setup

This experiment aims to evaluate two closely related requirements:

- Create a flexible system built for expandability.
- Maintain a simplistic, consistent and lightweight approach.

This experiment evaluates the extent to which the system as a whole facilitates future extension and whether the current implementation maintains a consistent and simplistic approach. The overall aim is to identify the boundary after which it becomes impractical to use the system over directly hard coding interfaces.

4.2.6.2 Execution

Quantifying the flexibility of the system is a challenging problem particularly when the lack of rigorous testing is considered. As such a qualitative approach is taken whereby the various technologies and practices used are evaluated.

The example applications created for this evaluation demonstrate that the system can be used to create a diverse range of applications. One of the principle limitations of the entire system is the element set. In order to ensure universal compatibility only a prescribed set of elements can be used. Disregarding elements which are only available in a subset of GUI toolkits limits the flexibility of the system however allowing elements on a per-framework basis would undermine one of the foremost principles of the project. Solutions to this issue are discussed further in 5.1.2 Enhanced Element Set. The internal functionality of slot methods within client applications is not limited at all by the system. The functions performed by slots are bounded solely by the programming language and platform.

One of the founding principles of XML is facilitating extensibility [28]. New element types can be introduced to the file format without requiring excessive effort as long as the editor and all API libraries are updated to handle the new element. Adding new elements to the editor and API library is a simple case of extending from the correct base class (`XElementBranch` or `XElementLeaf`) depending on the type of element. Although

documentation for the system is lacking the required variables, methods and functionality of new elements can be easily based on existing elements of the same type. The use of lists for storing element properties within the editor removes the requirement for implementing serialisation/deserialisation logic.

Throughout the implementation a simplistic and lightweight approach has been adopted. The API library demonstrates how a minimalistic approach has been taken. All of the required functionality has been implemented using the smallest number of methods possible whilst still maintain separation of concerns. The file format does not include any unnecessary data. For example property data types are determined within the editor or API library rather than being specified within the file.

A consistent naming scheme for properties is used throughout the system mitigating the risk of confusing or redundant terminology. For example the *text* property refers to the inner text of any element including windows, text boxes, labels and buttons. This avoids confusion often found in certain GUI toolkits where *title*, *text*, *label* and *caption* are often used to refer to the text component of an element. Similarly conventions are used for class naming. Any class which belongs to the system is prefixed with the letter *X*. This avoids problems such as name conflicts as some platforms may already contain classes with names such as `Button` or `Slot` (these classes would be named `XButton` and `XSlot` instead). This also avoids the complication of using namespaces particularly for languages which do not support them such as C.

It is not possible to identify a concrete boundary after which the system becomes infeasible for a given use case. For certain purposes such as generic, linear applications the system performs well. The system has also proven to be feasible for creating certain advanced applications such as those which require advanced interaction with other libraries. Areas in which the system may demonstrate inadequate performance include resource intensive processing and applications requiring dialog boxes or child windows. Development of applications requiring highly specific, non-standard behaviour from the GUI toolkit may not achievable within the system.

4.2.6.3 Results

- The system can be used in a versatile range of applications.
- The limited element set may decrease flexibility however it helps to fulfil the more important goal of universality.
- Slot functionality is not specifically limited by the API library.
- The use of XML and dynamic property lists facilitates and greatly simplifies expansion of the system.
- A minimalistic approach simplifies the system hence lowering the barrier to entry.
- Consistency and conventions help to simplify the system use and mitigate misunderstanding.

- The system can be used to create a wide variety of applications however there is no clear boundary separating feasible and infeasible applications.
- The system is not suitable for applications requiring non-standard elements or behaviours.

4.2.7. Produce Example Applications

4.2.7.1 Setup

This experiment demonstrates the core functionality and versatility of the system via an example application. The following three initial requirements are evaluated:

- Produce a small number of example applications.
- Allow both small/trivial and larger scale/complex applications to be created.
- Facilitate a high level of GUI customisation.

The example application demonstrates a fairly unique application in addition to showing how the API library can interoperate with existing Python libraries. The example also demonstrates how a large number of elements can be included within an interface and how DOM style API methods can be used to implement the required functionality.

4.2.7.2 Execution

To demonstrate some of the core functionality and versatility of the system a basic image processing application was created. This allows a user to load an image from a file and apply basic filters to it using the Python Imaging Library. The entire interface was created using the Interface Editor without any manual editing of the XML file needed. The `XImageBox` accessor and mutator methods handle encoding and conversion of image formats allowing images in PIL format to be used interchangeably with the wxPython image format.

The application suffers from stability issues when large images, typically above 1MB, are processed. This is due to the threading issues as mentioned previously (see 3.2.4.5 Display method). A multithreaded approach would allow long running or resource intensive processes, such as image processing, to run in a separate thread to the main event loop.

Figure F shows screenshots of the application, Figure I in the appendices shows the XML code used for the interface and Figure J shows the Python code behind the application.

