Durham University
School of Engineering and Computing Sciences

Design Document

2010-11

Project Name:

Durham Software For Wind Tunnels

Team Name: Creative Solutions

Team Number: 7

Team Members:
Carl Dobson
Joseph Flynn
Christopher Miley
David Roberts
Adam Wardle

Document Information

Project Name:	Wind Tunnel Software		
Prepared By:	Creative Solutions	**Preparation Date:**	22/02/2011
Email:	cs-seg7@durham.ac.uk		
Document Version No:	2.7	**Document Version Date:**	22/02/2011

Version History

Version	Ver. Date:	Revised By:	Description
1.0	14/12/2010	Carl Dobson	Created document and began beginning of Introduction Draft Section 1.1
1.1	10/02/2011	Joseph Flynn	Merged all sections together.
1.1a	13/02/2011	Carl Dobson	Added initial draft of change of requirements table.
1.2	16/02/2011	Adam Wardle	Updated the Architectural Form and High Level Overview sections.
1.3	16/02/2011	Christopher Miley	Added initial draft of Process Descriptions.
1.4	16/02/2011 & 17/02/11	David Roberts	Appended current version of static decomposition and inter-module deps.(sections 3/3.1). Added links to requirements
1.5	18/02/2011	David Roberts	Made high lev.todecomp. Progression more explicit. Rearrunged inter-module/inter-process and expanded diagrams
1.6	19/02/2011	Carl Dobson	Updated initial draft of change of requirements, added ATL to section 1.4.
1.7	19/02/2011	David Roberts	Diagram work on inter process dependencies& figure labelling in struct. Decomp. May need to nuke the page limit to do justice to everything...
1.8	20/02/2011	Adam Wardle	Added advantages and disadvantages to the Architectural Form section, and updated the References section.
1.8	20/02/2011	David Roberts	Working on sequence diagrams for frontend/view
1.9	20/02/2011	Christopher Miley	Added second draft of section 2.2.
2.0	20/02/2011	Christopher Miley	Finalised descriptions of section 2.2 and added new scenarios to section.
2.1	21/02/2011	Carl Dobson	Added section 1.6 overview, added mockflow and gui to section1.4
2.2	21/02/2011	David Roberts	Condensed some pages together to save space for ending diagrams.
2.3	21/02/2011	Christopher Miley	Proof read and re-formatted text in section 2.2 and added new sequence diagrams for section.
2.4	21/02/2011	David Roberts	Final diagrams added: clean-up and adding missing text from now on
2.5	22/02/2011	Carl Dobson	Changed images on section 2.1 also added detailed description under workflow wizard images.
2.6	22/02/2011	Christopher Miley	Added final version of diagrams and finalised text for section 2.2.
2.7	22/02/2011	Adam, David, Carl and Chris	Proofreading and various corrections

Table Of Contents

1 Introduction

1.1 Purpose

The design document will outline all the different parts of the software and their functionality. We will be looking at the system from a high level[1], defining exactly what the system will do and how it will do it. We will use an object orientated design approach to the project as the project is being built to run in Java[2]. Object orientated design is the process of planning a system of interacting objects for the purpose of solving a software problem.

The purpose of Aero Tunnel Lab(ATL)[3]is to provide an easy to use graphical interface, for a pre existing suite of programs that are being used by the Engineering department within Durham University.

The objectives of the system are to aid the use of the current suite of software and to avoid the current human error in the text based input of the files in a text editor and commands needed to run the suite in the command line[4].

1.2 Scope

When completed the system will be able to:

- Run all the programs within the Durham Wind Tunnel software suite
- Add/edit/delete programs
- Add/edit/delete custom workflows
- Add/edit/delete programs within custom workflows
- Keep a log of all events/warnings/errors
- Provide an in depth user guide integrating existing and new manuals
- Offer user tool tips

1.3. Changes to requirements

Since we published our requirements document in Term 1, there have been a number of changes that we have had to make to the requirements that our system is to conform to, as an end product. The changes are vital in regards to functionality and will improve the usability of the system for the end user.

Requirement	Change	Reason	Consequence
6.1.1.11	Change requirement. Rewording. The new wording is as follows: Add workflow to a project. Projects are used to allow batch processing of multiple files. Across the same set of programs. Priority Low, this is an added extra to the functionality of the system not a core component.	Wording was slightly unclear.	Functionality is un altered, just that the wording is clearer for the programming team to be able to implement this requirement.

6.1.1.12	Change requirement. Rewording. The new wording is as follows: Modify execution order of workflows within a project. So that a user can pick which files should be processed first within a project. Priority Low, as per requirement 6.1.1.11	Wording was slightly unclear.	Functionality is un altered, just that the wording is clearer for the programming team to be able to implement this requirement.
6.1.1.13	Change requirement. Rewording. The new wording is as follows: Remove a workflow from a project. So that if a user wishes not to process a file, then they can remove that file from the project. Priority Low, as per requirement 6.1.1.11	Wording was slightly unclear.	Functionality is un altered, just that the wording is clearer for the programming team to be able to implement this requirement.
6.1.1.14	Remove requirement.	As templates can be created, by simply storing a workflow, rather than having two separate requirements (6.1.1.15) which suit the same purpose.	The functionality of the system is still the same: the user is still able to make templates by saving a workflow. Then all that the user needs to do is alter the parameters of the programs in the workflow.
6.1.1.21	Remove requirement.	This is because David Sims-Williams (our client) wishes for executable files to remain stored on the W:\ drive and to be accessed directly from there rather than from the local hard drives. Originally this requirement was suggested by a user of the system.	This changes the functionality of the system, and means that the system does not need to copy and keep track of programs on the local hard drive. Instead, the system only has to look at the global location for the programs on the W:\ drive.

6.1.1.22	Remove requirement.	Due to requirement 6.1.1.21 being removed.	Again as above for the consequence of section 6.1.1.21

1.4. Definitions, Acronyms and Abbreviations

Aero Tunnel Lab – The full name of our software

ATL – abbreviated form of Aero Tunnel Lab

GUI – Graphical User Interface, one of the ways a user can interact with software

UI – user interface, a form of interaction with a system, be it graphical or otherwise

Mockflow – a website [http://www.mockflow.com/] that allows you to create and click through a mock up of a GUI design allowing the user to see how the screens will flow from one to another.

DSW – Durham Software for Wind tunnels, used often to refer to applications which are part of the wind tunnel suite

1.5 References

[Throughout document: Original project brief used for essential requirements]

[1] High Level - A high-level design provides an overview of a solution, platform, system, product, service, or process. (http://en.wikipedia.org/wiki/High-level_design, Top Line, Accessed 18/01/2011 – 14:55)

[2] Java – Software development language (http://www.java.com/en/, Java Homepage)

[3] Aero Tunnel Lab (ATL) - Aero Tunnel Lab is the name of our software, with ATL as short hand for the software.

