Design Document

2011-02

Project Name: Durham Software Wind Tunnels

Team Number: 1

Team Members: Chris Salt, Amir Aliyu, Joe Rivett, Craig Condron, Andy Dykes

Document Information

Project Name:	Durham Software Wind Tunnels		
Prepared By:	Andrew Dykes	**Preparation Date:**	
Email / Phone:	andrew.dykes@durham.ac.uk		
Document Version No:	1.0	**Document Version Date:**	**02/24/10**

Version History

Ver. No.	Ver. Date	Revised By	Description
0.1	24-11-2010	Chris Salt	First layout of what is needed.
0.2	18-12-2010	Craig & Amir	Layout filled with descriptions
0.3	15-01-2011	Chris Salt	DFD added
0.3.5	17-01-2011	Craig Condron	Section 1.6 complete
0.4	18-01-2011	Andrew & Amir	Introduction finished
0.5	27-01-2011	Joe Rivett	Section 2.1 complete
0.6	6-02-2011	Amir Aliyu	Section 3.1 complete
0.7	6-02-2011	Joe Rivett	Section 3.2 complete
0.7.5	08-02-2011	Craig & Andrew	Section 3 complete
0.8	16-02-2011	Joe Rivett	All sections except 2.2 complete
0.9	21-02-2011	Andrew Dykes	Section 2.2 complete
0.9.5	22-02-2011	Andrew Dykes	Proof reading
1.0	24-02-2011	Craig Condron	Final proof reading

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide an in depth insight into both high and low level design of a wind-tunnel data analysis system. The document will link design decisions to the system requirements defined in the Requirements Document (SoftSpot, 2010) for the same project.

The document must be at a depth where it can be passed directly to the implementation team who can then accurately provide the specified solution. Consequently the document will contain a thorough and unambiguous description of all modules and the system architecture.

1.2 Scope

1.2.1 Requirement Scope

Any requirements specified in the requirements document which have priority: very high; high or medium will be contained in the final solution along with any basic functionality not stated in the requirements document which is required to offer working deliverables. In addition to this all low priority requirements will be considered depending on time and resources.

1.2.2 System Scope

The system will give a user a graphical user interface in which they can run prewritten programs for processing data recorded from a wind tunnel. The system will allow the user to create sequences (multiple programs to be run consecutively). The system will allow the user to specify files to be processed along with parameters relating to the processing. The system is not required to attempt error correction on the parameters entered by the user. The system is not required to take any functionality of the prewritten programs (It is only required to run the programs).

1.3 Changes to Requirements

Requirement Number	Change Description
F014	New Description: Provide ‘DropDown’ parameter selection as an alternative to typing.
F018	Add New Requirement: Provide option for user to give the output from other programs, earlier in a sequence as a parameter for a program.
F015	New Description: Provide ‘RealTime’ search results when searching for a program to run or add to a sequence.

1.4 Definitions, Acronyms and Abbreviations

IDE: Integrated Development Environment, This is a piece of software which provides comprehensive features to facilitate the development of software.

UI: User Interface, The software element which interacts with the user.

GUI: Graphical User Interface, this is a graphical representation of what state the system is in and what actions the user can carry out to Interact with this state.

Tooltip: a small pop-up text box, describing what the graphical component does, which is displayed when a user hovers over an item in a GUI.

XML: eXtensible Markup Language is a file format which is useful for storing structured information in a simple to read form.

DSW: Durham Software for Wind tunnels, these are the programs developed by durham university engineering department for manipulating data collected from a wind tunnel.

1.5 References

Java JTree, Swing Library, Oracle Corporation, http://download.oracle.com/javase/1.4.2/docs/api/javax/swing/JTree.html

Java JTabbedPane, Swing Library, Oracle Corporation, http://download.oracle.com/javase/1.4.2/docs/api/javax/swing/JTabbedPane.html

Earl Hunsinger, User Interface Design, or Making Stuff Easy to Use, 20.02.11 15:39

S. Burbeck, 1987, Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC))

Ian Sommerville, Software Engineering 9th Edition, pg 178- 193, 2011

SoftSpot, SoftSpot Requirements Document, 2010

1.6 Overview

This section will provide an outline of the remaining major sections of the document and what is contained within each section.

Section 2.1 - contains a description of what our system does through the graphical user interface and the development of the GUI design. Alongside the developments and changes made, the purpose of each component of the GUI is explained.

Section 2.2 – contains simple scenarios of use for certain tasks that our system will perform when complete. It covers the processes our system will perform and describes them through use case diagrams.

Section 3.1 – provides an insight into the architectural patterns and styles used when developing the system and the major design decisions taken. A high level overview of the system is also given, highlighting the responsibilities of each component.

Section 3.2 – contains class diagrams to show the intermodule dependencies and overall structure of the system.

Section 3.3 – covers the interprocess dependencies of the system and its dynamic behaviour through the use of sequence diagrams. The sequence diagrams highlight the communication between objects in the system. Activity diagrams have also been used to show the behaviour of major elements and the responsibilities of each component.

2 Interface descriptions

2.0 Interface Introduction

The following section details how the Graphical User Interface (GUI) will be produced and provides an overview of its components and features.

When designing the GUI the purpose will be to provide an elegant and simple user interface, which is familiar to some users (F017). In addition to these goals, we will use feedback obtained from interviewing stakeholders to generate a clearer and more concise view of what would be required of the User Interface. Furthermore, we will use guidelines attained from the principles of Human Computer Interaction (HCI) to decide on the following criteria our GUI should fulfill:

- Above all else, the GUI must be intuitive and easy to use
- The GUI should be well structured and aesthetically pleasing but not to the detriment of usability.
- The GUI should be created in a manner where it performs all the functions the user is capable of in the existing system.
- The GUI should provide some extra functionality not provided by the existing system.

2.1 Interface Descriptions

It is vital that the system has a user interface which will accommodate users of all experience levels (F016, NF001). We must come up with a system that provides the full range of functionality detailed in the functional requirements whilst also working to develop a GUI which vastly simplifies the operation of the wind tunnels.

The following subsection details the initial user interface design phase and how the designs develop based on functional requirements, user feedback and group opinions.

As we highlighted likely system functions in the Requirements Document, it became clear to us that in order to develop a system which is usable to all users it may be beneficial to take inspiration from the user interface of existing programs. This lead to the conception of F017: to use a simple GUI format which is familiar to some users. Media player programs (such as iTunes or Windows Media Player) all adopt a simple Playlist creation feature which allows users to simply drag and drop songs into a playlist and reorder the songs by dragging them vertically. Pursuing this inspiration, we have come up with a rough interface idea whereby the user could create a sequence of programs by dragging them from a programs list into a sequence list. This has given rise to a columnar approach to the user interface, and we will design the interface so that the majority of necessary functions are displayed on a single frame.

2.1.1 Prototype development

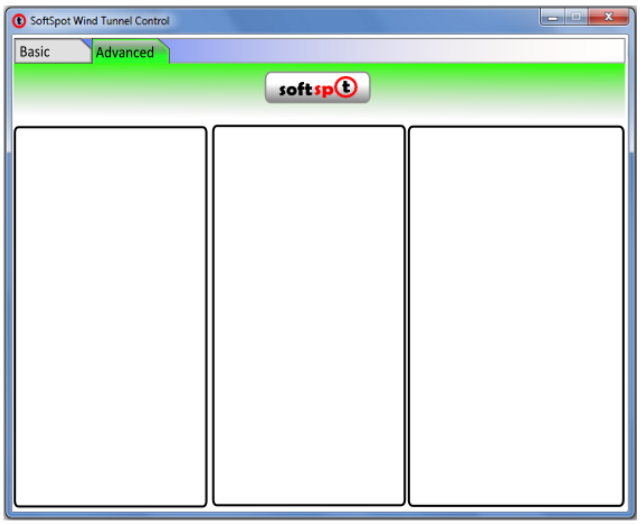


Figure 2.1.1 is a screenshot of a prototyped GUI we have developed. The design adopts three columns: the leftmost to display a list of runnable programs; the centre column to display the sequence of programs to be run; and the rightmost to display a list of parameters required by each sequenced program. The user interface allows for two views, one would be a simplified version for use by new users and the second would be a more complex view allowing more experienced users to handle more advanced features.

Figure 2.1.1: Initial aesthetic and layout

When this prototype was put forth to existing users it emerged that having dual functionality for new and experienced users is undesirable, as this presents another learning curve when migrating from the use of beginner features to those of the advanced view. We have therefore moved on to developing the user interface designs with only one structure, which realises the problem of how to present a GUI with all the desired features whilst also being simple enough for new users to use.

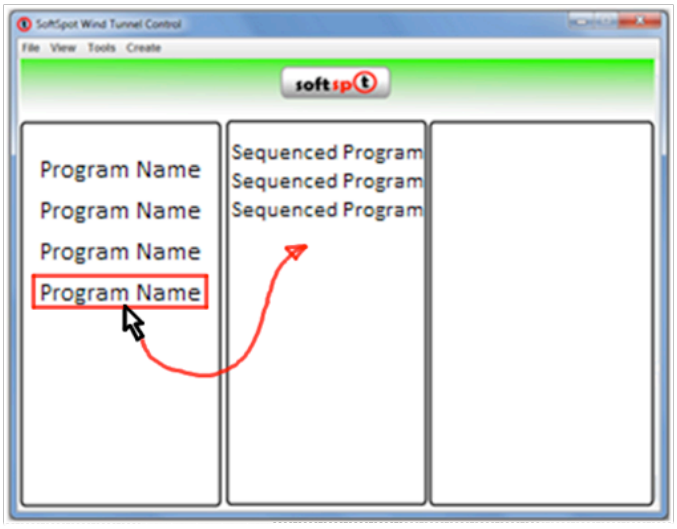


Figure 2.1.2: Developing UI based on user

During the aforementioned prototype demo the user confirmed satisfaction with the basic concept of a columnar design with simple drag and drop program operation. The programs in Figure 2.1.2 can simply be dragged into the sequence list. They are both displayed in a simple list format, with the sequence list having the ability to move listed items up or down to alter the order in which the programs are run.
This method of displaying programs for user selection is a very simple concept, however on reviewing the requirements the importance of displaying programs based on their category is fairly key. F008 specifies that programs must be divided into sub categories, allowing the user to quickly navigate to a program based on what they wish to accomplish. The programs must be displayed in two sub-categories, ("Data Logging Programs" and "Data Processing

Programs"), meaning that the programs must be displayed in separate categorised entities. The sketch in Figure 2.1.3 demonstrates two possibilities: one where the program names are displayed, still in list form, in two scroll boxes; and one where the programs are displayed in an expandable tree menu similar to those found in many programs to demonstrate hierarchy.

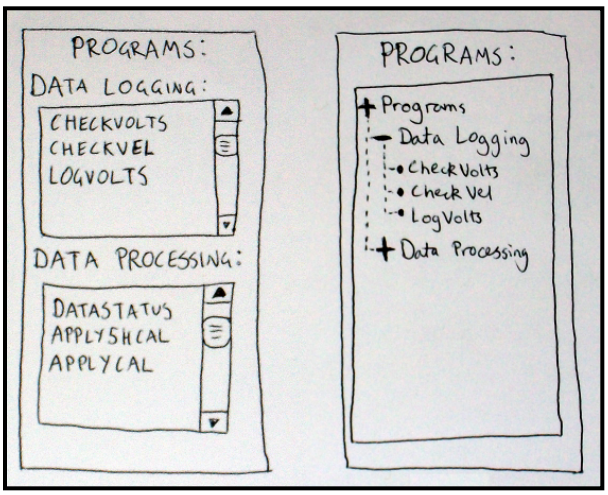


Figure 2.1.3: Sketches of potential methods of program categorisation

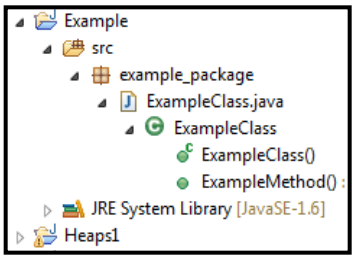


Figure 2.1.4: Example of tree menu in the Eclipse IDE

The hierarchical tree method means that our programs could be intuitively laid out by displaying a root node "Programs", with child nodes "Data Logging" and "Data Processing", each then with children (and tree leaves) stating a list of the program names implicit within each category. This feature can be realised in Java using its JTree Component (Java V1.6, Oracle Corporation) in the Swing library, which provides sufficient means for program list manipulation and ordering.