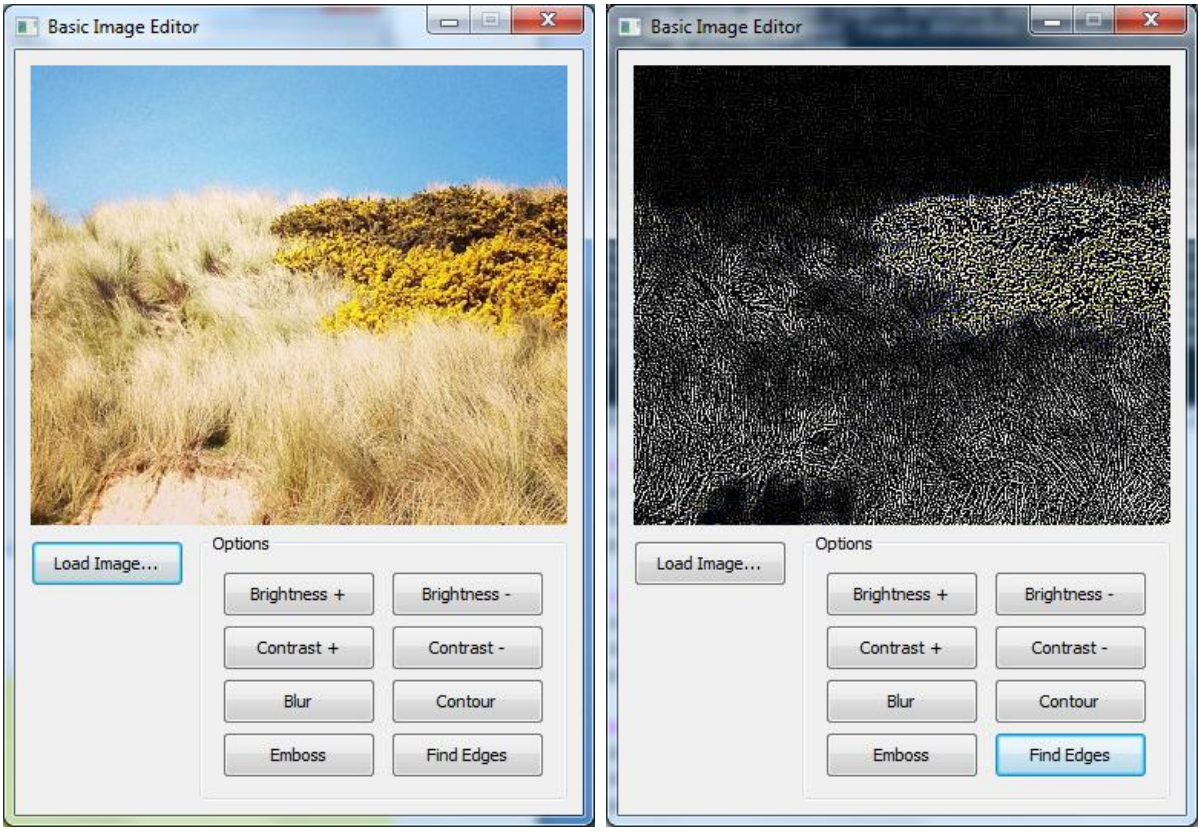

Figure F – The example image processing application showing initial state and "Find Edges".

4.2.7.3 Results

- The system succeeds in rendering a versatile range of elements.
- The editor allows a fully interactive interface to be created without requiring manual editing of the XML file.
- The API is simple to use requiring very few calls to load and render a fully interactive interface.
- The signals and slots mechanism provides a simple and effective approach to abstractly handling events.
- Performance issues exist that must be resolved before the system can be considered viable for more resource intensive applications.

4.2.8. Create and Document XML Schemas

4.2.8.1 Setup

This experiment evaluates whether XML schemas and documentation have been successfully produced for the system.

4.2.8.2 Execution

It was initially believed that the use of XML schemas would be fundamental to the project however during the course of system implementation it was found that they were not essential. The original intention was to use XML schemas to validate an interface file to

ensure it conformed to the required standards as it was loaded via the editor or API. In place of schema validation individual properties are validated as they are changed within the editor. Although this limits dynamicity it greatly simplifies the editor backend and allows for more intuitive user feedback. A significant investment of time would be required to implement a method of parsing a schema file and dynamically checking conformance on a per-property basis. A simpler approach may be to simply re-validate the entire XML file on every property change however this is inefficient and would require backtracking if invalid property values were provided. Additionally non-trivial re-engineering would be required to implement this as the editor does not generate XML in real-time. Instead it is generated from the ground up on each file save.

From the user's perspective XML schemas do not present an intuitive form of documentation particularly for users with a non-technical background. As such written technical documentation would be a more effective means of explaining system use and functionality over document schemas. The conventions and practices adopted within the system are fairly self explanatory and can easily be learnt by observing examples. The self-describing nature of XML further enforces this. Furthermore there are few situations which would require the user to manually edit the XML file. This would be largely undertaken by advanced users who would likely posses the technical capability to instinctively understand the system. It is usually necessary to cater to the majority particularly when developing under a strict time schedule as with this project.

4.2.8.3 Results

- Although the system fails to satisfy this requirement it has not had a profound, negative effect on the overall functionality of the system and does not hinder usability to an observable degree.
- It may be beneficial to produce XML schemas for future releases of the system particularly as additional, more complex element types are supported.
- In the interests of community development and potential open sourcing a formally prescribed standard would prove invaluable. This would enable third party developers to create their own tools or libraries based upon the standard in much the same way that tools have been created to support development with HTML.

4.2.9. Accommodate a Variety of Abilities

4.2.9.1 Setup

This experiment aims to evaluate whether users of varying abilities are accommodated within the system. This is achieved by analysing the various demographics for which the system is relevant and justifying the extent to which the system would benefit them. Software design methodologies are considered to evaluate whether the system is particularly suited to certain ones and at which stages within the software development lifecycle it is beneficial.

4.2.9.2 Execution

The intuitive design of the system accommodates a large and varied user base. The principle demographic is expected to be experienced developers however the low barrier to entry allows other participants in the software development lifecycle to engage. The following table outlines some of the main users who could benefit from the system and to what extent.

User	Role	Justification
User Interface Designer	Captures requirements and creates basic wireframe designs based upon them.	Interfaces can be created at a high level without involving any low level technical knowledge.
Programmer	Implement the programmatic functionality of applications.	Slot methods are implemented in the client application with connections to signals being created in the editor.
Tester	Test whether an application meets requirements.	Various aspects of the system can be tested individually e.g. the interface can be tested separately from the functionality.
Researcher	Implement algorithms, evaluate data sets and interface with command line applications.	Quick and simple interfaces can be created allowing focus to remain on the functionality.
Student or Hobbyist	Create a wide variety of applications not necessarily professionally.	The system has a very small learning curve lowering the barrier to entry.

In addition to offering a viable alternative to hard coding user interfaces the system also doubles as a valuable Rapid Application Development tool particularly for the user design and construction phases. The second stage of the RAD methodology, known as the user design phase, involves interaction between analysts and users leading to the development of models and prototypes of all system processes, inputs and outputs [27]. This fits well with the functionality of the editor whereby all of the methods and inputs must be modelled as slots however outputs are not taken into consideration. The visual designer could be seen as a simple, high level prototyping tool with the added benefit of allowing prototypes to lead to fully interactive interfaces mitigating the need to replicate the design during production.

Although many of the features of the editor present a high level view of an application low level knowledge of the chosen programming language is still required in order to implement the actual functionality. The system offers compartmentalisation in that certain teams or individuals could work solely on the interface and user experience whereas others could work solely on the programmatic functionality within the client application.

4.2.9.3 Results

- The system allows users with a wide range of abilities to make effective use of the system however certain users may find it more useful than others.
- To fully implement an application lower level knowledge of the chosen programming language is still required in order to implement the slot methods.
- The system fits particularly well with projects adopting the RAD methodology however doesn't penalise users for choosing alternatives.