[4] Command Line – Is a way of communicating with a computer through the use of text based commands. (http://en.wikipedia.org/wiki/Command-line_interface, Accessed 18/01/2011 – 15:05)

[5]Model-View-Controller architecture diagram (http://msdn.microsoft.com/en-us/library/ff649643.aspx, Accessed 1/2/2011)

[6]Pros & Cons of MVC architecture (http://www.ghytred.com/ShowArticle.aspx?MVC#ProsandConsofbasicMVC, Accessed 20/02/2011)

[7]Front End Back End architecture (http://en.wikipedia.org/wiki/Front_end_and_back_end, Accessed 01/02/2011)

[8]Call and Return architecture [Lecture Slides: Software Engineering (Design): Lecture 2]

[9]Monolithic Application architecture (http://en.wikipedia.org/wiki/Monolithic_application, Accessed 03/02/2011]

1.6 Overview

The rest of the document will portray a detailed description of how the core components of our system “Aero Tunnel Lab” will be designed. This helps give a better understanding on how to develop the system off the back of our system requirements document.

This document is technical due to the main audience of the document being a programming team and will need to have a good understanding of technical terminology. There is however an Acronym and definition section if the reader does not know specific terminology. The reason for this being the audience of this document is so the programming team have a good understanding as to what work needs to be carried out, while producing the system in the way it is intended and to not stray away from this specific guideline. All information is provided to help the programming team to do so.

This document is structured with all major components expected by the IEEE standards. There will also be diagrammatic explanations to help clarify and simplify our ideas and understanding of the professional design principles. Some information may be left out or not touched upon greatly in this document if and only if it is very intuitive or greatly beyond the scope of the document.

The remainder of the document will contain an overview of our preliminary design for the system including user interface mock-ups. After that there is a description of common processes that the system uses. Other areas of the document show a complete decomposition of the whole system.

2. Interface Descriptions

2.1 User interfaces

It is extremely important that the user interface reflects the intention to simplify the usage of the wind tunnel applications: it must cater for an audience of mixed ability who may not understand the meaning of the jargon in the original documentation. To this end, the priority for the interface design is simplicity. We have designed how the GUI will look using a program called ‘MockFlow’ [see abbreviations]. This program has allowed us to develop the following screens and we have also generated a sample of how the program will operated.

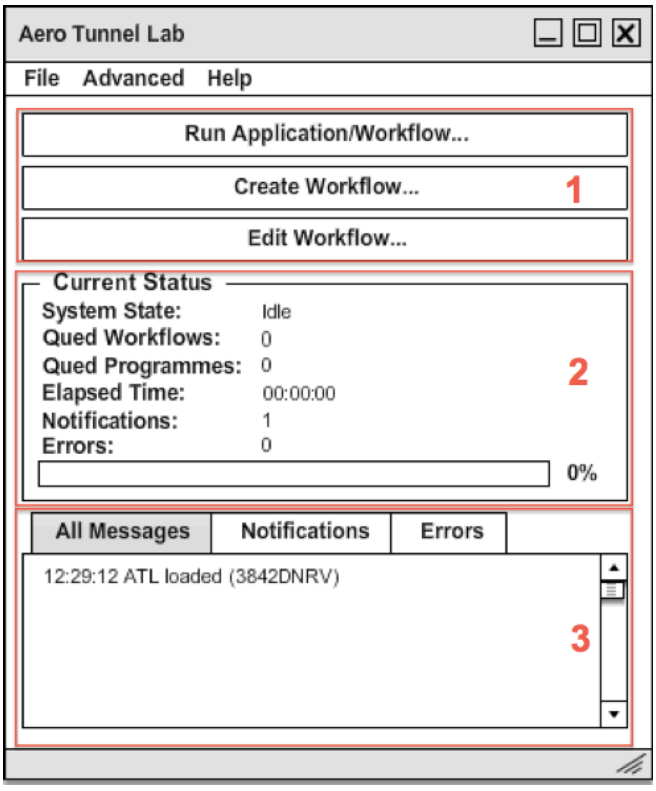
ATL – Main Overview

This is the overview screen that is broken down into three sections.

The first[1] contains buttons at the top of the screen to allow the user to quickly run a single application or workflow, create new workflows or edit existing flows.

The second[2] shows the status of the program. This will show any queued programs or applications and elapsed time.

The third[3] section displays a detailed log since the software was loaded, showing all errors, warnings and notifications.



Wizard Screens

Navigation between the different sections will be standardized, so that no matter what operation you are trying to achieve, the procedure will be similar. This will ensure that an end user will be able to easily perform all functions. We will use a 'wizard' system, which will enable us to split each task into different stages. Each stage will have clear simple instructions on the screen.

There will be wizards for four different functions:

- Run application or workflow
- Create workflow
- Edit workflow
- Add new program