2.1.2 Program Parameters

The third vital column in our initial design is needed to edit the parameters used when running the programs. The programs take files as input variables upon execution, and it is vital that the user can edit the variables and values contained in these files. The parameters panel on the GUI must adapt based on which sequenced program is selected (F013) – this includes adapting based on the number of parameters taken in the program control (.in) file in addition to adapting to differing variable names.

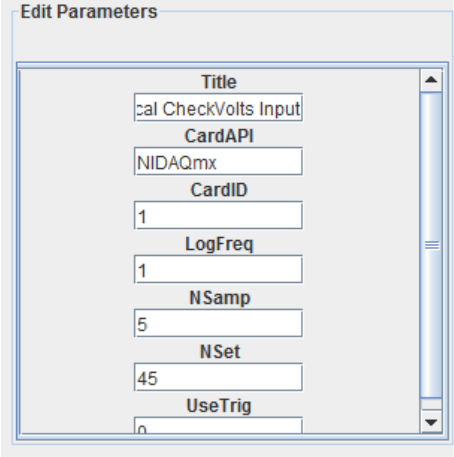


Figure 2.1.5: Initial Parameters panel layout

Figure 2.1.5 is an initial design for the parameters panel. The clear structure means that a novice user would be able to add parameters to the control file fairly simply, with variable names set above text fields for variable value input. However, this design has no support for editing other inputs required by some programs- it only allows for editing the parameters used in the control files. Some programs (mainly those used for data processing) require the user to specify further input files and the names of output files when initially executing the program. Figure 2.1.6 illustrates a tabbed method of allowing the user to edit these additional execution variables. In the diagram 3 tabs are shown: one for allowing the editing of the control file; one for specifying the location of an input data file and one for specifying the name of an output data file. This tabbed approach is increasingly used in public domain software and provides a simple means of displaying different information in the same area of a program window. We wish to keep the number of open windows in the program down to a minimum, giving a modular and easy to use solution, so we will pursue this tabbed idea and build it into our system.

In Java a Swing component called a JTabbedPane (Java V1.6, Oracle Corporation) allows the use of tabbed panels in one location on a frame. These tabs allow the use of tooltips (see illustration in Figure 2.1.6) so using a JTabbedPane could be further utilised to provide help information to the users (F012)

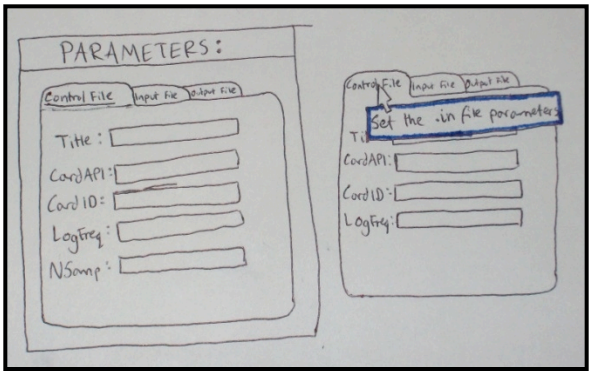


Figure 2.1.6: Tabbed structure for configuration of input files

2.1.3 Program Running Feedback

There must be some means on the program Graphical User Interface of giving the user some feedback with regards to the current state of running programs (F011). Currently the GUI consists of 3 columns, which leaves two possibilities for displaying runtime program information. The onscreen output of each program, which gives its running state as well as details of any encountered errors, could be displayed in a separate frame that works on top

of the current interface. Below the line this would give simpler control of program threads, but would also introduce more windows complicating the use of the system. On the other hand, a text area could be incorporated into the current frame design; simply displaying the current running status in a text area in some position adjacent to the existing components. We feel that this is the best option, simply because a novice user must find it as simple as possible to use the system (NF001) and the introduction of multiple windows would further complicate the learning process.

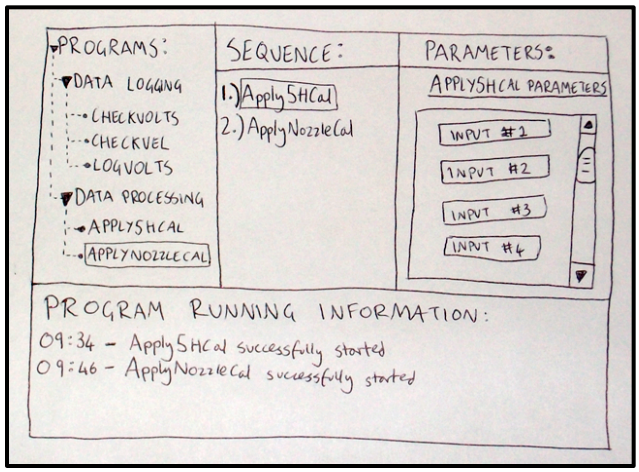


Figure 2.1.7: Adding a running state output text area to the current frame

2.1.4 Designing Help Functions

Each program in the manual issued by the Engineering department contains a program description along with a usage example. It would be very useful to new users if this information was accessible to them directly from the system, so as part of help system (F010, F012) we must find some way of displaying it. The simplest means of doing so would be to show the description text in a standard message box (Figure 2.1.8). We must decide upon the method by which the message box is invoked, such as whether it would be displayed when the user double clicks a program name or whether they follow some menu option (i.e. “File Menu/Help/About This Program”).

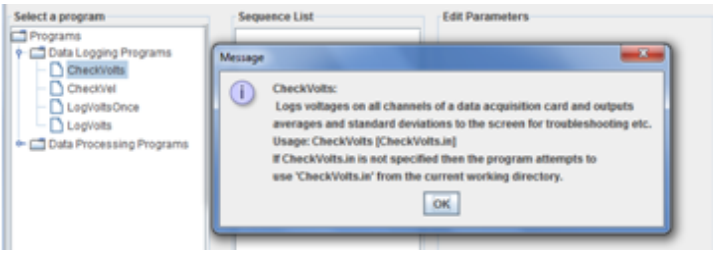
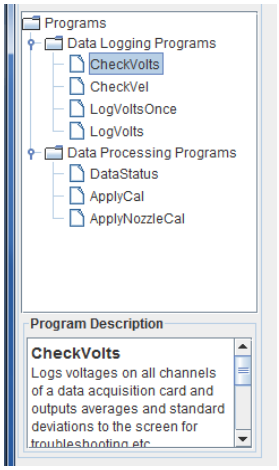


Figure 2.1.8: Displaying program descriptions in a message box

The other option is to follow the single window structure developed so far and add a text panel to the current frame. We could set a small text area into the frame below the sequence list, placing the program description virtually in the centre of the window.

However, on review of the current design (illustrated by the freehand sketch in Figure 2.1.7) we have further developed this concept. The design currently allows for a large amount of space for the program output text. The size of this text area can be decreased as programs do not generally output a large string of continuous text- thus potentially leaving more space in the lower panel of the current GUI. By taking this into account and thinking as a team about where the description information would be most conveniently displayed, we have arrived at the outcome shown in Figure 2.1.9. This would update to show a description whenever the user clicks a program in the tree menu, and means that the width of the program output panel now spans only two columns.



Aside from offering program descriptions, we must think about other methods which we can implement to help a novice user when using the program. As mentioned in relation to the JTabbedPane, it would be useful to use tooltips to describe to the user some helpful tips when they hover over a component. The parameters for the control file are something that could offer a large degree of confusion to any user, beginner or not, so it would be useful if tooltips were displayed when the user is inputting data into the parameters panel.

Figure 2.1.9: Final position of the description panel

2.1.5 Programs Search Function

The addition of the Program Description panel below the programs tree menu means that there is less space in the tree for program names to be listed, which becomes a problem when there are a large amount of programs imported into the system. It would be useful to implement some search method from which the user could quickly add a program to a sequence, without having to scroll through the full list. F015 states that auto-complete should be implemented when adding programs to a sequence, meaning that this search method must return results in real time as you type. The search bar must consist of a text input field and a button for cancelling the current search.
Figure 2.1.10 shows the design development of this feature, with the use of a standard button to cancel the search moving to the use of a custom icon.

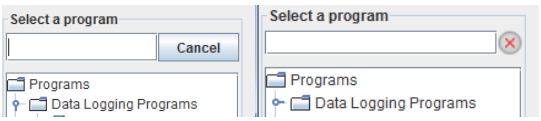


Figure 2.1.10: Developing the search bar

2.1.6 Sequence List

The sequence of programs added by the user will be displayed in a simple list format. We will implement a method of importing a sequence from file so that the user does not always have to create a commonly used sequence from scratch (F001). There must therefore also be some method of saving the current sequence, and the simplest means of allowing the user to initiate either of these features would be to add two command buttons to the GUI below the sequence list. This is shown in Figure 2.1.11.

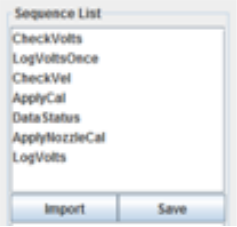


Figure 2.1.11: Sequence List

2.1.7 Running Programs

Once the user has created the desired sequence and edited all the required parameters, they will click a “Run” button at which point our system must create the final control files and initiate the execution of each selected program. The “Run” button must be strategically placed so that it is not accidentally pressed before the user has finalised the sequence. The ill-placing of important components is a commonly made mistake in UI development so we must work to avoid it (Earl Hunsinger, 2011). There are a few locations on the current GUI design which could accommodate a “Run Sequence” button, the logical possibilities for which are shown in Figure 2.1.12.