4.3 Summary

A number of key areas requiring improvement have been identified as a result of the evaluation experiments. Despite this the project has shown potential and with further development could provide a viable alternative to hard coding user interfaces within production environments. It has been shown that there is a somewhat blurred boundary after which the using system becomes infeasible. Certain constraints such as the closed nature of the API library and the lack of support for custom data types make applications requiring non-standard interaction with GUI elements or custom data types implausible. Recommendations for future improvements based on this evaluation are described in 5. Future Work.

5. Future Work

This section discusses future enhancements and extensions that could be made to the system. The outcomes of the evaluation highlighted many areas in which the system offers sub optimal performance or lacks certain features. As a result many of the recommendations suggested in this section are based upon the evaluation. Other recommendations are solely features which may be appealing to developers in order to improve the usability of the system or facilitate extra functionality. This section offers a succinct starting point for anyone looking to further build upon the system produced during this project.

5.1 Future Enhancements

It was found that some of the aims and objectives of this project could not be fully realised due to time constraints. The following 9 recommendations highlight some of the key areas in which the system is currently lacking covering both the editor and API library as well as the overall architecture. These are the areas of development which have been considered yet have not been acted upon due to the limited time available for project completion.

5.1.1 Additional API Libraries

Currently only a single API library has been implemented targeting the Python programming language. In order for this project to be considered truly universal it is necessary to produce API libraries for a variety of platforms and programming languages. The most rational approach would be to target the most appropriate, popular programming languages in order to cater for a large cross section of the development community. Java, C, C++ and C# account for around 50% of the most popular programming languages in use today and as such would be the most desirable languages to implement API libraries for [27]. Implementing libraries for traditionally non-desktop languages such as PHP may not be the best strategy with the current system architecture despite their popularity. This is discussed further in 5.1.3 Mobile Support.

5.1.2 Enhanced Element Set

The current element set includes most of the standard components required by a user interface like text boxes, buttons and labels. This could be extended to include the full array of components supported by each framework. This may introduce some consistency issues for example if certain elements are not natively supported in all pertinent GUI toolkits. One possible solution would be to implement composite elements within API libraries where the underlying GUI toolkit does not natively support the element. For example the Microsoft .NET framework supports a checked list box control which allows the user to check/uncheck list items [28]. The Java SWING toolkit does not include a native element with these

characteristics however it could be achieved by compositing checkbox and label instances within a panel and encapsulating the behaviour of the element as a whole within a class.

A further, more complex enhancement would be to allow users to define their own elements in a similar manner to user controls found in the Microsoft .NET Framework [29]. A new element could be created by the user by combining existing elements in the same visual manner used to create full interfaces. A similar XML representation could also be used. This would enable reuse of commonly occurring layouts, sticking to the DRY (Don't Repeat Yourself) principle, and would greatly expand the dynamicity of the system [30]. These custom elements could have user defined properties and signals although this may prove to be a non-trivial feature to implement within the current architecture.

5.1.3 Mobile Support

One of the most predominant computing trends in recent times is the increased popularity of portable devices such as tablet computers and Smartphones. Whilst the desktop computer remains an important platform an increasing number of consumers are purchasing Smartphones and tablet devices. Around 50% of the UK population currently own a Smartphone or tablet computer [31]. Naturally it would be sensible to extend the system to work on common portable device operating systems such as Apple iOS, Android and Java ME. Implementing libraries for these 3 operating systems alone would provide support for approximately 95% of all Smartphones and tablets [32].

Many of the elements within desktop GUI toolkits have a similar counterpart within each of the aforementioned mobile platforms. For example the Android API includes input controls such as buttons, check boxes and text fields which have much the same behaviour as their counterparts in the wxWidgets toolkit [33]. Arguably this may be one of the most challenging enhancements to fulfil due to the behavioural differences between desktop computers and mobile devices. Whilst rendering interfaces on mobile platforms may not prove particularly challenging re-engineering the slots and signals subsystem may turn out to be a non-trivial task. This is largely due to the discrepancies between events within a desktop environment and events on mobile platforms. The Android API defines many events which are non-existent or uncommonly used within desktop applications such as touch and gesture events. Supporting events of this kind whilst maintaining backwards compatibility with desktop environments would require significant research into each platform in addition to major re-engineering of the existing system. A simple solution would be to limit mobile implementations of the system to use only the events available within the desktop APIs. Whilst this would make the implementation simpler it would greatly impact the user experience by imposing limitations on interactivity.

5.1.4 Fluid Layouts

Fluid layouts are a concept originating from web design whereby the majority of elements on a webpage are assigned percentage sizes instead of fixed sizes [34]. This allows webpages to dynamically adjust to the user's screen size creating a more user friendly and visually attractive interface. Currently only fixed positioning and sizing is available within the system meaning that an interface will be rendered in the same manner regardless of device. The support of percentage based sizing and positioning would allow fluid interface layouts to be created increasing the robustness of the system and facilitating future developments within the market such as increasing screen sizes. It would also allow interfaces to be more accurately rendered across multiple devices such as tablets and Smartphones.

5.1.5 Security

Currently interface files are stored in a simple, plain text XML format. Whilst this makes creating and maintaining interfaces more straightforward it also introduces a potential security flaw. If an end user were to access an interface file they would be able to freely edit it potentially changing much of the behaviour of the client application. This could allow users to bypass security mechanisms, such as login screens, or inadvertently break parts of the client application.

One solution to this problem would be to generate checksums of interface files through the editor or a command line application for advanced users. The outputted checksum could then be verified within the API library by implementing a method such as `VerifyIntegrity(checksum)` to ensure that the file has not been modified. Whilst solving the security issue this naïve approach may introduce new problems such as limiting the client application to using a single interface file. This may prove particularly troublesome when a new version of an interface file is produced as it would also require a new version of the client application to be distributed including the new checksum.

5.1.6 Multi-threading

As described previously the use of a single threaded approach within the API library can, in some cases, greatly impact the performance and responsiveness of the interface (see 3.2.4.5 Display method). Introducing multi-threading capabilities to the API library would alleviate most of the current performance issues and addition to allowing the system to be used in a wider variety of resource intensive scenarios such as scientific computing. A substantial number of alterations would need to be made to the API library to implement this however ample documentation and support is available for the wxPython toolkit [21].

As a further extension multithreading could be built into the signals and slots mechanism to allow certain slots to be run in separate threads or allow signals to trigger multiple slots

running in parallel, synchronised threads. This could facilitate additional future expansions such as distributed processing (see 5.1.7 Network Integration).

5.1.7 Network Integration

The original design goals of XML include simple usability over the internet [35]. This could be leveraged within the system to provide a wide range of network connected services. The most obvious use case would be automatic updates. On launching a client application or during idle time an update server could be contacted to check for newer versions of the interface. The new version could be automatically downloaded and put into use without requiring modification or recompilation of the client application. A local copy of the interface file would be stored for the sake of redundancy in case a network connection is not available.