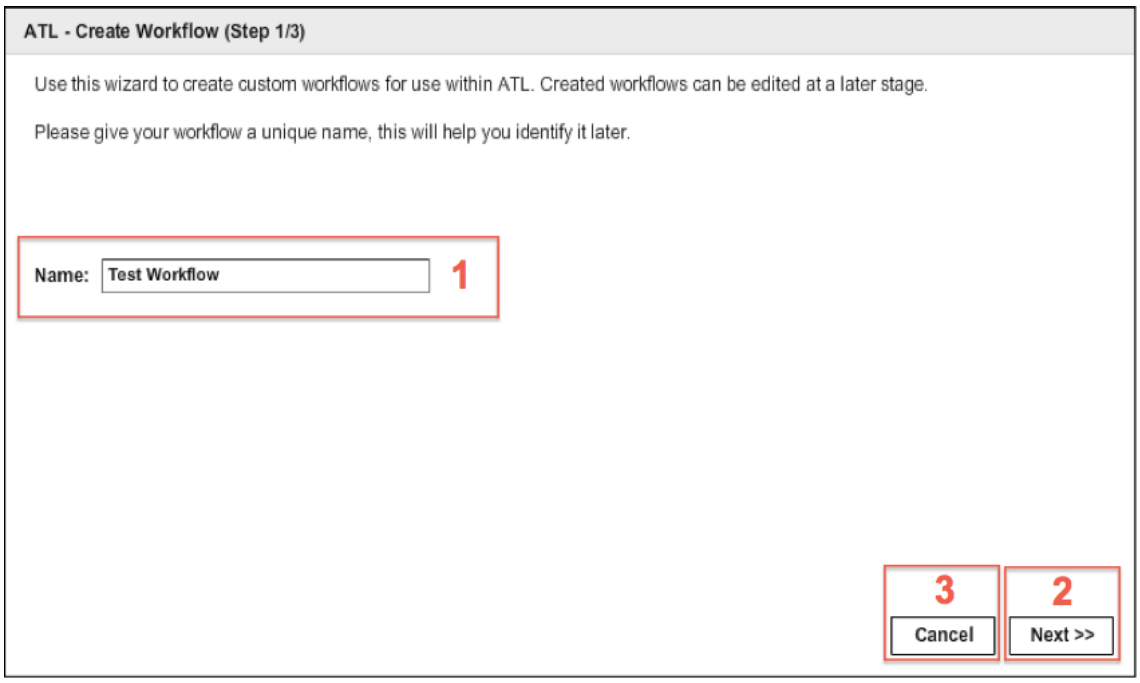


***Fig 2.1.2** – Create new workflow wizard (stage 1/3)*

The above figure shows the initial screen that is displayed when a user tries to create a new workflow. The screen offers the user the chance to name[1] the workflow, using a text box. It also has two buttons for the user to continue on to the next screen[2] and a button to cancel[3] the creation of the workflow. The next button is located on the right and the cancel on the left, as the software is designed to be used in a western civilisation which read left to right and therefore intuitively looks to continue to the right and to cancel/go back to the left.

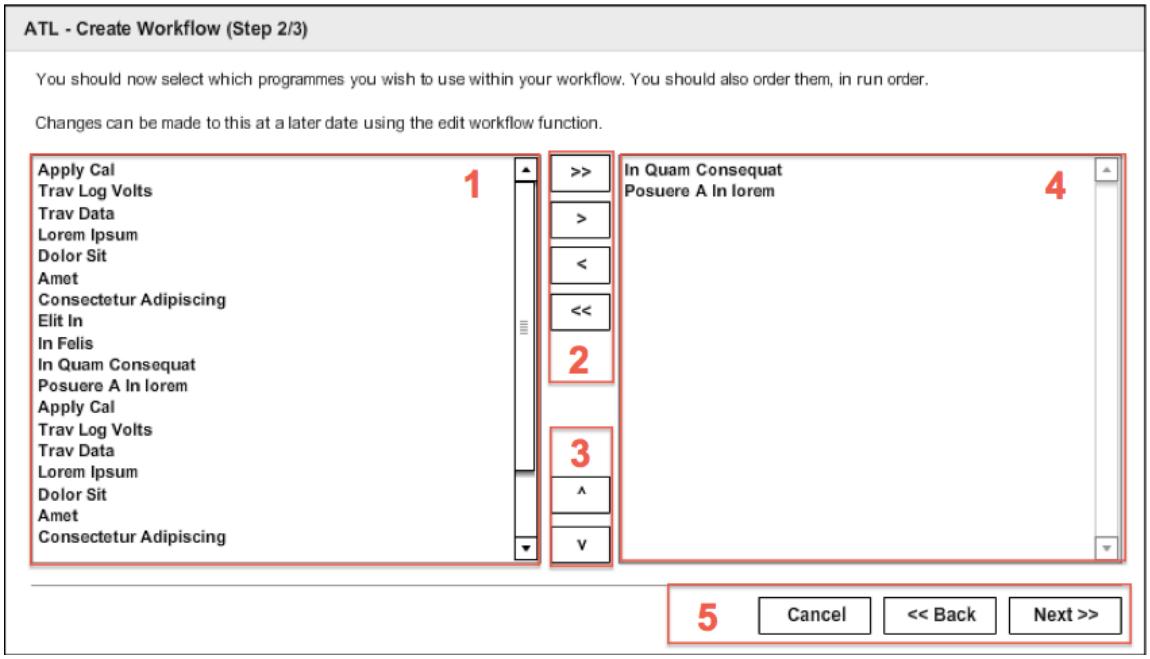


Fig 2.1.3 *– Create new workflow wizard (stage 2/3)*

The above figure shows the second stage of creating a new workflow. The screen shows a list of all of the available programs[1] from the W:\ drive. Upon selecting one of these programs you can then use the buttons[2] to add or remove applications from your workflow, the top button adds all, the second adds selected program, third button removes selected program and the fourth button removes all programs from the workflow. The list of programs[4] shows the programs which are currently in the workflow, upon selecting one of these programs the user can then change the execution order of the program, through buttons:[3] the first one moves the selected program up one spot and the second button moves the selected program down one spot in execution order. Along with buttons[5] which allow the user to cancel, go back and to move forward to the next section of the workflow creation wizard, again with the move forward buttons to the right and cancel and go back to the left.

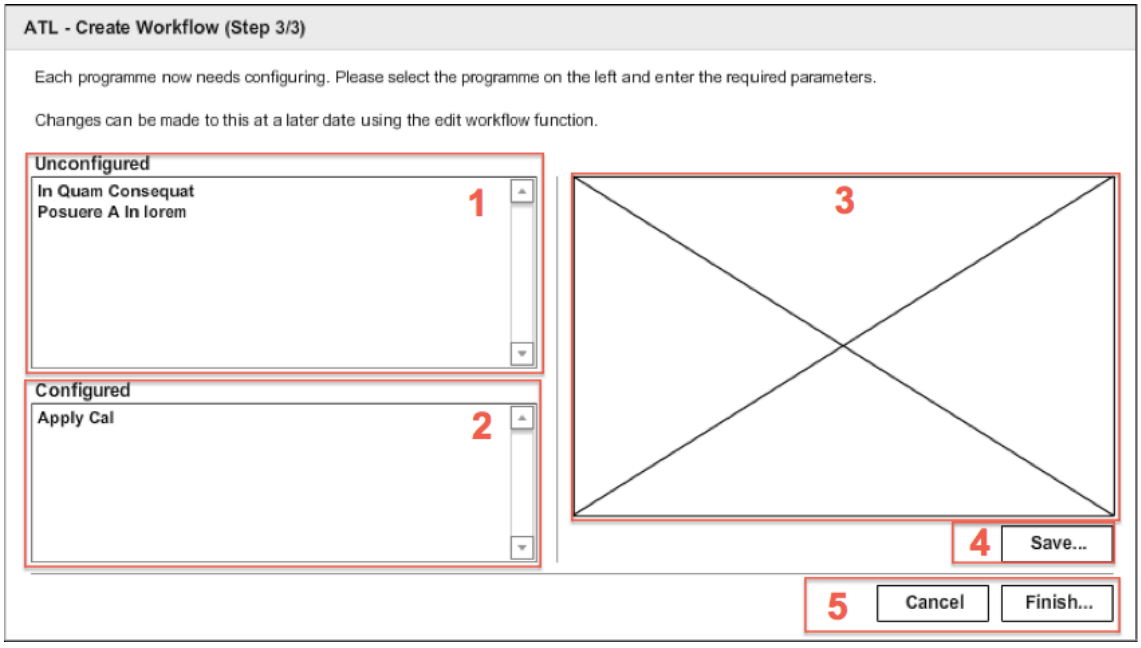


Fig 2.1.4 *– Create new workflow wizard (stage 3/3)*

The above figure shows the third and final screen of the workflow creation wizard. This screen shows the user all of the programs in their workflow again, also showing which programs have had their parameters configured. With the un-configured programs appearing in section[1] and the configured in section[2]. Upon clicking on any of the programs the user is then given all of the parameters and are able to edit them in section[3]. There are also buttons on this page to allow the user to save[4] the workflow. Cancel and Finish[5] the finish will then attempt to run the workflow but only all of the programs in the workflow have been configured.