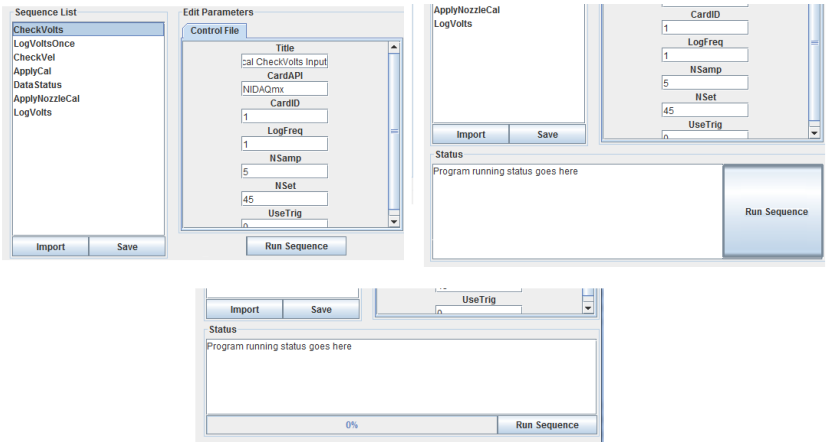


Figure 2.1.12: Developing the location of the “Run” button

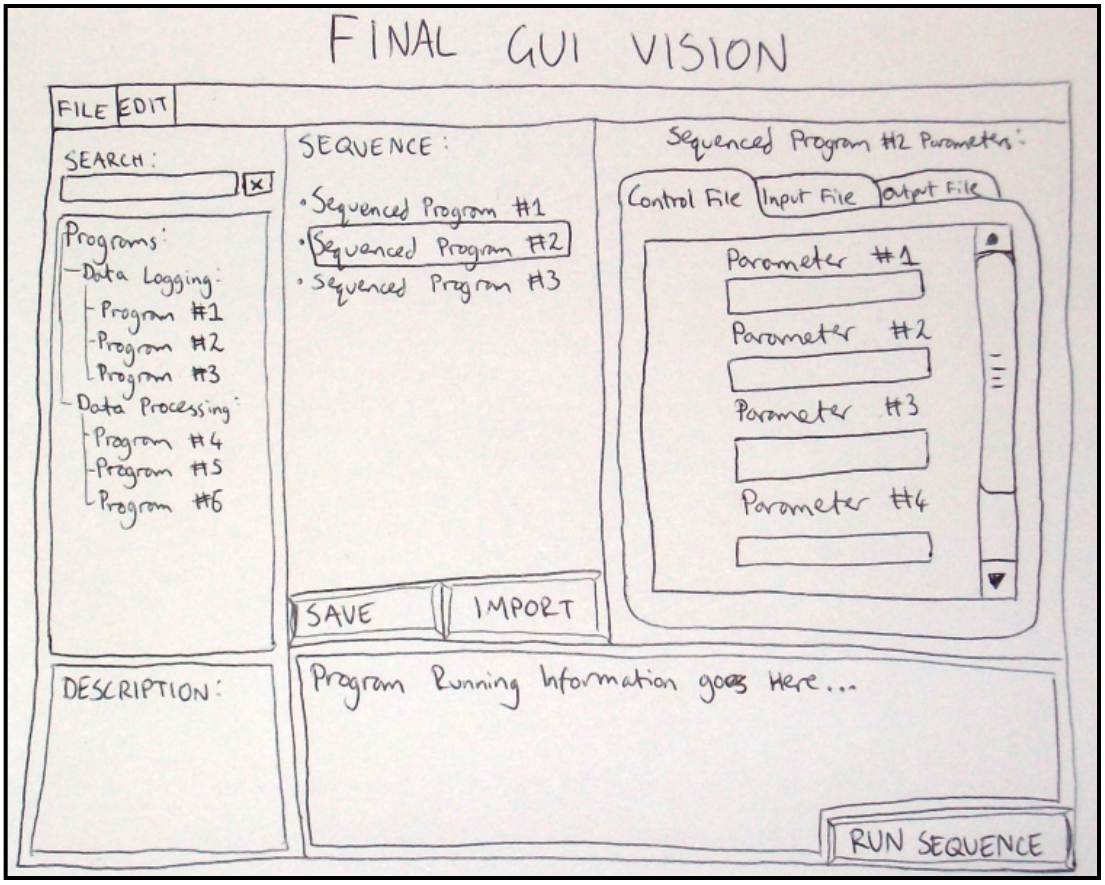
2.1.8 Add Program GUI

We will incorporate some means of allowing the user to add a program to the system so that it can be used as part of a sequence. The class Graphical User Interface will be a simple one containing only a set of input boxes in which the user can add the name, the type and the definition of the program. There will also be a set of input boxes allowing the input of program parameter requirements, including the parameter name, description and default value.
This GUI will be the only panel displayed to the user in a window separate from that of the main GUI.

2.1.9 Final Design View

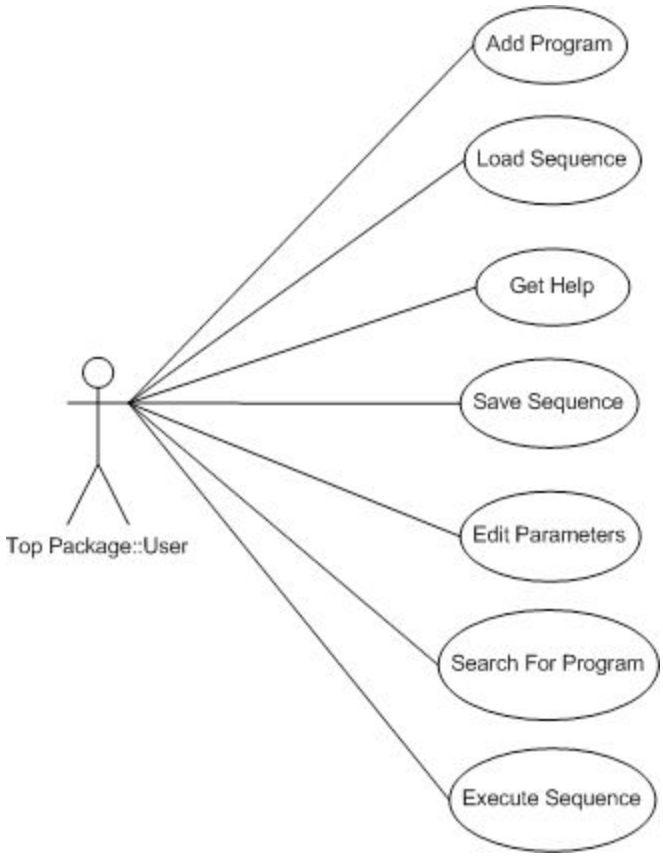
Figure 2.1.13 demonstrates the final design concept we have developed. It is such that all required information is displayed on a single window, meaning very simple operation and a very short learning period. We feel that this design successfully fulfils Functional Requirement F017, in that it provides a simple layout which would be familiar to some users both aesthetically and ergonomically.

Figure 2.1.13: Sketch of the final Graphical User



2.2 Process Descriptions

The following subsection provides brief details of what occurs when a user interacts with the system. UML Use Case diagrams offer a simple means of requirements discovery (Sommerville , 2011) and they are useful for demonstrating processes which must be contemplated at this point in the design phase. The use case diagram below is a simple illustration of the scenarios of use for which our final system must have functionality.



An overview of the processes involved in each use case can be seen below. If not in this section, a selection of these use cases are further decomposed with the use of sequence diagrams in section 3.2- Interprocess Descriptions.

2.2.1 Changing a parameter

When a program is clicked, the user sees a list of that program's parameters, with each parameter having a default value. If the user decides to change the value of a parameter, the system waits until the user has finished entering values and then saves the custom values for use when running the program.

2.2.2 Searching for a program

The user types in the name of the program they are looking for into the search text box. Each time a key is pressed, the System creates an array of programs that have a name that matches search criteria. The array is then sent to the GUI and the updated list is then displayed to the user. Figure 2.2.2 shows the process that will take place upon each keypress in the search input box.

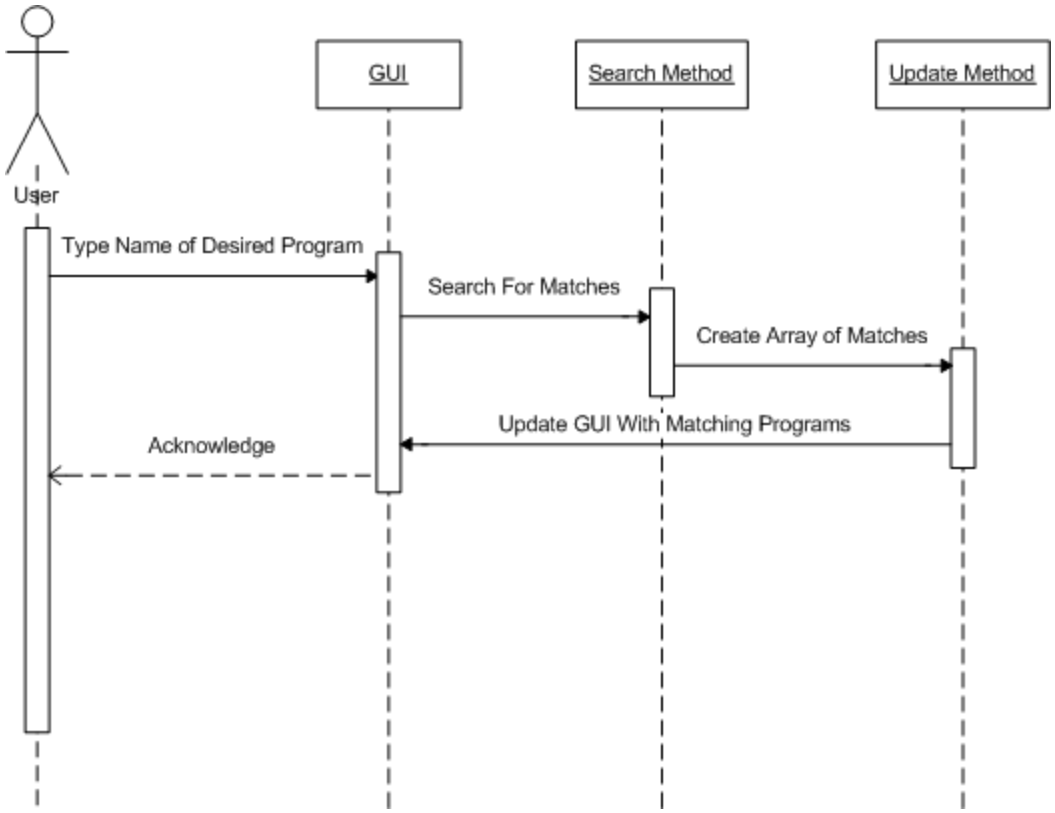


Figure 2.2.2- Sequence diagram showcasing search for a program

2.2.3 Getting help information

The user selects the type of program they want (such as “Data Logging”) and then selects the program they wish to use from within that subheading. The help information is then displayed in the “Program Description” panel. The user can then hover the mouse over a parameter name, at which point some help information regarding the parameter is displayed in the form of a tooltip.

2.2.4 Adding a program

The user clicks “Add Program” from within the “Create” menu. A separate window is displayed and the user is asked to provide information regarding the program being added. The user must enter the program name, type and description and then must proceed to enter parameter information. If they wish to add more parameters it will be possible to click “Add Parameter”, which will provide another set of inputs to specify information about another required parameter.

2.2.5 Create/Load sequence

The user will have two options when initializing a sequence. Firstly, they will be able to create a sequence by selecting the programs to run and dragging them into the sequence list. Secondly, the user can load a sequence by clicking the import button in the Graphical User Interface and selecting a pre-saved sequence.

2.2.6 Saving a sequence

When the user has completed developing their sequence, it will be possible to save the ordered programs and their set parameters for future use by clicking the “Save Sequence” button. When this is pressed, the user will be prompted to enter the save location for the sequence file, allowing a file to be created on disk detailing the sequence information.

2.2.7 Executing a sequence

The user creates/loads a sequence (see previous use case) and then will click the “Run Sequence” button. The system will then proceed to running the programs in order and relay state information back to the user via the GUI.
A detailed sequence diagram for this use case is available in Interprocess Dependencies (Section 3.2), Figure (3.2.1)

The diagram below (Figure 2.2.3) demonstrates data flow within the system.