Additionally distributed processing features could be integrated into the system. This could allow slow running processes to be distributed across a local network or across the internet using technologies such as Apache Hadoop [36]. This could enable advanced scientific and graphical processing such as distributed compilation as utilised by Valve Corporation's Source Engine for compiling game maps across a local network [37]. The concept of server-side slots could be introduced whereby signals could invoke processes or methods on a remote machine in a similar manner to Java Remote Method Invocation (RMI), possibly making use of it [38].

5.1.8 Live XML Serialisation

Many popular Integrated Development Environments (IDEs) feature live translation between code and a visual representation. For example the Eclipse IDE, when used with the ADT (Android Development Tools) Plugin, provides seamless, two-way conversion between visual, drag-and-drop based interface designs and XML. This allows advanced developers to quickly move between using the visual designer and hard coding XML. It also allows novice developers to see how changes in the visual editor are reflected within the XML. This is not possible in the current system as the XML representation of an interface is only generated once the document is saved and subsequently is not stored outside the scope of function. In order to efficiently implement this functionality the XML structure would need to be held in memory throughout the duration. Elements and attributes would need to be selectively added, removed and modified using the DOM style techniques found within the JDOM library. Synchronisation would need to be carefully implemented as to avoid discrepancies between the visual representation and XML. This would be required to work bidirectionally so that changes to either the visual representation or XML would be accurately reflected in either view.

5.1.9 XML Schemas

As mentioned previously no XML schemas currently exist for the XML interface format (see 4.2.8. Create and Document XML Schemas). XML schemas are useful for validating documents to ensure they conform to the prescribed format. Validation could be used in both the editor and API library to ensure that documents are valid before attempting to parse them in order to avoid errors or invalid state. Although the interface editor is unlikely to allow invalid XML files to be created advanced users may wish to manually write interface files. This is prone to human error such as typing mistakes or misunderstanding of syntax.

5.2 Summary

A number of strategic enhancements to the system have been proposed in this section which present a viable starting point for future development. Some of these enhancements are fairly trivial to implement however others require a certain degree of system re-engineering or more comprehensive analysis before commencing. Solutions to some of the defects exposed within the evaluation have been addressed with plausible solutions to render the system more robust and universal. Other features have been suggested which are not critical but would offer developers more compelling reasons to use the system.

6. Conclusions

This section offers a summative and objective conclusion of what has been achieved throughout the course of the project. A number of aims and objectives were outlined within the Interim Report. Each of these aims and objectives will be considered and the extent to which they have been fulfilled described. The technical achievements of the project will be discussed in addition to justification as to whether the project offers a viable alternative to traditional methods of creating user interfaces.

One of the original aims was to produce XML schemas describing the required format for interfaces. Due to time constraints this was not achieved however it did not have a significant impact on the project. The goal of using schemas is to ensure that a common standard for XML documents is adhered to. Whilst no technical schemas exist the principles of standardisation have still been followed. Additionally the system demonstrably shows that layouts can be accurately constructed from an XML file across all major desktop platforms.

It was initially envisioned that two API libraries would be developed for both the Java and Python programming languages. Due to limitations on time only one API library was implemented targeting the Python language. This was essentially an arbitrary decision as the same style of API could be implemented in any language. The remaining objectives relating to the API have been fully achieved. A lightweight API has been implemented which allows XML user interface definitions to be loaded and rendered with a minimal code footprint. Certain aspects of the Document Object Model have been reproduced within the API allowing a large degree of programmatic control over the interface.

A visual GUI editor has been successfully implemented as per the original aim. It allows interfaces to be designed in an interactive, drag-and-drop fashion appealing to both novice users and advanced users. Full interfaces and associated interactivity can be developed almost entirely within the editor with only slot functions requiring implementation in the client application. No explicit constraints are imposed on manually editing XML files allowing advanced users the freedom to create or edit interfaces textually. A diverse range of elements is offered allowing a high degree of customisation achievable through abstract properties.

A small number of example applications were produced satisfying the original objectives. These demonstrate the full range of elements available and exhibit the flexibility and diversity of the system.

The envisaged merits of the project described in the Initial Plan and Interim Report have been realised and a substantial starting point for continued development has been suggested. Overall the project has fulfilled the principle goal of allowing cross platform user interfaces to be described in XML format.

7. Reflection on Learning

This project was one of the most substantial pieces of work I have completed throughout my university career. I found the entire process of designing and implementing a system from a simple specification an enjoyable and rewarding experience. I feel this project in particular had the additional benefit of allowing me to actually put the system into use by developing example applications. This project gave me the opportunity to develop and acquire new skills which will prove beneficial in a future career in industry.

One of the principle lessons I learnt through completing this project is time management. Developing a clear schedule with a number of definitive milestones before commencing the project ensured that my aspirations were achievable in the given time frame. It also provided me with a clear indication of project progress throughout and helped in identifying situations in which I had fallen behind or had missed a milestone. Time management is a beneficial skill to obtain holding particular value for potential employers. I have learnt the importance and significance of developing a time schedule before commencing a project.

Initial research into software development methodologies gave a good insight into which strategies are appropriate for certain projects. I feel that I have greatly enhanced my project management abilities and view of how professional software projects are undertaken through active adoption of the agile approach. This will be beneficial in any future career as a large and increasing number of employers are adopting the principles of the agile manifesto.

Regular meetings with my project supervisor helped me develop interpersonal and communication skills. Due to the complex nature of the project it can be difficult to explain the technologies and practices used. Conveying ideas to my supervisor helped to refine my technical knowledge in addition to increasing my ability to articulate complex concepts, a skill which is most desirable in industry.

The numerous technical obstacles I faced throughout the design and implementation helped to refine my knowledge of both the tools and technologies directly involved with the project and the wider high level concepts. One of the areas in which I feel I spent too much time researching was XML schemas. Although this ultimately proved superfluous it helped me to understand the true goals of schemas and when they are necessary. Researching various different areas within the field of computer science helped to give me a clearer idea of which areas particularly interest me and highlight ones which I may want to pursue a future career in.

Overall this project has been a fulfilling experience from which I have picked up a high level of technical knowledge. I have also developed many interpersonal skills which will prove invaluable in the future.