2.2 Process Descriptions

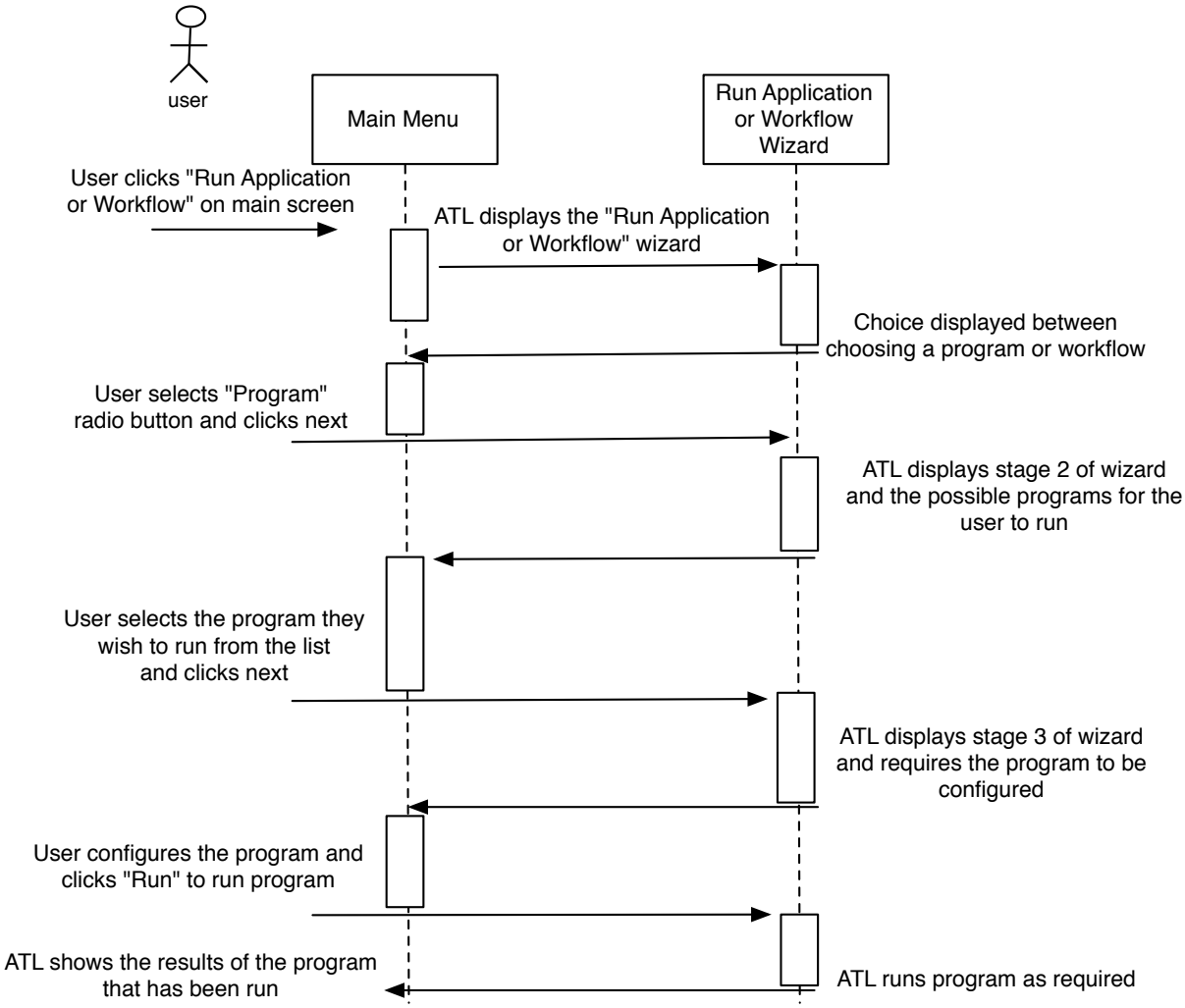
This section of the Design Document details a number of example scenarios, for the use of Aero Tunnel Lab and how the system allows a user to complete their required tasks. The section contains brief descriptions of how a user will achieve a task, with flow/sequence diagrams included, which demonstrate the processes, and these are shown below each description of the scenario.

Running a Program

The main functionality that Aero Tunnel Lab offers is the ability to run a program (application) easily, using the run program wizard.

Once the user has accessed the wizard, the intuitive layout of the wizard and the descriptive information that is contained where required, makes the wizard as easy as possible for the user to use. The wizard contains 3 logical steps, which break down the process of running a program, so that it is not confusing for the user. If the user has any problems with using the wizard, there is a page in the help browser with step-by-step instructions of how to run a program; so the user can refer to that page if they have any issues.