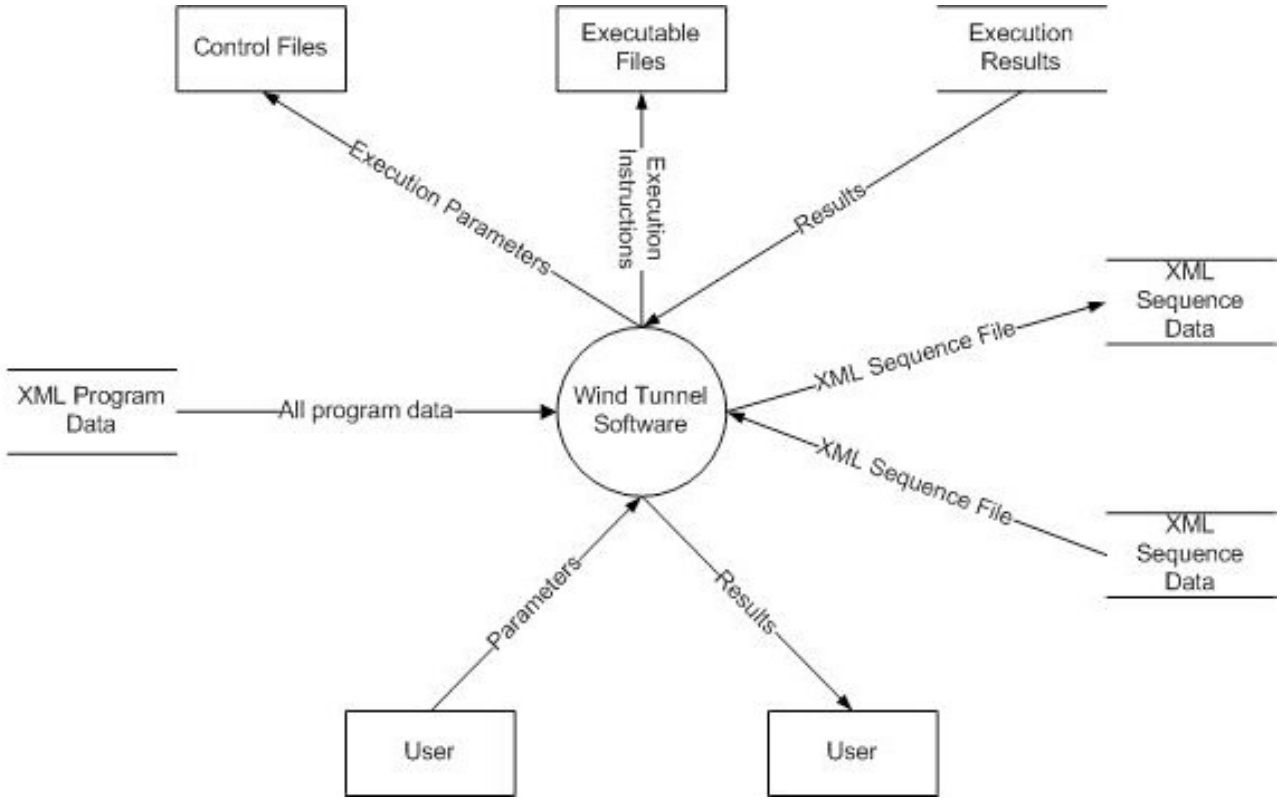


Figure 2.2.3- Context Diagram

3 Decomposition Descriptions

3.0 Architecture

3.0.1 Abstractions

The following section refers to design decisions at two levels of abstraction: Architectural Level and Design Level.

3.0.2 Architectural Pattern

The system is being designed using the MVC architectural pattern (S. Burbeck, 1987). This pattern defines the system as three distinct elements: Model, View and Controller. Specifically, in this system those three elements have the following roles:

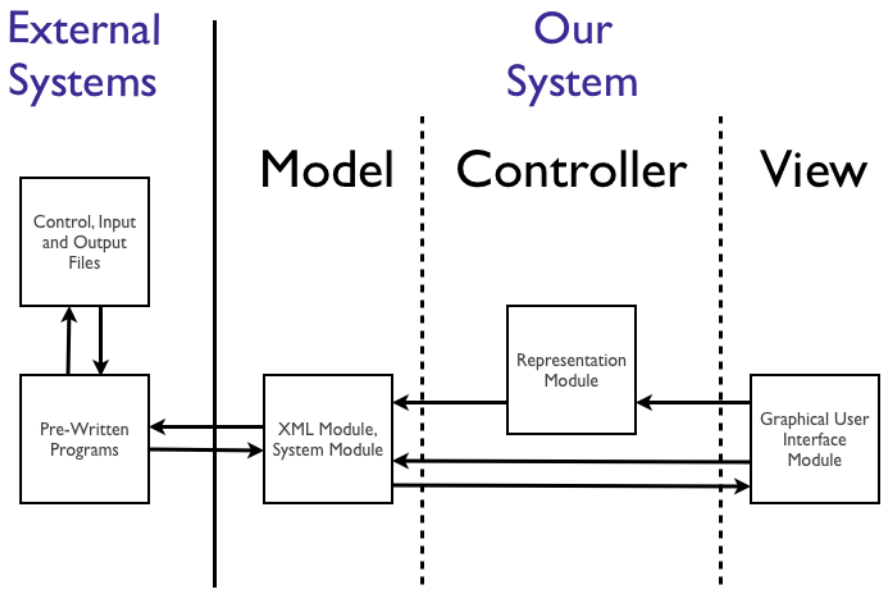


Figure 3.0.1- System representation of Model View Controller

The View - is the graphical user interface. The purpose of this interface is to give the user information about the state of any running programs as well as allowing them to initiate the running of programs. The view communicates with both the model and the controller. Communication between the view and the model is detailed below. In our system communication between the view and controller is limited to one direction only (from view to controller) as there is only one interface so no change in interface is necessary. The view sends messages to the controller relation to a users actions for the controller to interpret.

The Model – element of the system deals with all communication with the command line. The Model can communicate with both the controller and the view. The communication between Model and View is based solely around the state of the model. The view may request updates about the state of the model which are relaid to the view. The other communication between these elements similar except being initiated by a change of state rather than a request.

The Controller – is the element of the system which deals which interpreting and carrying out a users commands. The controller receives commands from the view which are interpreted before being passed on to the model.

3.0.3 Architectural Style

The system is being designed using the Data Flow architectural style (S. Burbeck, 1987). This style is characterised by the movement of data through the system, data can only move from the user to the manipulation modules and then back to the user (no data can be sent to the user without any input).

3.0.4 High level overview

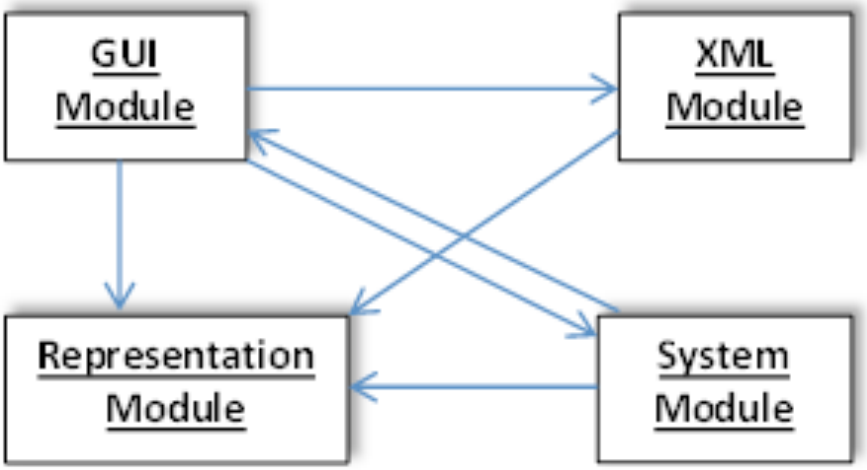


Figure 3.0.2: Block Diagram of the system

The GUI module will be responsible for displaying information to the user of the system, and interacting with the Representation module to save sequences of programs. The GUI module will relay output information provided by the programs ran via the System module. It will also interact with the XML module to create XML files for use when saving and loading sequences.

The XML module is responsible for creating, parsing and saving XML files relating to sequences, programs and parameters created in the Representation module and GUI module. This will be done using the GUI module to construct the sequences which will then be interpreted by the XML module to be saved for future use.

The Representation module will be responsible for creating the java objects that our system uses, which represent each program, sequence and the individual parameters for each program.

The System module will be responsible for executing the sequences passed from the GUI module and sending the information about the sequences and their status back to the GUI module, to be displayed for the user of the system. It will also send data to the Representation module so that objects can be created to represent the programs and parameters in any running sequence, and the output files they create.

3.1 Intermodule Dependencies

3.1.1 XML Module

The XML module integrates with the core module in multiple ways to provide program scalability. The XML module is designed to save important program information in secondary storage, providing users with a suite of regularly used programs and allowing them to add more programs, and further providing them with a standardized format to do so. In addition the XML module provides the user the ability to save a list of programs in a particular order in which they will be executed, thus allowing them to create sequences.

The XML module consists of 4 main classes, which shall now be described. The XML generator class interfaces with a GUI to allow the user to generate an XML file containing the program's data. The Parser class reads in an XML file and generates a Program object. The Sequence class, saves a list of programs in a specified order to a specified XML file format. The SequenceParser class, reads in and parses a Sequence XML file, storing the information in a sequence object. The module contains 2 interfaces, ParserInterface and GeneratorInterface. The interfaces helped with the design in that they allowed for a uniform list of method names to be implemented for classes which offered similar functionality.

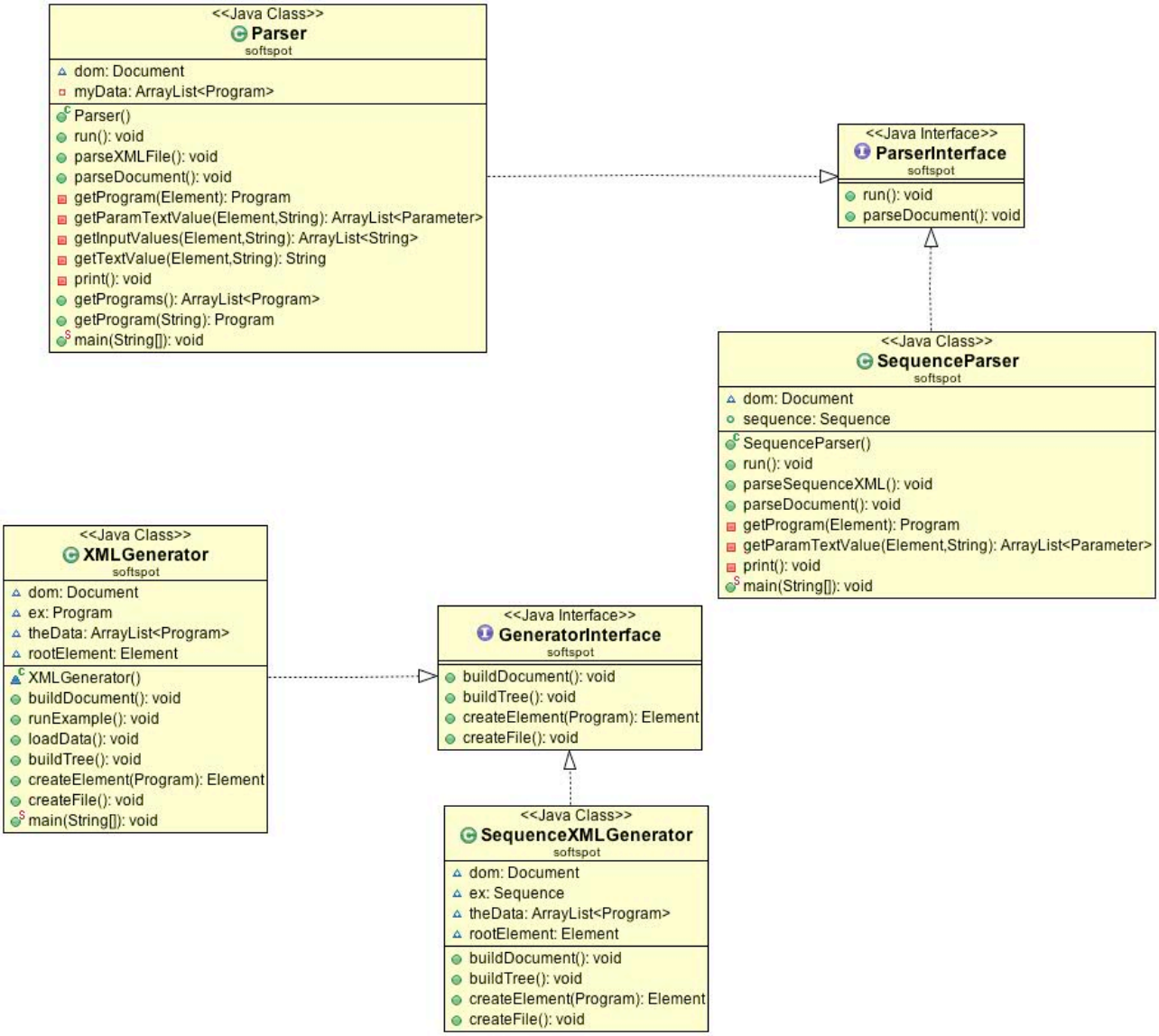


Figure 3.1.1- Class Diagram for XML module

3.1.2 System Module

The System module is designed to connect the command line with the GUI module and to create the in files necessary for the execution of the DSW. The System module consists of 4 classes. The Command class represents a command line argument, containing a path to a program and some parameters (.tec and .in files). The InFileCreator class creates an in file with a given name and value pair. The ExecuteShellCommand class goes through a list of command line arguments and executes them. Once execution has finished, the class sends communicates with the GUI module, asking it to update. The Execute class is used to create

the list of command objects which it executes using the execute method from the ExecuteShellCommand class. The execute class further allows the user to choose the location of a programs output.