Table of Abbreviations

API - Application Programming Language

ASCII - American Standard Code for Information Interchange

DOM - Document Object Model

GUI - Graphical User Interface

HTML - Hypertext Markup Language

IDE - Integrated Development Environment

MVC - Model View Controller

PIL - Python Imaging Library

RAD - Rapid Application Development

SAX - Simple API for XML

UI - User Interface

UTF-8 - UCS Transformation Format - 8-bit

W3C - World Wide Web Consortium

XML - Extensible Markup Language

XUL - XML User Interface Language

Glossary

Element - A GUI widgets within an interface such as a text box or button.

GUI toolkit - A set of widgets for use within applications requiring a GUI. The functionality of the toolkit is exposed through an API. The toolkit also handles events, for example the clicking of a button of selected value change of a dropdown box.

Java ME - Java Platform, Micro Edition – a Java platform designed for embedded devices such as Smartphones.

JDOM - A Java implementation of the Document Object Model which uses XML parsers to construct a document [39].

Signal - An event generated by interaction with an element such as *click* or *text changed*.

Slot - A function called when a signal is fired.

Slot Mapping - A dynamic mapping of element properties to the parameters required by a slot.

Unicode – An industry standard for character encoding.

wxPython - A Python wrapper for wxWidgets. An alternative to the Tkinter toolkit (the de facto GUI toolkit for Python).

wxWidgets - A GUI toolkit written in C++ for creating cross platform user interfaces.

Appendices

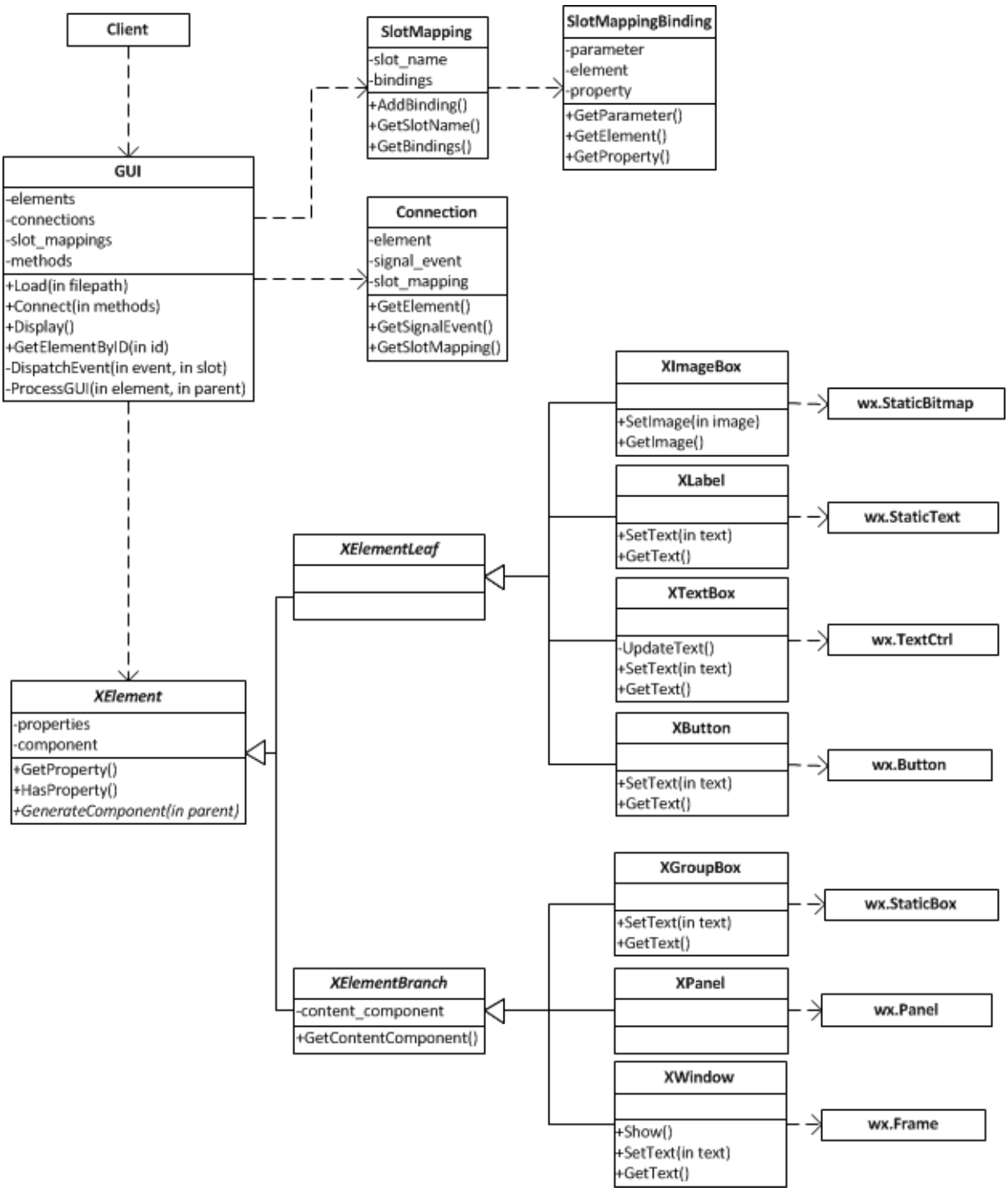

Figure G - Class diagram showing the relationship of data types in the API library.