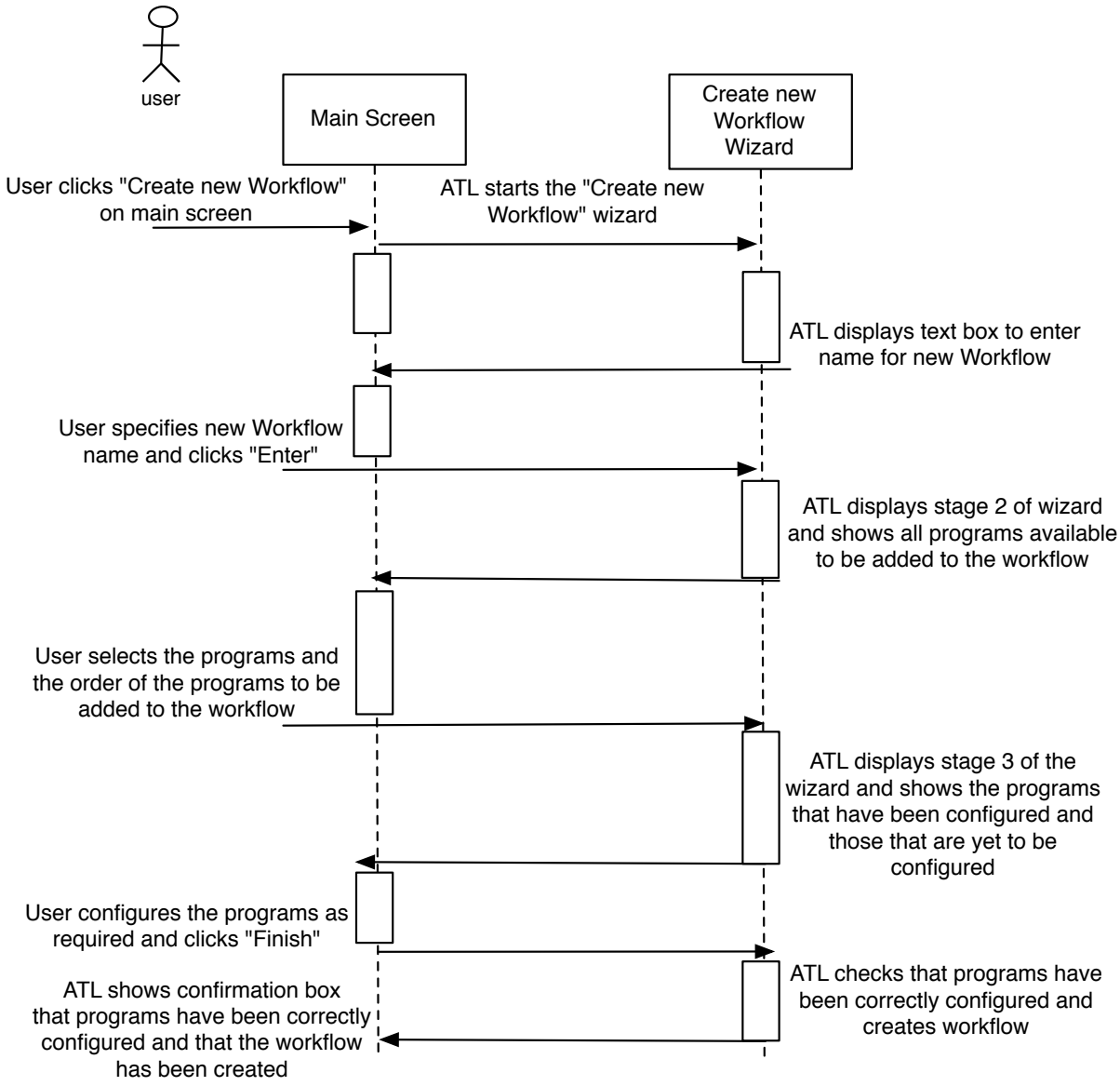
The usage sequence diagram below details a scenario of the user running a program (application) using Aero Tunnel Lab.



Creating a Workflow

Creating a workflow is the second key functionality that Aero Tunnel Lab offers. Creating a workflow allows the user to run a number of programs in succession, which is often a key requirement for the users of the current system. The workflow will be set up, in terms of the programs that are run in the workflow being configured, prior to the workflow being executed. This is achieved by the user running the Create a Workflow wizard, which is spilt into 3 logical steps, which allow the user to firstly name the workflow that they are creating, secondly choosing the programs that they wish to run in the workflow and thirdly configuring the programs that will be run in the workflow. The user is then able to execute the workflow that they have created at any time, by running the "Run Program or Workflow" wizard and selecting the workflow that they have created, from the drop down menu.

The usage sequence diagram below details a scenario of the user creating a workflow using ATL.

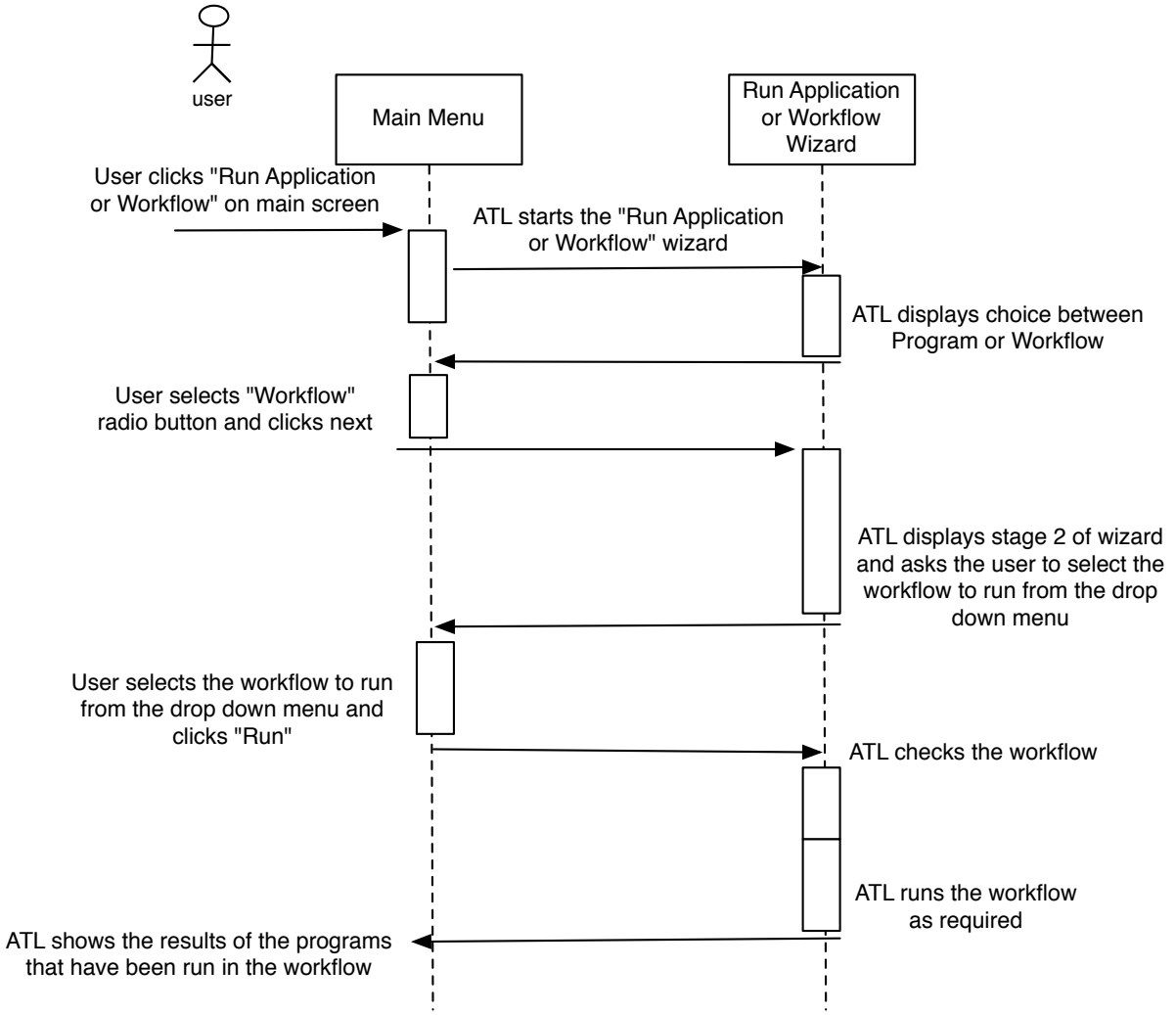


Running a Workflow

Running a workflow is a further key functionality that Aero Tunnel Lab offers. Running a workflow allows the use to run a number of programs in succession, which has been defined previously, which is often a key requirement for the users of the current system.