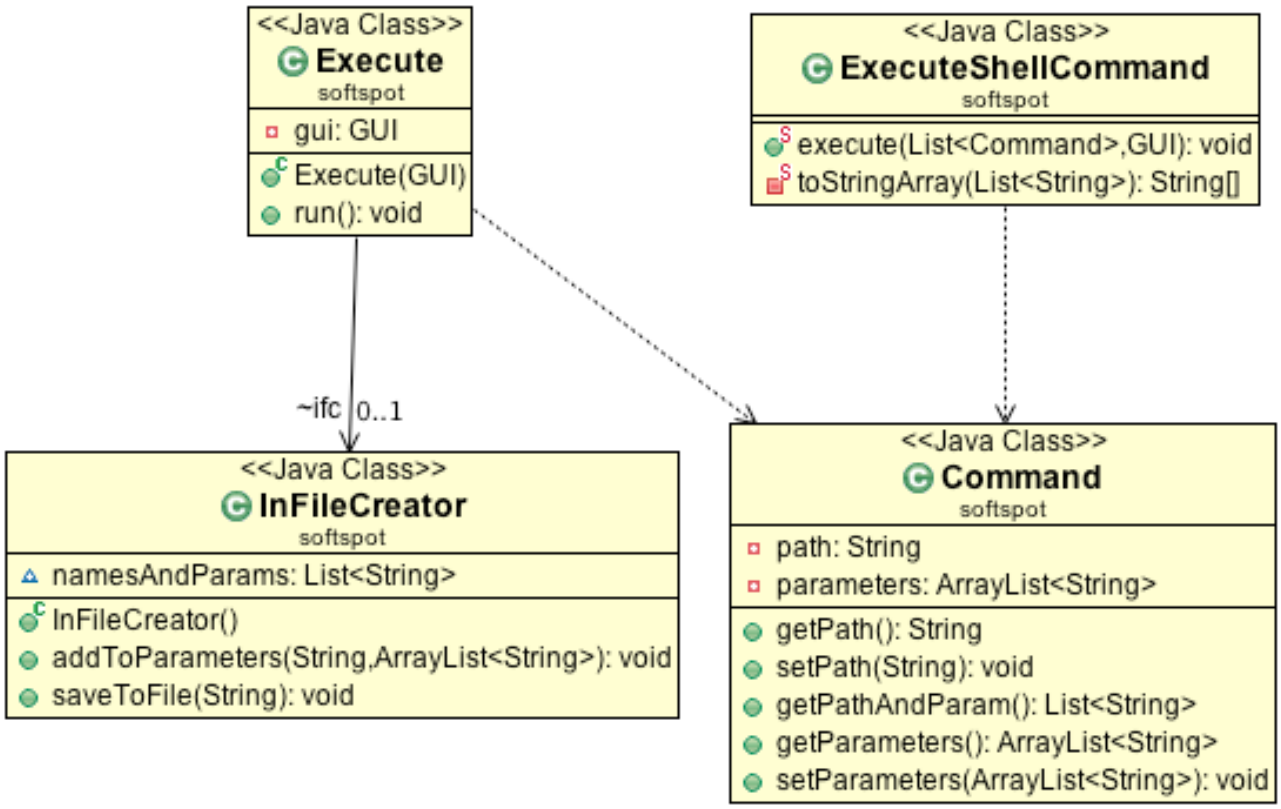


Figure 3.1.1- Class Diagram for System module

3.1.3 Representation Module

The representation module is designed to symbolize certain parts of the DSW and to help provide important functionality required in the DSWGUI brief. The module was also designed as a support to the XML module, by aiding in allowing the user to save both programs and sequences. The module was further designed with the XML module in mind as it helped in deciding a suitable format for the XML files. The module consists of 3 classes. The Parameter class defines a Parameter object with a name, value, data type and a default value. The Program class contains a name and a list of Parameter objects. The Program class also allows for the setting of a programs inputs and it further provides functionality for setting the location of a program on disk. The sequence class contains a name and a list of program objects, stored in a given order.

The class diagram diagram is shown on the next page.

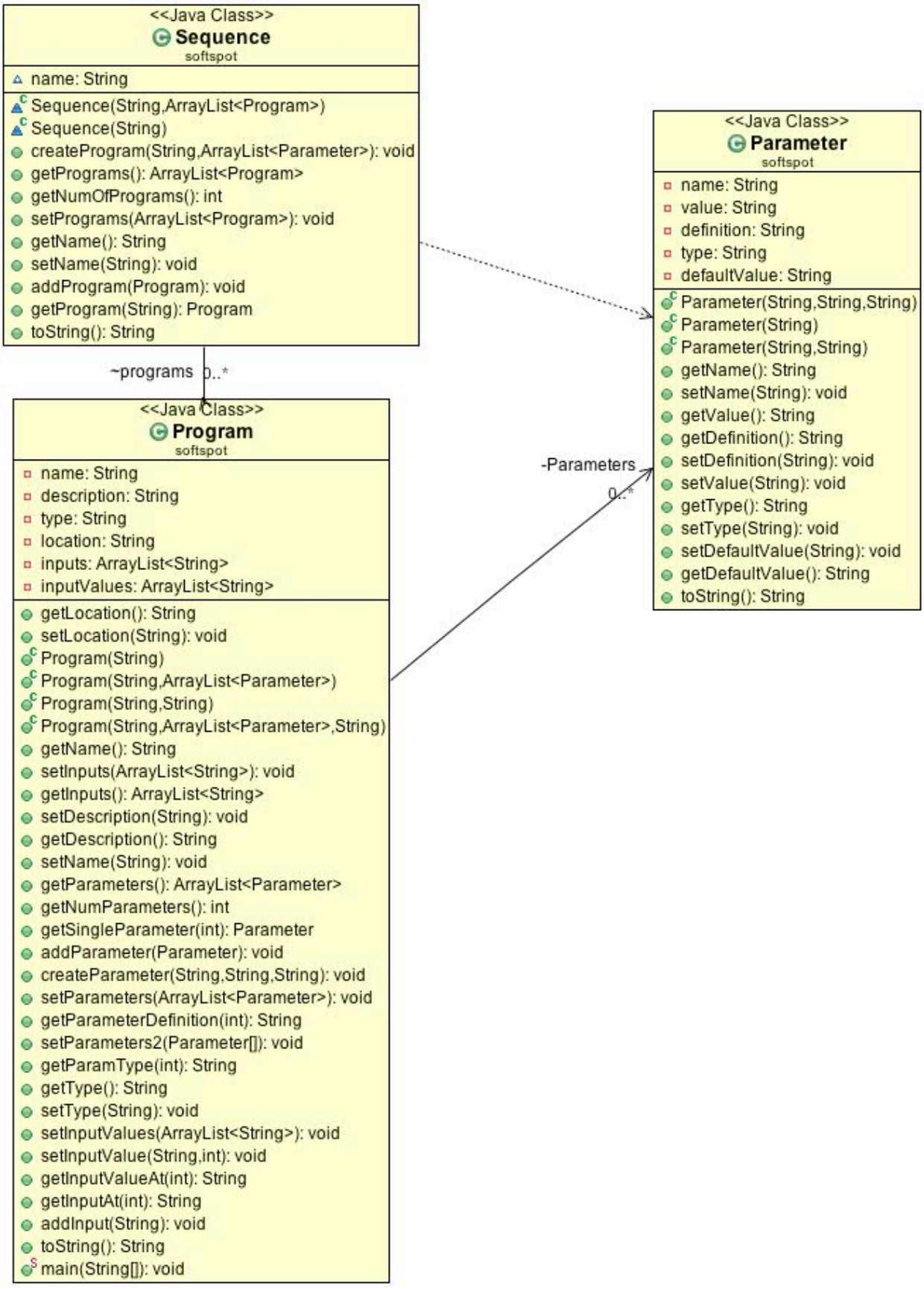

<<Java Class>>
Sequence
softspot
name: String
Sequence(String,ArrayList<Program>)
Sequence(String)
createProgram(String,ArrayList<Parameter>): void
getPrograms(): ArrayList<Program>
getNumOfPrograms(): int
setPrograms(ArrayList<Program>): void
getName(): String
setName(String): void
addProgram(Program): void
getProgram(String): Program
toString(): String
~programs 0..*
<<Java Class>>
Program
softspot
name: String
description: String
type: String
location: String
inputs: ArrayList<String>
inputValues: ArrayList<String>
getLocation(): String
setLocation(String): void
Program(String)
Program(String,ArrayList<Parameter>)
Program(String,String)
Program(String,ArrayList<Parameter>,String)
getName(): String
setInputs(ArrayList<String>): void
getInputs(): ArrayList<String>
setDescription(String): void
getDescription(): String
setName(String): void
getParameters(): ArrayList<Parameter>
getNumParameters(): int
getSingleParameter(int): Parameter
addParameter(Parameter): void
createParameter(String,String,String): void
setParameters(ArrayList<Parameter>): void
getParameterDefinition(int): String
setParameters2(Parameter[]): void
getParamType(int): String
getType(): String
setType(String): void
setInputValues(ArrayList<String>): void
setInputValue(String,int): void
getInputValueAt(int): String
getInputAt(int): String
addInput(String): void
toString(): String
main(String[]): void
-Parameters
0..*
<<Java Class>>
Parameter
softspot
name: String
value: String
definition: String
type: String
defaultValue: String
Parameter(String,String,String)
Parameter(String)
Parameter(String,String)
getName(): String
setName(String): void
getValue(): String
getDefinition(): String
setDefinition(String): void
setValue(String): void
getType(): String
setType(String): void
setDefaultValue(String): void
getDefaultValue(): String
toString(): String

Figure 3.1.1- Class Diagram for Representation module

3.1.4 GUI Module

The GUI module represents some of the core functionality required by the DSWGUI brief. The purpose of the module is to provide a robust and fully operation user interface as outlined in the requirements of the brief. The module consists of 3 classes. The GUI class is the main class in this module, providing the an interface for the command line used by the DSW and allowing for user interaction. It is further responsible for using parts of the XML module to read in a list of saved programs and sequences, which will displayed by the interface. Furthermore, the interface provides help functionality by displaying useful information on programs and parameters. The AddProgramGUI provides a suitable interface for the user to create their own programs.