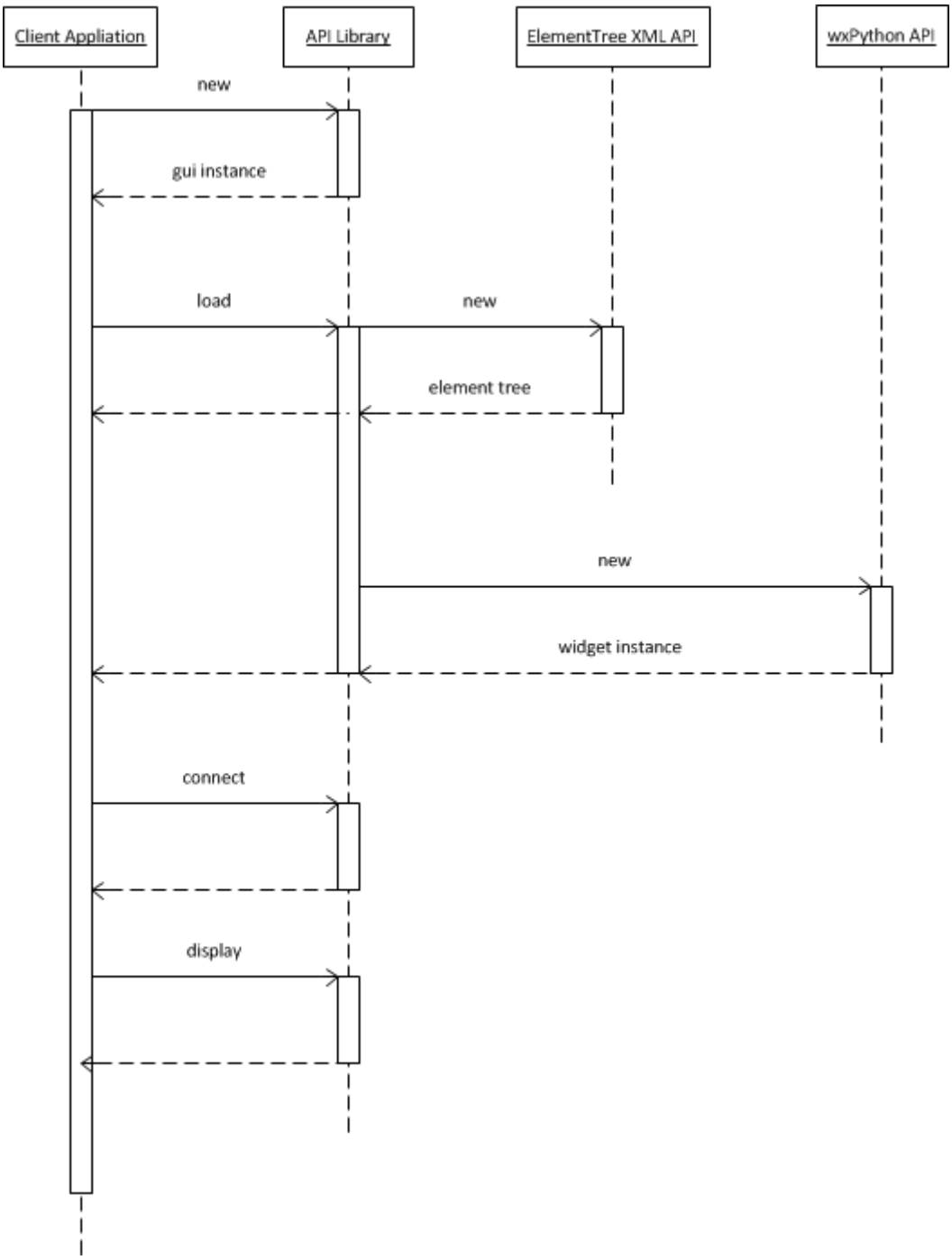

Figure H - Sequence diagram showing interaction between a client application, API library, ElementTree and wxPython.

```
<?xml version="1.0" encoding="UTF-8"?>
<xgui>
  <elements>
    <XWindow id="Window1">
      <width>370</width>
      <height>499</height>
      <text>Basic Image Editor</text>
      <resizable>False</resizable>
      <closable>True</closable>
      <minimizable>True</minimizable>
      <maximizable>False</maximizable>
      <elements>
        <XImageBox id="ImageBox1">
          <left>10</left>
          <top>10</top>
          <width>350</width>
          <height>300</height>

<imagesource>file:///C:/Users/James/Pictures/8b011e869ece11e28e8322000a1f9686_7.jpg
</imagesource>
          <halign>Middle</halign>
          <valign>Middle</valign>
        </XImageBox>
        <XButton id="btnLoadImage">
          <left>10</left>
          <top>320</top>
          <width>100</width>
          <height>30</height>
          <text>Load Image...</text>
        </XButton>
        <XButton id="btnEdge">
          <left>245</left>
          <top>445</top>
          <width>100</width>
          <height>30</height>
          <text>Find Edges</text>
        </XButton>
        <XButton id="btnEmboss">
          <left>135</left>
          <top>445</top>
          <width>100</width>
          <height>30</height>
          <text>Emboss</text>
        </XButton>
        <XButton id="btnContour">
          <left>245</left>
          <top>410</top>
          <width>100</width>
          <height>30</height>
          <text>Contour</text>
        </XButton>
        <XButton id="btnContrastDown">
          <left>245</left>
          <top>375</top>
          <width>100</width>
          <height>30</height>
          <text>Contrast -</text>
        </XButton>
        <XButton id="btnBrightnessDown">
          <left>245</left>
          <top>340</top>
          <width>100</width>
          <height>30</height>
          <text>Brightness -</text>
        </XButton>
        <XButton id="btnBlur">
          <left>135</left>
          <top>410</top>
          <width>100</width>
          <height>30</height>
          <text>Blur</text>
        </XButton>
        <XButton id="btnContrastUp">
          <left>135</left>
```

```
          <top>375</top>
          <width>100</width>
          <height>30</height>
          <text>Contrast +</text>
        </XButton>
        <XButton id="btnBrightnessUp">
          <left>135</left>
          <top>340</top>
          <width>100</width>
          <height>30</height>
          <text>Brightness +</text>
        </XButton>
        <XGroupBox id="groupOptions">
          <width>240</width>
          <height>175</height>
          <top>315</top>
          <left>120</left>
          <text>Options</text>
        </XGroupBox>
      </elements>
    </XWindow>
  </elements>
  <slots>
    <slot name="incBrightness">
      <parameters />
    </slot>
    <slot name="decBrightness">
      <parameters />
    </slot>
    <slot name="incContrast">
      <parameters />
    </slot>
    <slot name="decContrast">
      <parameters />
    </slot>
    <slot name="blur">
      <parameters />
    </slot>
    <slot name="contour">
      <parameters />
    </slot>
    <slot name="emboss">
      <parameters />
    </slot>
    <slot name="findEdges">
      <parameters />
    </slot>
    <slot name="loadImage">
      <parameters />
    </slot>
  </slots>
  <slot_mappings>
    <slot_mapping name="btnLoadImage_Click">
      <slot>loadImage</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnEdge_Click">
      <slot>findEdges</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnEmboss_Click">
      <slot>emboss</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnContour_Click">
      <slot>contour</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnContrastDown_Click">
      <slot>decContrast</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnBrightnessDown_Click">
      <slot>decBrightness</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnBlur_Click">
```

```
      <slot>blur</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnContrastUp_Click">
      <slot>incContrast</slot>
      <bindings />
    </slot_mapping>
    <slot_mapping name="btnBrightnessUp_Click">
      <slot>incBrightness</slot>
      <bindings />
    </slot_mapping>
  </slot_mappings>
  <connections>
    <connection>
      <element>btnLoadImage</element>
      <signal>Click</signal>
      <slot_mapping>btnLoadImage_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnEdge</element>
      <signal>Click</signal>
      <slot_mapping>btnEdge_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnEmboss</element>
      <signal>Click</signal>
      <slot_mapping>btnEmboss_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnContour</element>
      <signal>Click</signal>
      <slot_mapping>btnContour_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnContrastDown</element>
      <signal>Click</signal>
      <slot_mapping>btnContrastDown_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnBrightnessDown</element>
      <signal>Click</signal>
      <slot_mapping>btnBrightnessDown_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnBlur</element>
      <signal>Click</signal>
      <slot_mapping>btnBlur_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnContrastUp</element>
      <signal>Click</signal>
      <slot_mapping>btnContrastUp_Click</slot_mapping>
    </connection>
    <connection>
      <element>btnBrightnessUp</element>
      <signal>Click</signal>
      <slot_mapping>btnBrightnessUp_Click</slot_mapping>
    </connection>
  </connections>
</xgui>
```