As with running a program, running a workflow is fairly simple in ATL. This is achieved by using the "Run Program or Workflow wizard". As with running a program, the wizard contains 3 logical steps, which break down the process of running a workflow, so that it is not confusing for the user. The first step of the wizard asks whether the user wishes to run a program or workflow (in this case the user must select workflow), the second step asks the user to choose the workflow that they wish to run from a drop down menu showing all available workflows and the third step requires the user to configure any programs that remain to be configured, in order for the workflow to correctly execute. Once the user has used the wizard, the workflow will execute and the programs in the workflow will be run as required.

The diagram below details the scenario of the user running a workflow using ATL.

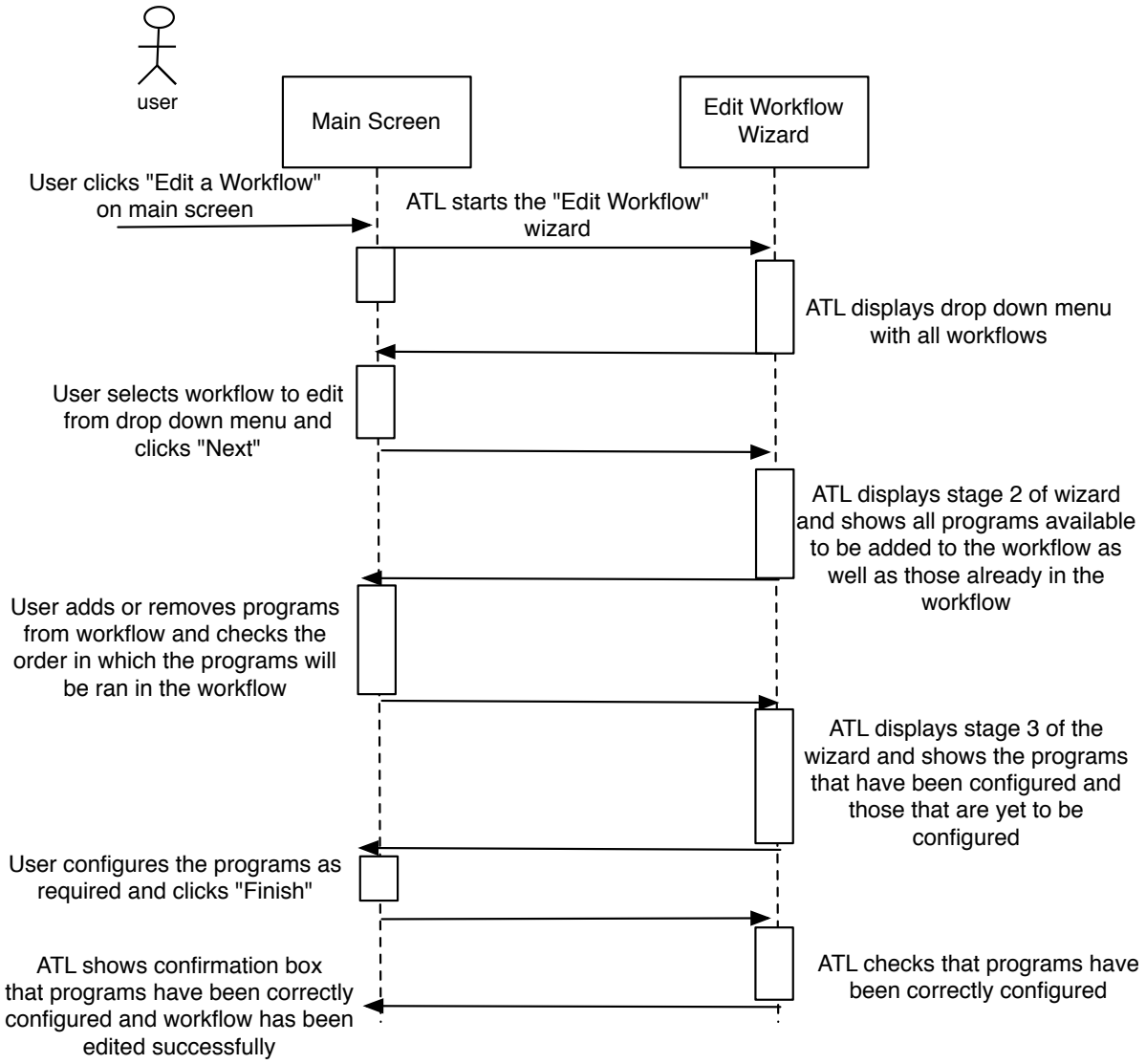


Editing a Workflow

Aero Tunnel Lab offers the option to edit any workflows that have been created by users. By doing this, this allows users to add or remove programs to be run in the workflow, without the need to create a completely new workflow from scratch, which is useful for the user.

To achieve this, the user is required to run the “Edit a Workflow Wizard”, which like the “Run Program or Workflow Wizard” and “Create a Workflow Wizard”, is broken into 3 stages. The first stage requires the user to select the workflow that they wish to edit, from the drop down menu showing all of the available workflows. The second stage shows the programs that are available and not in the workflow and the programs that are in the workflow. The user is able to use the available add and remove buttons to add or remove programs to the workflow. The third stage then shows the programs that have been configured in the workflow and the programs that are yet to be configured; the user must then configure the unconfirmed programs, in order for the workflow to then be saved. Once this has been done, the workflow will is then edited and will be available for use immediately.

The usage sequence diagram below details the scenario of the user editing a workflow (flow of applications running in sequence) using ATL.