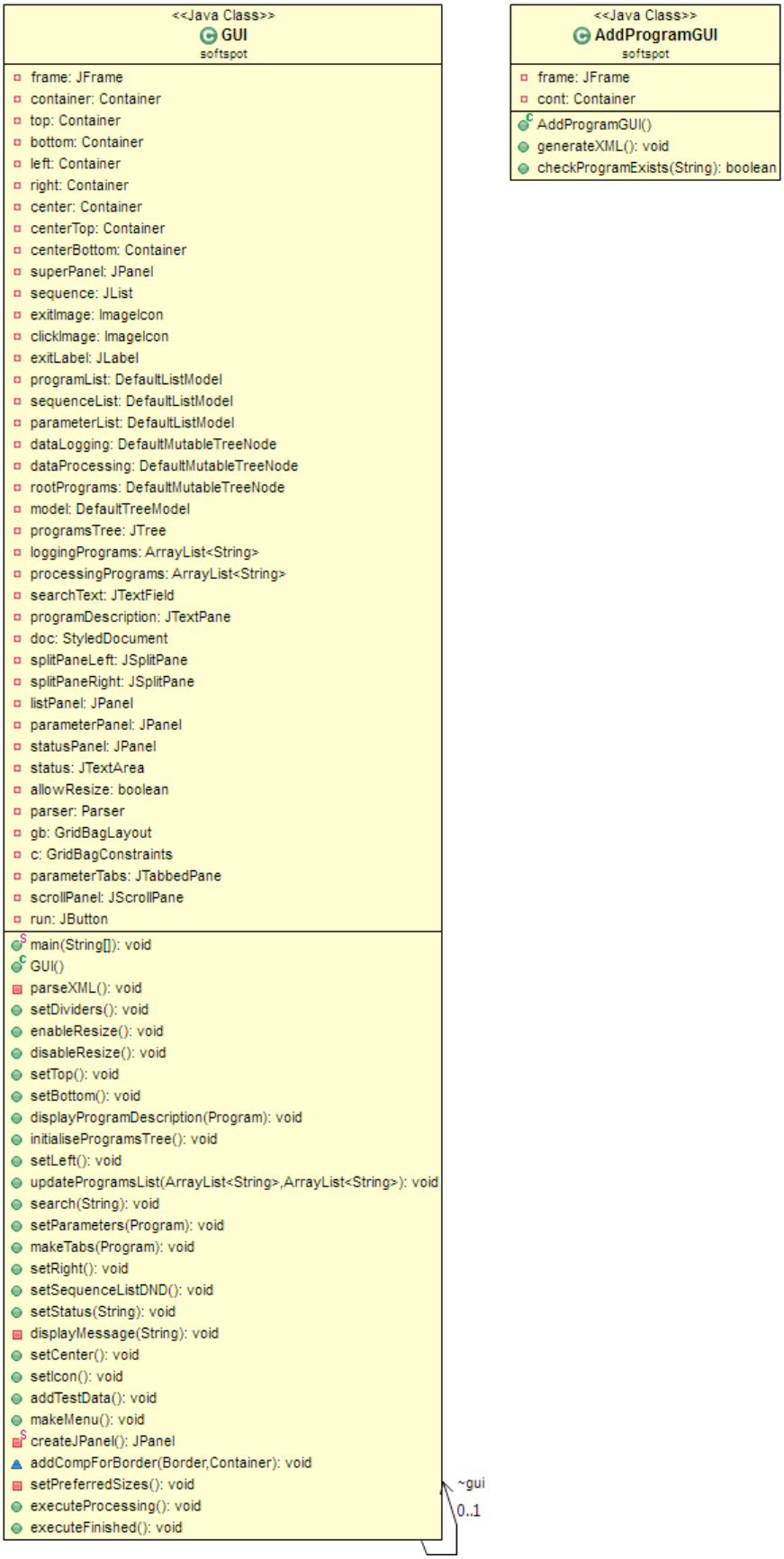

<<Java Class>>
GUI
softspot
frame: JFrame
container: Container
top: Container
bottom: Container
left: Container
right: Container
center: Container
centerTop: Container
centerBottom: Container
superPanel: JPanel
sequence: JList
exitImage: ImageIcon
clickImage: ImageIcon
exitLabel: JLabel
programList: DefaultListModel
sequenceList: DefaultListModel
parameterList: DefaultListModel
dataLogging: DefaultMutableTreeNode
dataProcessing: DefaultMutableTreeNode
rootPrograms: DefaultMutableTreeNode
model: DefaultTreeModel
programsTree: JTree
loggingPrograms: ArrayList<String>
processingPrograms: ArrayList<String>
searchText: JTextField
programDescription: JTextPane
doc: StyledDocument
splitPaneLeft: JSplitPane
splitPaneRight: JSplitPane
listPanel: JPanel
parameterPanel: JPanel
statusPanel: JPanel
status: JTextArea
allowResize: boolean
parser: Parser
gb: GridBagLayout
c: GridBagConstraints
parameterTabs: JTabbedPane
scrollPanel: JScrollPane
run: JButton
main(String[]): void
GUI()
parseXML(): void
setDividers(): void
enableResize(): void
disableResize(): void
setTop(): void
setBottom(): void
displayProgramDescription(Program): void
initialiseProgramsTree(): void
setLeft(): void
updateProgramsList(ArrayList<String>,ArrayList<String>): void
search(String): void
setParameters(Program): void
makeTabs(Program): void
setRight(): void
setSequenceListDND(): void
setStatus(String): void
displayMessage(String): void
setCenter(): void
setIcon(): void
addTestData(): void
makeMenu(): void
createJPanel(): JPanel
addCompForBorder(Border,Container): void
setPreferredSizes(): void
executeProcessing(): void
executeFinished(): void
~gui
0..1
<<Java Class>>
AddProgramGUI
softspot
frame: JFrame
cont: Container
AddProgramGUI()
generateXML(): void
checkProgramExists(String): boolean

Figure 3.1.1- Class Diagram for GUI module

3.2 Interprocess Dependencies

The purpose of this subsection is to show the dynamic behaviour of the complete system; including descriptions of what exactly will take place inside and between each of the modules stated previously. Shown initially is a set of sequence diagrams which demonstrate the processes running inside modules, and how these processes interact with others in external modules. Following these is a sequence of activity diagrams further detailing the processes themselves, including which classes and which methods are used within them.

3.2.1 GUI Module

The GUI Module is responsible for outputting all data for use by the user as well as initiating processes in other modules based on user input. It is a very static module as no main functions are run inside it- it is only responsible for creating objects and initiating processes in external modules.

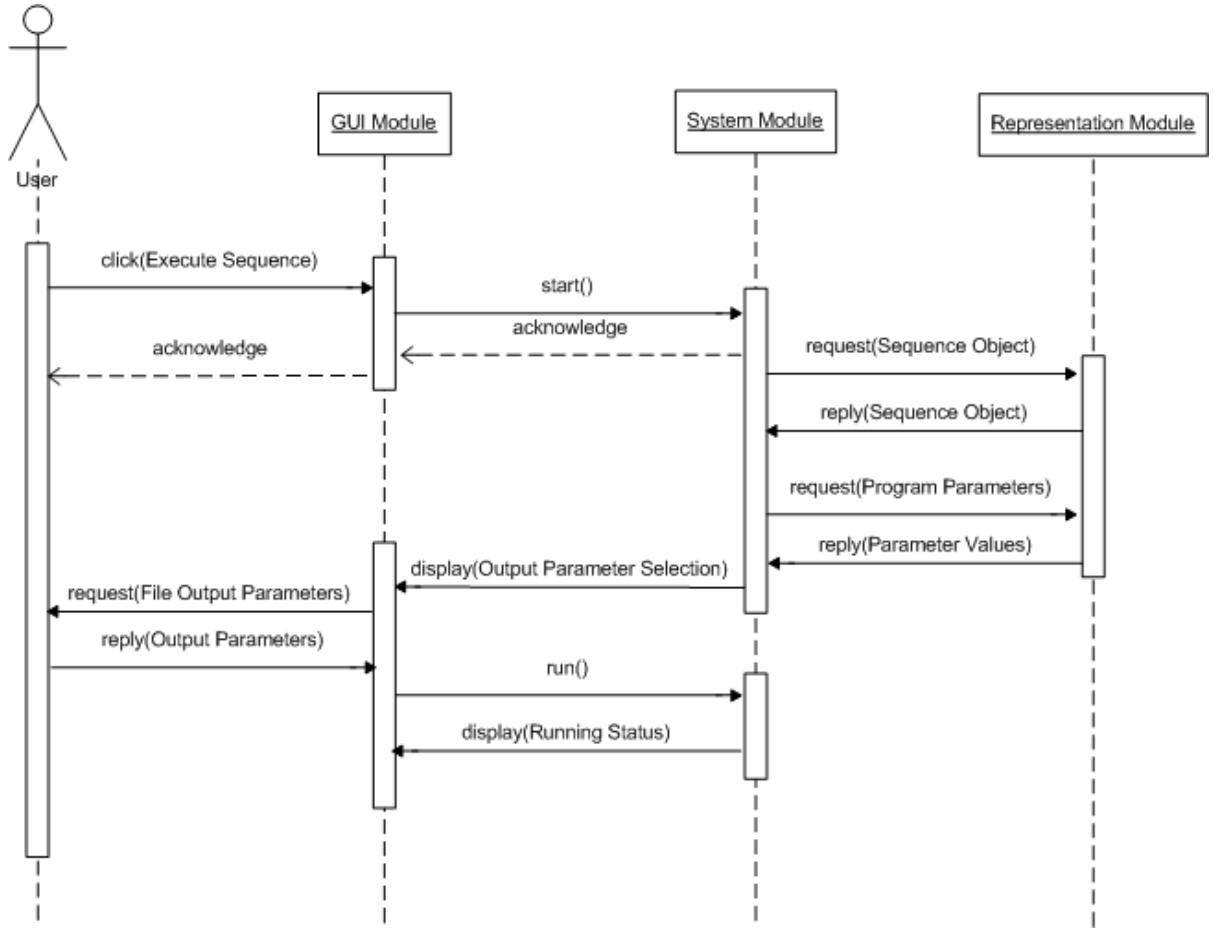


Figure 3.2.1- A UML Sequence Model showing the process interactions when a sequence is run from the GUI

For figure 3.2.1 it is best to demonstrate the process by using a sequence diagram and showing the information relay between the processes in each module. In later described scenarios, the processes are more mono-modular so they will be best demonstrated using activity diagrams on a single module.
Note that in the process in Figure 3.2.1 there is no need to access the XML store to retrieve pre-saved sequence data. When the "Run Sequence" button is pressed, the sequence data will be stored in a Sequence object inside the Representation Module.

3.2.2 Class Description: GUI class

The GUI Class will be the main class for controlling the user interface. It will be made up of a large array of nested JPanels each containing one or more components- i.e. The Programs JTree or the Parameter editing inputs. It is responsible for initiating processes based on the user's input and adjusting the interface accordingly. Because we aim to keep as much of the system functionality down to a single window, this is the class which will be relied upon

most in the whole system, meaning that it must be as stable and reliable as possible. The processes inside this class include:
- The search function, to allow the user to search for a program if they know its name.
- Managing the drag and drop functionality, enabling the user to create and customise their sequence of programs to be run.
- Adjusting the display based on parameter types, quantities and requirements.

Although the class is mainly responsible for initiating processes in other modules, there are some occasions when this isn't the case - such as the System Module (ExecuteShellCommand object) changing GUI components to display the up-to-date program running states.

3.2.3 Class Description: AddProgramGUI

The only other non-nested GUI class: this will be used when the user wishes to add details of a new program to the system. It will take the program name, type, description and parameters as inputs using simple text input fields. Once the user has input the necessary details, it initiates the XMLGenerator class in the XML module. This is required in order to create an XML file based on the program information the user has input.

3.2.4 System Module

The System Module is responsible for interfacing with the command line and running the programs.
The below activity diagram represents exactly what happens inside the System Module when the user runs a sequence.