Figure I - XML interface file used by the image processing application.

```
import xgui
from PIL import Image
from PIL import ImageFilter
from PIL import ImageEnhance
import wx

# Instantiate a GUI instance
gui = xgui.GUI()


# Define the slots

def loadImage():
	dlg = wx.FileDialog(gui.GetElementByID("Window1")._component,
message="Choose a file", defaultFile="", style=wx.OPEN | wx.CHANGE_DIR)
	if dlg.ShowModal() == wx.ID_OK:
		path = dlg.GetPath()
		gui.GetElementByID("ImageBox1").SetImage(Image.open(path))
		print path
	dlg.Destroy()

def incBrightness():
	im = gui.GetElementByID("ImageBox1").GetImage()
	en = ImageEnhance.Brightness(im)
	en.enhance(2.0)
	gui.GetElementByID("ImageBox1").SetImage(im)

def decBrightness():
	im = gui.GetElementByID("ImageBox1").GetImage()
	en = ImageEnhance.Brightness(im)
	en.enhance(0.5)
	gui.GetElementByID("ImageBox1").SetImage(im)

def incContrast():
	im = gui.GetElementByID("ImageBox1").GetImage()
	en = ImageEnhance.Contrast(im)
	en.enhance(2.0)
	gui.GetElementByID("ImageBox1").SetImage(im)

def decContrast():
	im = gui.GetElementByID("ImageBox1").GetImage()
	en = ImageEnhance.Contrast(im)
	en.enhance(0.5)
	gui.GetElementByID("ImageBox1").SetImage(im)

def blur():
	im = gui.GetElementByID("ImageBox1").GetImage()
	im = im.filter(ImageFilter.BLUR)
	gui.GetElementByID("ImageBox1").SetImage(im)

def contour():
	im = gui.GetElementByID("ImageBox1").GetImage()
	im = im.filter(ImageFilter.CONTOUR)
	gui.GetElementByID("ImageBox1").SetImage(im)

def emboss():
	im = gui.GetElementByID("ImageBox1").GetImage()
	im = im.filter(ImageFilter.EMBOSS)
	gui.GetElementByID("ImageBox1").SetImage(im)

def findEdges():
	im = gui.GetElementByID("ImageBox1").GetImage()
	im = im.filter(ImageFilter.FIND_EDGES)
	gui.GetElementByID("ImageBox1").SetImage(im)


# Load, connect and display the GUI

gui.Load("../image_editor.xgui")
gui.Connect([loadImage, incBrightness, decBrightness, incContrast, decContrast,
blur, contour, emboss, findEdges])
gui.Display()
```

Figure J - Python code for the image processing application.

References

[1] K. Beck and e. al., “Principles behind the Agile Manifesto,” [Online]. Available: http://www.agilemanifesto.org/principles.html. [Accessed 3rd May 2013].

[2] Sun Microsystems, Inc, “Code Conventions for the Java Programming Language,” Oracle Corporation, 1999 20th April. [Online]. Available: http://www.oracle.com/technetwork/java/codeconv-138413.html. [Accessed 4th April 2013].

[3] Mozilla, “Mozilla Developer Network - document.getElementById,” 10th August 2012. [Online]. Available: https://developer.mozilla.org/en-US/docs/DOM/document.getElementById. [Accessed 10th April 2013].

[4] World Wide Web Consortium, “Extensible Markup Language (XML),” 24th January 2012. [Online]. Available: http://www.w3.org/XML/. [Accessed 11th April 2013].

[5] Debian Project, “x64 Ubuntu : Intel® Q6600® quad-core - Computer Language Benchmarks Game,” [Online]. Available: http://benchmarksgame.alioth.debian.org/u64q/python.php. [Accessed 13th April 2013].

[6] Oracle Corporation, “Oracle and Java - Features & Benefits,” [Online]. Available: http://www.oracle.com/us/technologies/java/features/index.html. [Accessed 13th April 2013].

[7] Oracle Corporation, “LinkedHashMap (Java Platform SE 6),” [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/LinkedHashMap.html. [Accessed 15th April 2013].

[8] J. Hunter and B. Mclaughlin, “JDOM: Mission,” The JDOM Project, [Online]. Available: http://www.jdom.org/mission/index.html. [Accessed 17th April 2013].

[9] H. E. Wes Biggs, “Simplify XML programming with JDOM,” IBM, 1st May 2001. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-jdom/. [Accessed 17th April 2013].

[11] Unicode, inc, “What is Unicode?,” 12th December 2011. [Online]. Available: http://www.unicode.org/standard/WhatIsUnicode.html. [Accessed 18th April 2013].

[12] J. Hunter and B. Mclaughlin, “Format (JDOM v2.0.4),” The JDOM Project, 2012. [Online]. Available:

http://www.jdom.org/docs/apidocs/org/jdom2/output/Format.html#getPrettyFormat(). [Accessed 18th April 2013].

[13] J. Hunter, B. McLaughlin, D. Schaffer, P. Nelson, A. Rosen and R. Lear, “SAXBuilder (JDOM v2.0.4),” The JDOM Project, [Online]. Available: http://www.jdom.org/docs/apidocs/org/jdom2/input/SAXBuilder.html. [Accessed 19th April 2013].

[14] “Simple API for XML,” Oracle Corporation, [Online]. Available: http://docs.oracle.com/javaee/1.4/tutorial/doc/JAXPSAX.html. [Accessed 19th April 2013].

[15] Python Software Foundation, “TimeComplexity - Python Wiki,” 25th August 2012. [Online]. Available: http://wiki.python.org/moin/TimeComplexity. [Accessed 20th April 2013].

[17] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, S. Csomor, F. Montorsi and B. Petty, “wxWidgets: wxFrame Class Reference,” wxWidgets, 20th April 2013. [Online]. Available: http://docs.wxwidgets.org/trunk/classwx_frame.html. [Accessed 21st April 2013].