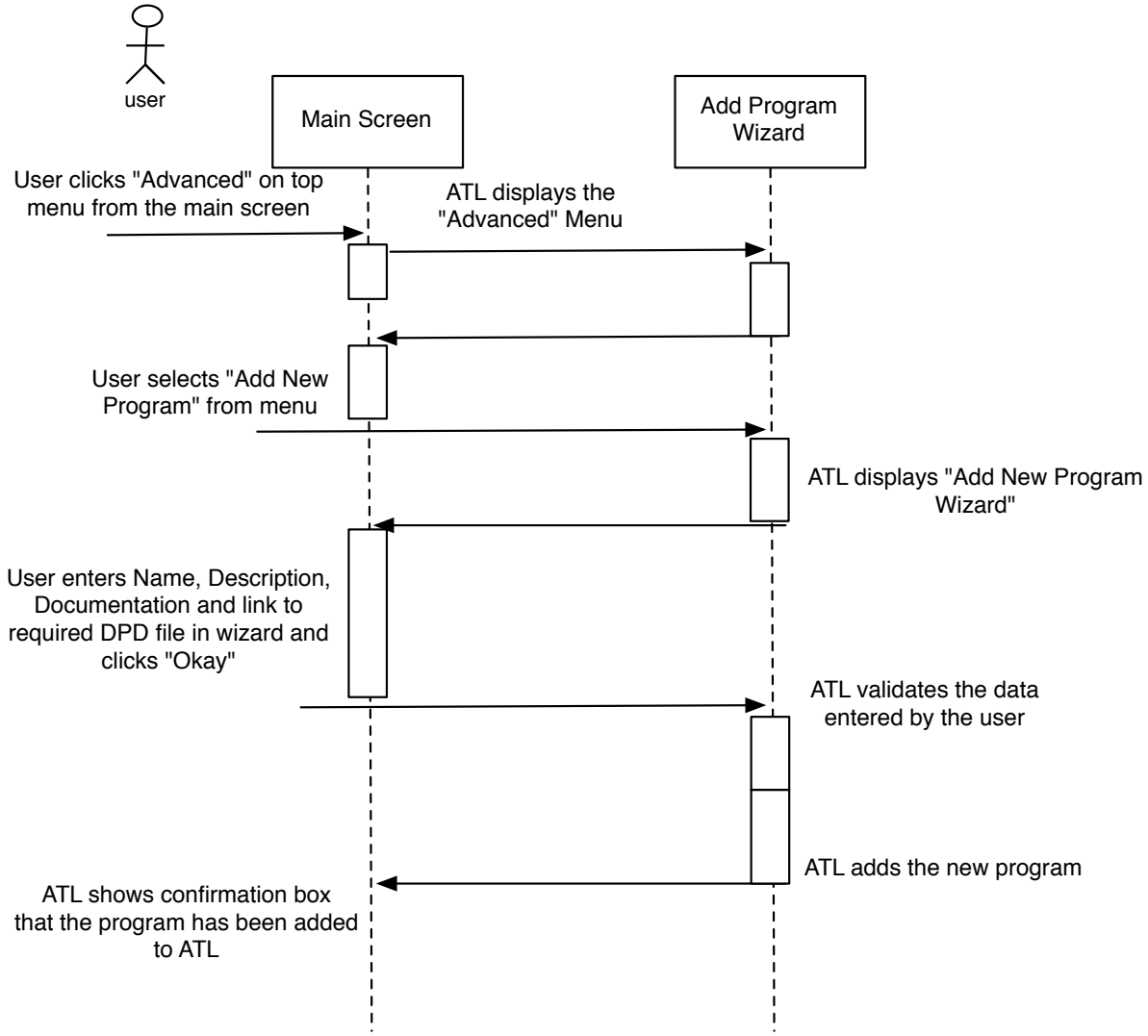
Adding a new Program to ATL

Aero Tunnel Lab offers the functionality to easily add a new program (application) to ATL, to be used in the future. This is an easy process to achieve.

The user running the "Add New Program Wizard", which is accessed by clicking the "Advanced" menu, achieves this. This wizard allows the user to easily input the required information (Name, Description, Documentation and link to the program description file[1]) and thus adds the new program to ATL, in order for it to be run in the future by all users of the system. The wizard contains validation for all fields in the wizard form, to ensure that the data inputted into the wizard is in the required format.

The usage sequence diagram below details the required process that a user will need to do, in order to achieve this.

[1] Please refer to later sections describing the program definition parser


user
Main Screen
Add Program Wizard
User clicks "Advanced" on top menu from the main screen
ATL displays the "Advanced" Menu
User selects "Add New Program" from menu
ATL displays "Add New Program Wizard"
User enters Name, Description, Documentation and link to required DPD file in wizard and clicks "Okay"
ATL validates the data entered by the user
ATL adds the new program
ATL shows confirmation box that the program has been added to ATL

3. Decomposition Descriptions

In order to make the system as simple as possible to understand, it will be broken down into lots of small, easily manageable sections. This will make it easier for team members to work on particular sections, without affecting others. This section describes how we intend to decompose the system into separate sections.

Architectural Form

From the requirements specification, it was clear that there were two key parts to the system: the user interface, and the running of the wind tunnel applications. In light of this, we have decided to use the Model-View-Controller (MVC) architecture. This architecture has the advantage that the user interface is kept separate from the core functionality of the system (i.e. running the wind tunnel applications). This allows the way we run the wind tunnel applications to be developed independently from the user interface, which means we can concentrate on implementing this before having to implement the user interface.

In accordance with the MVC architecture, the system has been split into three core areas, as follows:

Model

The application logic – The model will be split into two main components: the program model and field model.

The field model is made up of multiple individual fields, which are descriptions of the parameters fed into the wind tunnel applications via a configuration file.

The program model contains basic attributes about a wind tunnel program (the name of the program, the path to the executable) and contains the field model. It provides methods that allow the controller to enter data into the field model, and the view to access the data.

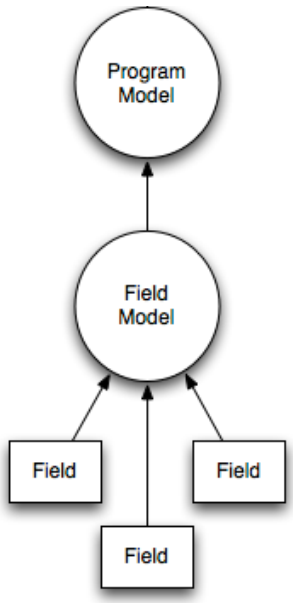


***Fig 3.1.1**– the hierarchy of the main components of the model. (It is of no specific diagrammatic form and is intended for simple/abstract explanation only)*

View

The user interface (GUI) – The system only requires a single view, which will display all relevant information on screen. In particular, it must display the wind tunnel applications and their required inputs, and allow the user to view workflows and their inputs. The view must be able to be updated by the controller, such as when the user enters parameters when running a wind tunnel application, in order to highlight any errors.

Controller

Interaction with the Model – It allows the user to enter the data required by a wind tunnel application, and run the program. It will call upon the model whenever the user enters parameters for a wind tunnel application, in order to synchronise the view and the model. It must also allow the user to edit and run workflows.