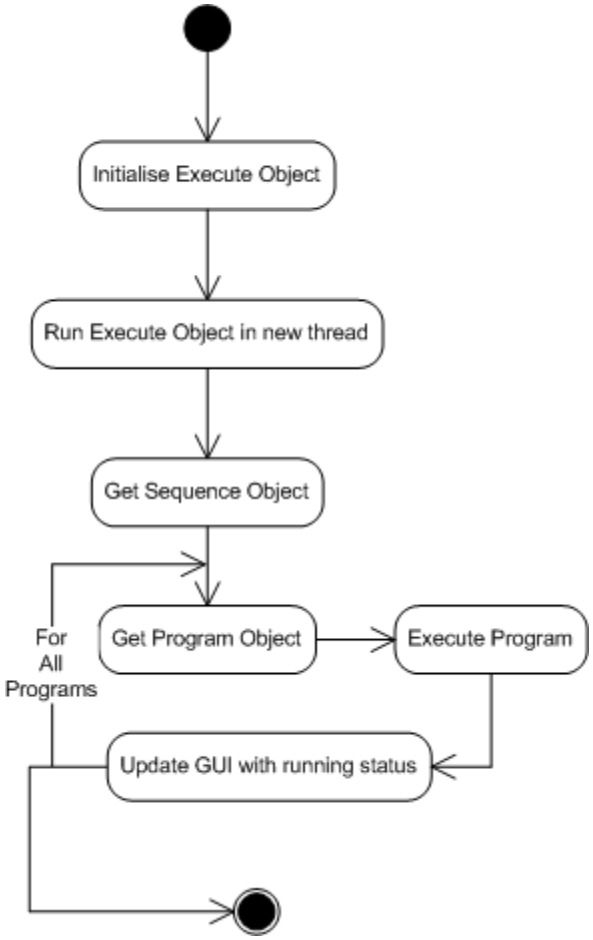


Figure 3.2.2- An activity diagram showing the processes involved in running a sequence

The "Execute Program" process in Figure 3.2.2 is expanded in the activity diagram below. It represents exactly what happens inside the System Module when a program is executed from within a sequence. The InFileCreator class is vital to create the control (.in) file for the program, based on the user's input parameter values. Once this has been created a command list is generated, which is a list of variables required by the program's main method upon execution. This list is then appended to the name of the program and passed to the command line interface to run it, as demonstrated below.

`CalcVel`	`CalcVel.in`	`InFile.tec`	`OutFile.tec`
Program name	Control file	Execute variable 1	Execute variable 2

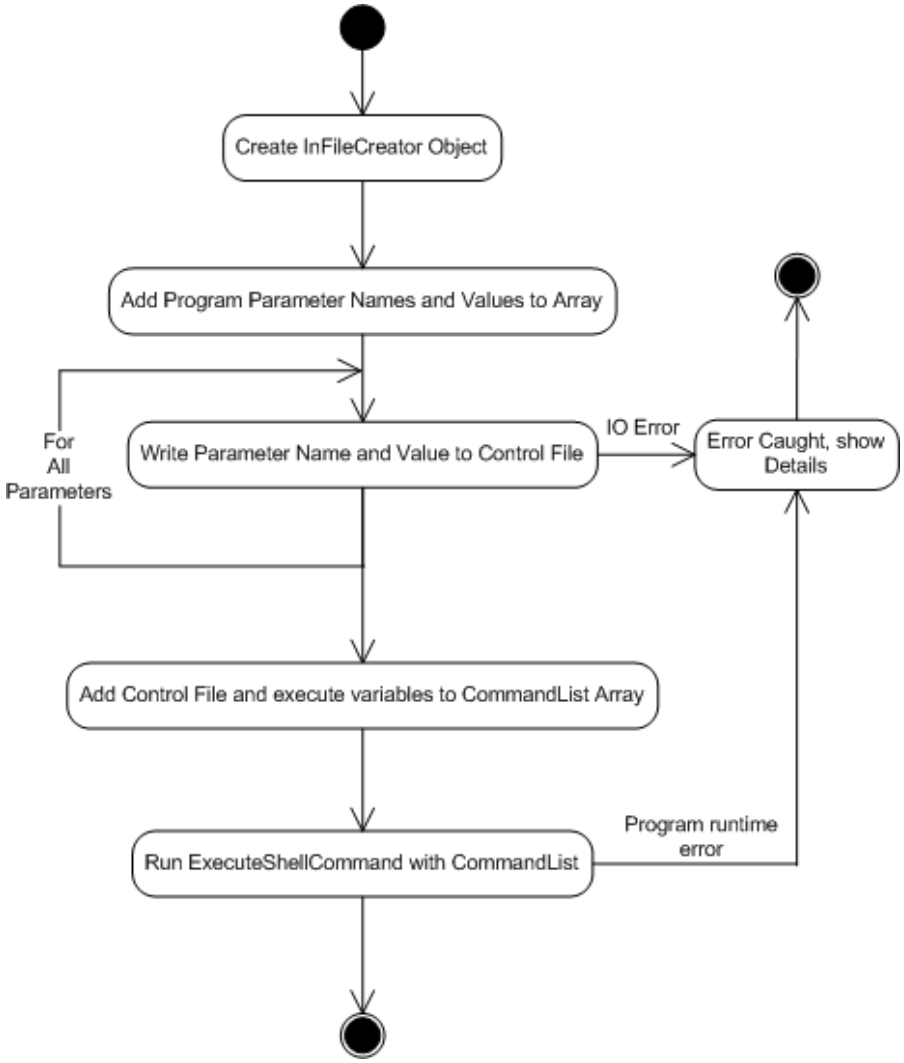


Figure 3.2.3- An activity diagram showing the process of running a program

3.2.5 Class Description: Execute

Execute is a class which will govern the main running of programs in the sequence. It will be run in a separate thread to that of the main GUI in order to have no affect on the GUI operation. This is useful as the user may wish to carry out other functions (such as save the current sequence) while programs are being executed. It is the Execute instance that will create the CommandList array referred to in Figure 3.2.3, based on inputs requested from the GUI.

3.2.6 Class Description: ExecuteShellCommand

The purpose of this class is to actually run the programs. It will be instantiated by the Execute class for each program in the Sequence object, at which point it will be passed the CommandList array and create a command line process to run the program. The class will continuously be sending information to the GUI in order to update the running status displayed to the user.

3.2.7 Class Description: InFileCreator

The InFileCreator class will be responsible for taking inputs from the GUI class and creating the .in file associated with the program. This file is required to run almost any DSW program as it contains the names and values of the parameters- the latter of which are custom to each run. The class will be a simple one whereby only two methods will be required; the first to add parameter name-value pairs to an array and the second to format these pairs and save them to a file.

3.2.8 Class Description: Command

The Command class is basically a representation of what would be entered into the command line in order to run a program. It will contain an array of parameters required to make the specified program run, including the program filepath, the control file and any other Input/Output parameters that are required.

3.2.9 XML Module

The XML Module is required for allowing the long term storage of files vital to the running of the system. It allows the reading of program and sequence information as well as having classes allowing the creation of XML files.

3.2.10 Class Description: Parser

The Parser class is the most important class in the module, as it is responsible for reading in the program information and thus indirectly allowing the user to run programs. The class will implement the ParserInterface class which is used to specify the two main methods required to parse any XML file. The class will be initiated by the GUI module upon startup, at which points its purpose is to parse the Programs.xml file and create Program objects as required. These Program objects will be stored in an array and passed to the GUI, before being displayed in list form to the user.

3.2.11 Class Description: SequenceParser

Similar to the Parser class, SequenceParser will also implement ParserInterface. It will be used to read in a sequence.xml file and will create a Sequence object based on the file nodes and attributes. The class will be instantiated by the GUI when the user clicks “Import Sequence”, and will introduce a pre-constructed sequence which includes all program names as well as each previously set parameter values.

3.2.12 Class Description: XMLGenerator & SequenceXMLGenerator

The purpose of the XMLGenerator class is to allow the user to save program information to secondary storage, meaning that the user can add new programs in addition to the ones the System provides.
The SequenceXMLGenerator class will be very similar, but instead will parse sequence.xml files in order to allow the reuse of sequences used in the past.
The main process within these classes will be the iterative createElement() method, which governs the writing of XML nodes to the specified file. Both classes will implement GeneratorInterface, which will specify the 4 vital methods needed when writing xml files.

Figure 3.2.4 illustrates the process involved in reading in an XML file detailing programs available to the user and creating object representations to allow user manipulations.

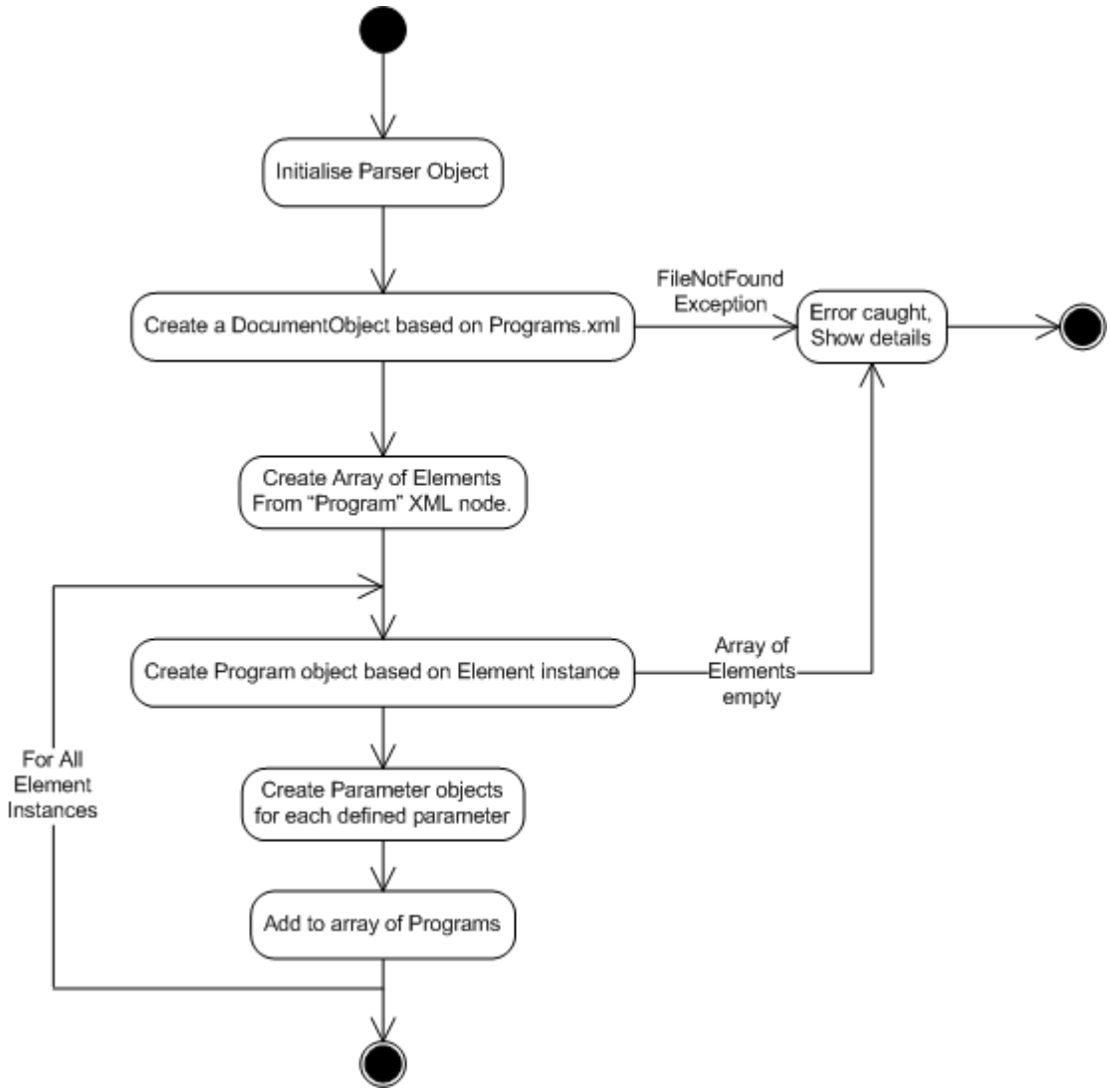


Figure 3.2.4- An activity diagram showing the processes inside the XML Module

3.2.13 Representation Module

Figure 3.2.5 is a sequence diagram demonstrating the process by which a user edits the parameters and how the processes inside the modules influence those in other modules.