[18] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, S. Csomor, F. Montorsi and B. Petty, “wxWidgets: Window Styles,” wxWidgets, 20th April 2013. [Online]. Available: http://docs.wxwidgets.org/trunk/overview_windowstyles.html. [Accessed 21st April 2013].

[19] Python Software Foundation, “Glossary - Python v2.7.4 documentation - duck-typing,” [Online]. Available: http://docs.python.org/2/glossary.html#term-duck-typing. [Accessed 21st April 2013].

[20] G. v. Rossum and B. Warsaw, “PEP 8 - Style Guide for Python,” Python Software Foundation, 18th April 2013. [Online]. Available: http://www.python.org/dev/peps/pep-0008/. [Accessed 21st April 2013].

[21] “9.6 Private variables through name mangling,” Python Software Foundation, [Online]. Available: http://docs.python.org/release/1.5/tut/node67.html. [Accessed 21st April 2013].

[22] “Non-Blocking GUI - wxPyWiki,” wxPython, 18th June 2009. [Online]. Available: http://wiki.wxpython.org/Non-Blocking%20Gui. [Accessed 23rd April 2013].

[23] “document.getElementById - Document Object Model (DOM) - MDN,” Mozilla Developer Network, 10th August 2012. [Online]. Available:

https://developer.mozilla.org/en-US/docs/DOM/document.getElementById. [Accessed 23rd April 2013].

[24] “4.7.4. Unpacking Argument Lists - More Control Flow Tools - Python v2.7.4 documentation,” Python Software Foundation, [Online]. Available: http://docs.python.org/2/tutorial/controlflow.html#unpacking-argument-lists. [Accessed 24th April 2013].

[25] Mozilla Developer Network, “<input> - HTML - Mozilla Developer Network,” Mozilla Foundation, 2nd April 2013. [Online]. Available: https://developer.mozilla.org/en-US/docs/HTML/Element/Input. [Accessed 26th April 2013].

[26] Oracle Corporation, “What are the system requirements for Java 7?,” [Online]. Available: http://www.java.com/en/download/help/sysreq.xml. [Accessed 18th April 2013].

[27] Oracle Corporation, “Licensing and Distribution FAQs,” [Online]. Available: http://java.com/en/download/faq/distribution.xml. [Accessed 3rd May 2013].

[28] J. Milutinovich, “Frequently Asked Questions,” The JDOM Project, [Online]. Available: http://www.jdom.org/docs/faq.html. [Accessed 3rd April 2013].

[29] Python Software Foundation, “Python Setup and Usage - Python v2.7.4 Documentation,” [Online]. Available: http://docs.python.org/2/using/index.html. [Accessed 18th April 2013].

[30] Python Software Foundation, “Python 2.7 license,” [Online]. Available: http://www.python.org/download/releases/2.7/license/. [Accessed 3rd May 2013].

[31] wxWidgets, “Licence - wxWidgets,” [Online]. Available: http://www.wxwidgets.org/about/newlicen.htm. [Accessed 3rd May 2013].

[32] J. Cooney, “Developer UI,” 29th October 2006. [Online]. Available: http://jcooney.net/post/2006/10/29/Developer-UI.aspx. [Accessed 2nd May 2013].

[33] Mozilla Developer Network, “element - Document Object Model (DOM) - MDN,” Mozila Corporation, 1st May 2013. [Online]. Available: https://developer.mozilla.org/en-US/docs/DOM/element. [Accessed 2nd May 2013].

[34] Mozilla Developer Network, “Node - Document Object Model (DOM) - MDN,” Mozilla Corporation, 1st May 2013. [Online]. Available: https://developer.mozilla.org/en/docs/DOM/Node. [Accessed 2nd May 2013].

[35] World Wide Web Consortium, “Extensible Markup Language (XML) 1.0 (Fifth Edition),” 26th November 2008. [Online]. Available: http://www.w3.org/TR/REC-xml/#sec-origin-goals. [Accessed 30th April 2013].

[36] G. B. Shelly and H. J. Rosenblatt, in *Systems Analysis and Design*, Boston, Course Technology Inc, 2011, pp. 146-147.

[37] R. S. King, “The Top 10 Programming Languages - IEEE Spectrum,” IEEE Spectrum, October 2011. [Online]. Available: http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages. [Accessed 29th April 2013].

[38] Microsoft Corporation, “CheckedListBox Class (System.Windows.Forms),” [Online]. Available: http://msdn.microsoft.com/en-us/library/system.windows.forms.checkedlistbox.aspx. [Accessed 29th April 2013].

[39] Microsoft Corporation, “Varieties of Custom Controls,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ms171725.aspx. [Accessed 29th April 2013].

[40] J. Atwood, “Coding Horror: Curly's Law: Do One Thing,” 1st March 2007. [Online]. Available: http://www.codinghorror.com/blog/2007/03/curlys-law-do-one-thing.html. [Accessed 29th April 2013].

[41] Ipsos MediaCT Germany, “Mobile Internet & Smartphone Adoption,” Google Inc., January 2011. [Online]. Available: http://services.google.com/fh/files/blogs/Final_Mobile_Internet_Smartphone_Adoption_Insights_2011v3.pdf. [Accessed 29th April 2013].

[42] NetMarketShare, “Mobile/Tablet Operating System Market Share,” March 2013. [Online]. Available: http://www.netmarketshare.com/report.aspx?qprid=8&qptimeframe=M&qpsp=170&qpch=350&qpmr=100&qpdt=1&qpct=3&qpcustomd=1&qpcid=fw619349&qpf=1. [Accessed 29th April 2013].

[43] Google Inc., “Input Controls - Android Developers,” [Online]. Available: http://developer.android.com/guide/topics/ui/controls.html. [Accessed 30th April 2013].

[44] K. Knight, “Fixed vs. Fluid vs. Elastic Layout: What’s The Right One For You?,” Smashing Media, 2nd June 2009. [Online]. Available: http://coding.smashingmagazine.com/2009/06/02/fixed-vs-fluid-vs-elastic-layout-whats-the-right-one-for-you/. [Accessed 30th April 2013].

[45] The Apache Software Foundation, “Welcome to Apache Hadoop,” 26th April 2013. [Online]. Available: http://hadoop.apache.org/. [Accessed 30th April 2013].

[46] Valve Corporation, “VMPI - Valve Developer Community,” 4th January 2013. [Online]. Available: https://developer.valvesoftware.com/wiki/VMPI. [Accessed 30th April 2013].

[47] Oracle Corporation, “Remote Method Invocation Home,” [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. [Accessed 30th April 2013].