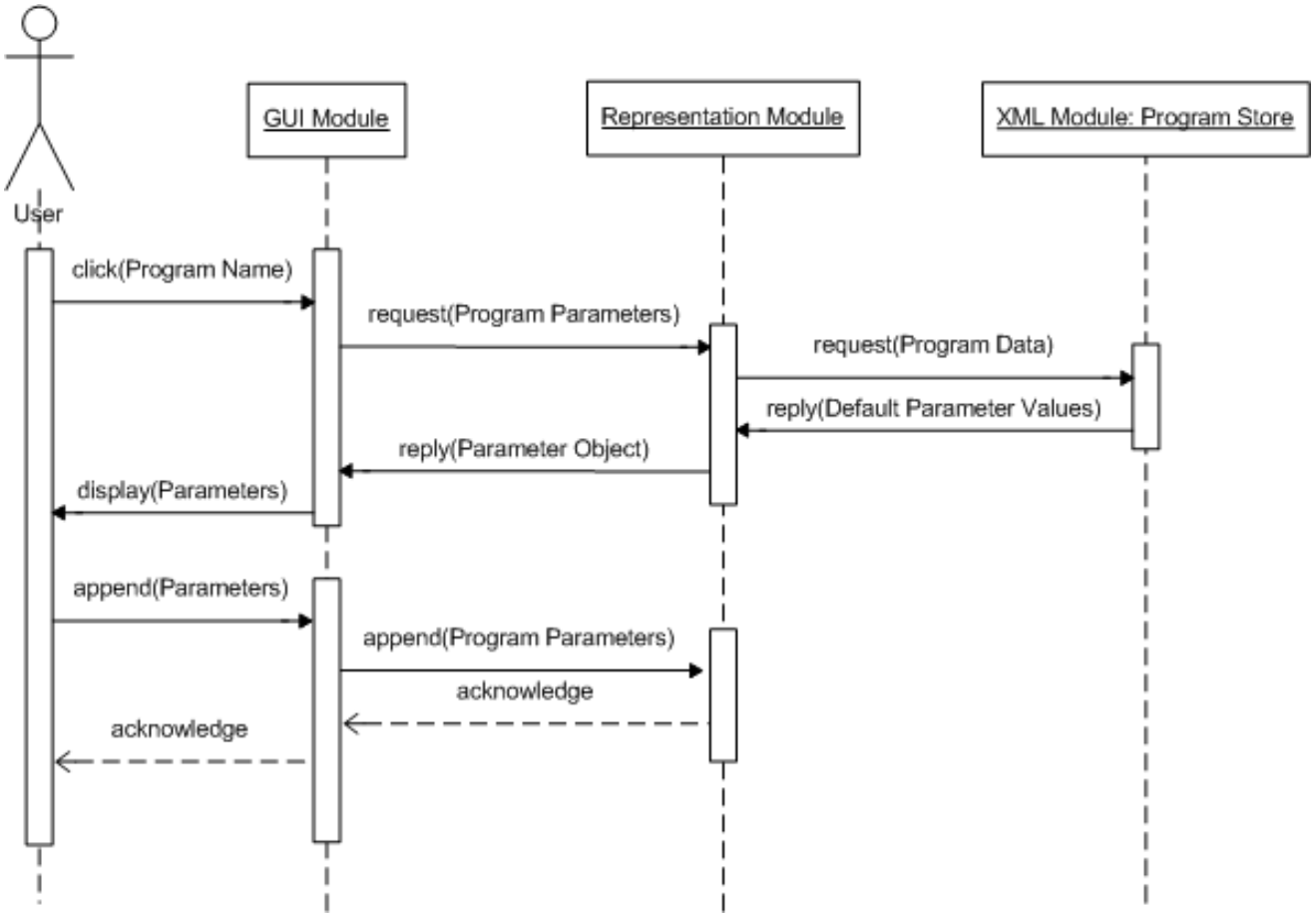


Figure 3.2.5- A UML Sequence Model showing the interactions between the processes inside each module when the user changes program parameters.

The XML module is queried to retrieve the default program parameters. These are returned and saved into the Program object inside the Representation Module. The existing parameter information is then retrieved from the object and displayed to the user. When these parameter values are changed, the GUI module sends the data to the Program object and the parameter values are saved. Note that no custom data is edited by the XML Module, as this module is only necessary for gathering default program information and has no contact with the user's personalised value.

In the case of Figure 3.2.5, the Representation Module is only using processes inside the Program and Parameter instances- the Sequence class is not used. The methods inside the Sequence class are used when the user is editing or importing from a file in order to re-use sequence parameters from a previous run. The process of saving a sequence to an xml file is shown below, with the focus being on the processes taking place inside the Representation Module (the Sequence Class) rather than the XML Module.

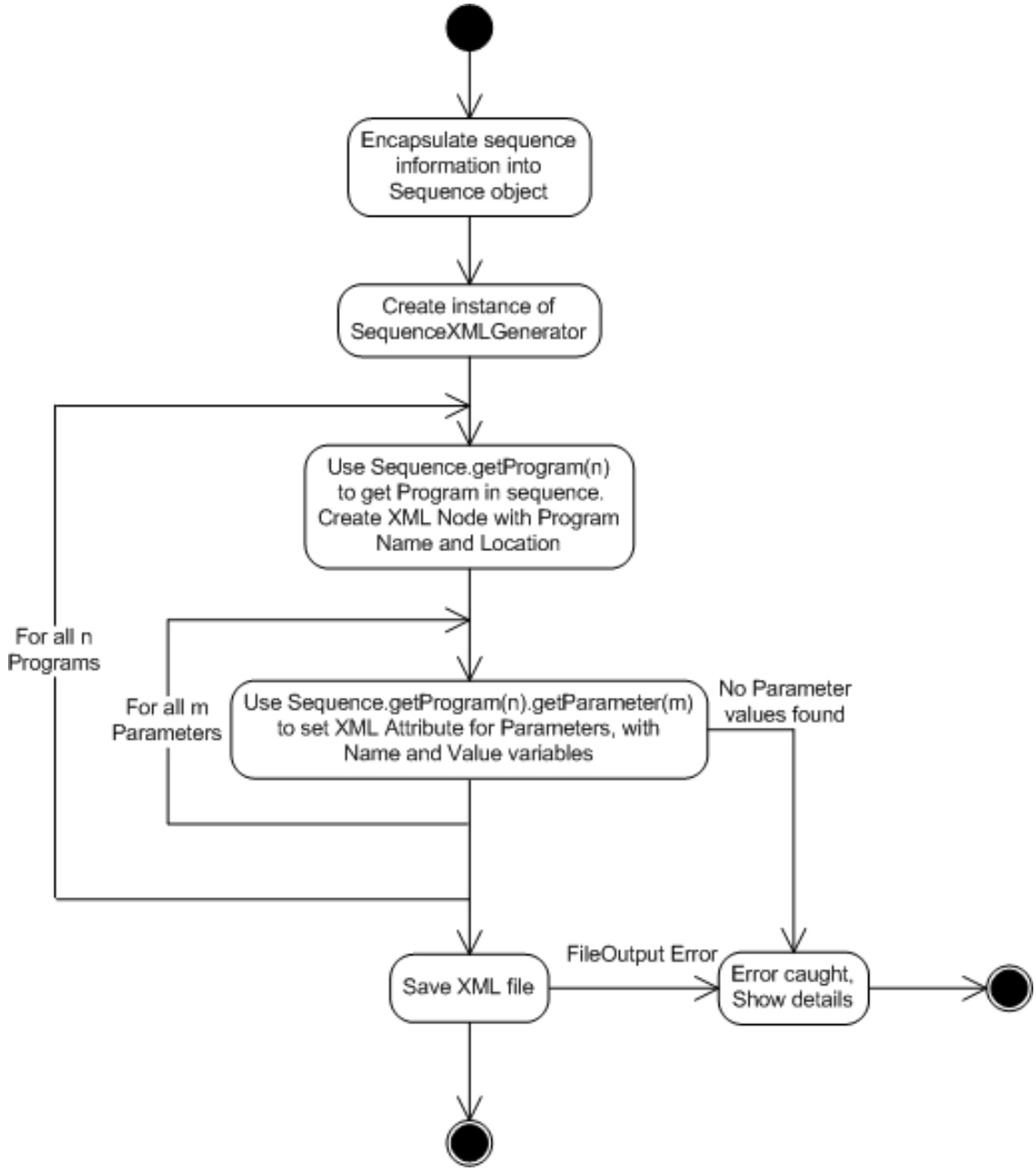


Figure 3.2.6- An activity model showing the processes inside the Representation Module when saving a sequence to a file.

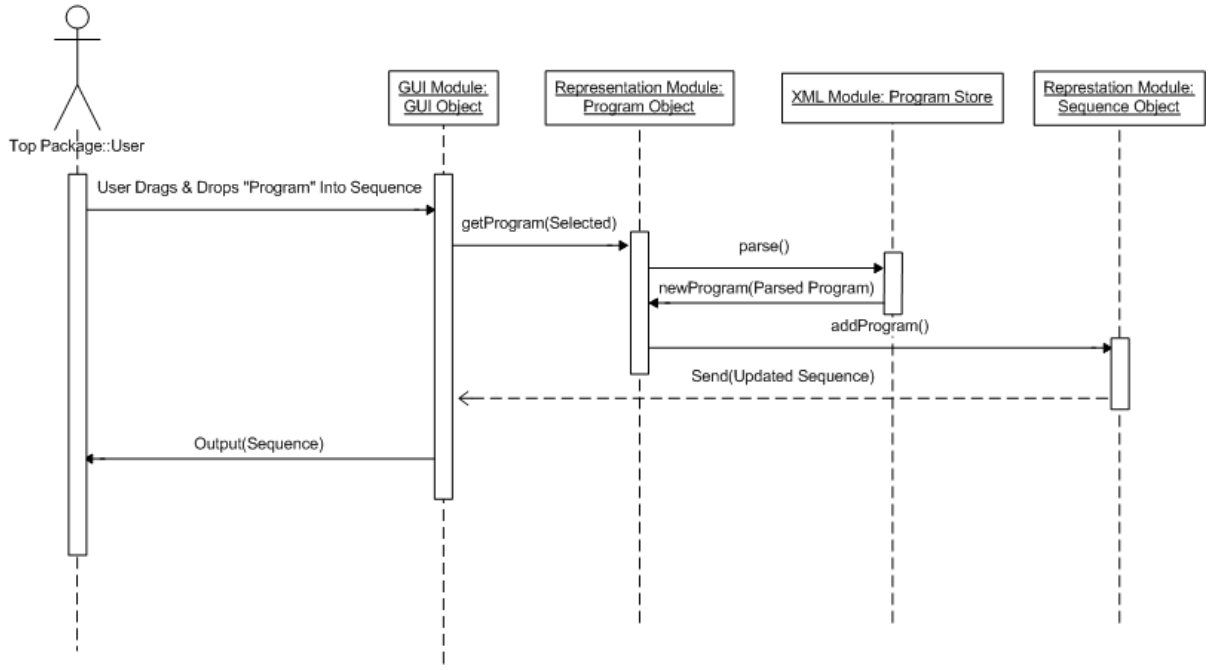


Figure 3.2.7- A Sequence Diagram representing the method calls inside the classes between different modules

Figure 3.2.7 shows the process that will take place between classes in the system when a user drops a program into the sequence. In the diagram the parse() method is called to get the detailed information regarding the program that has been added to the sequence. The process is a complex one in which many classes are interacting.

3.2.14 Class Description: Program

The Program class is one created by the XML module when a new program is read in from file. It will be used to encapsulate all data that would ever be required of the program, such as the type and description as well as more important attributes such as the program location and an array of Parameters.

3.2.15 Class Description: Parameter

The Parameter class will represent the parameters inside the control file of each program. Each Program instance will have an array of Parameter objects which are used by the InFileCreator class (System Module) to create the control file. Each Parameter will have a default value as well as a value which is set by the user upon alteration.

3.2.16 Class Description: Sequence

The Sequence class will simply store an array of Program objects, which will then be accessed by both the Execute class when running the sequence and by the SequenceXMLGenerator when saving a user's custom sequence to a file. The class will contain simple methods to add and retrieve programs from the sequence, meaning that it will be easy for other processes to access the programs in the sequence.