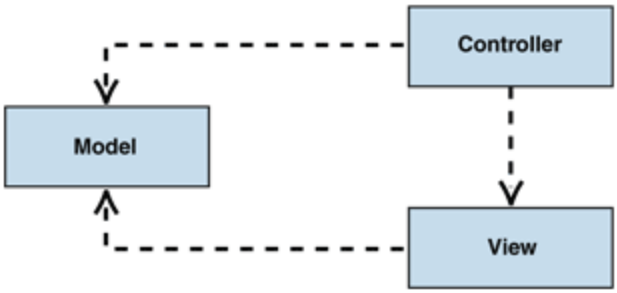


***Fig 3.1.2** – how the model, view and controller interact with each other.*[5]

Each of these main components will be subdivided into packages, which in turn will be subdivided into classes, as described in the Component Layout section. This ensures that the system is built in a modular way, which makes making changes to certain parts of the system easier.

Advantages

- The user interface is kept separate from the model. This allows the interface to be developed at a later stage than the model.
- The user interface can be changed without having to change the model.
- Testing the model can be kept separate from the testing of the user interface. This allows the more complex task of testing the user interface to be done separately.

Disadvantages

- If the data held in the model changes very frequently, it can cause the user interface to be overwhelmed by update requests.
- It is not suitable for small projects which require a very simple model.[6]

Alternative Architectural Forms

Front-End, Back-End:

This architecture splits the program into two sections, the user interface (front-end) and the application logic (back-end). The user interface handles the user input, checks it is valid, and converts it into a form that can be inputted to the back-end. The back-end then performs operations on the data. This form would not be very appropriate for this project, because it would require the user interface to validate all the user inputs into a wind tunnel application. As this is a fairly complex task, it would be better to keep it separate from the interface itself, to improve modularity.[7]

Call and Return:

This architecture uses a hierarchy of sub-programs, which are linked together through the use of method calls and return values. Therefore there is only a single thread of control throughout the whole program. This would be inappropriate for our system, due to the fact that certain features

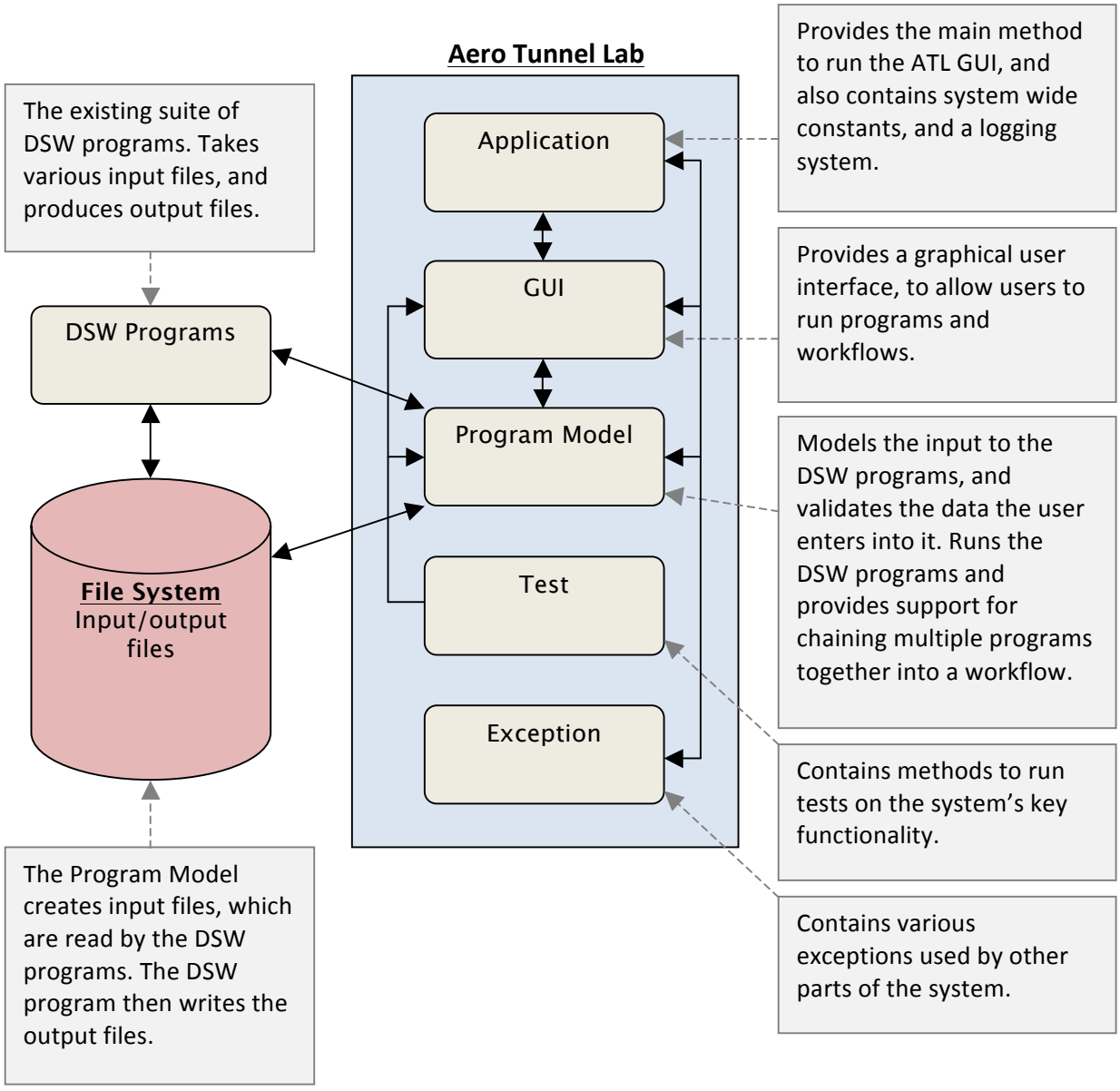
require the use of multi-threading. One such example is the way that the DSW programs will run in the background, whilst the GUI is running. [8]

Monolithic Application:

This architecture form does not use any modularity. For a system of this size it would be completely inappropriate, as it is far too complex to implement as a single Java object.[9]

High Level Overview

This section provides a brief overview of the main functionality of the system, and shows how it will be decomposed into core packages, and how these packages will be linked together. Each of the core packages shown in the diagram below will be subdivided into both classes and sub-packages as described in the Static Structure/Inter-module Dependencies section.



3. Decomposition Descriptions (continued)

The packages (and by extension, the components) of the system are split into model/view/controller via the following separation (Figure 1SD). Note how this separation reflects the high level overview (see previous figure) with the exclusion of 'Test', which is a set of compile-time only test harnesses, which are not part of the system itself

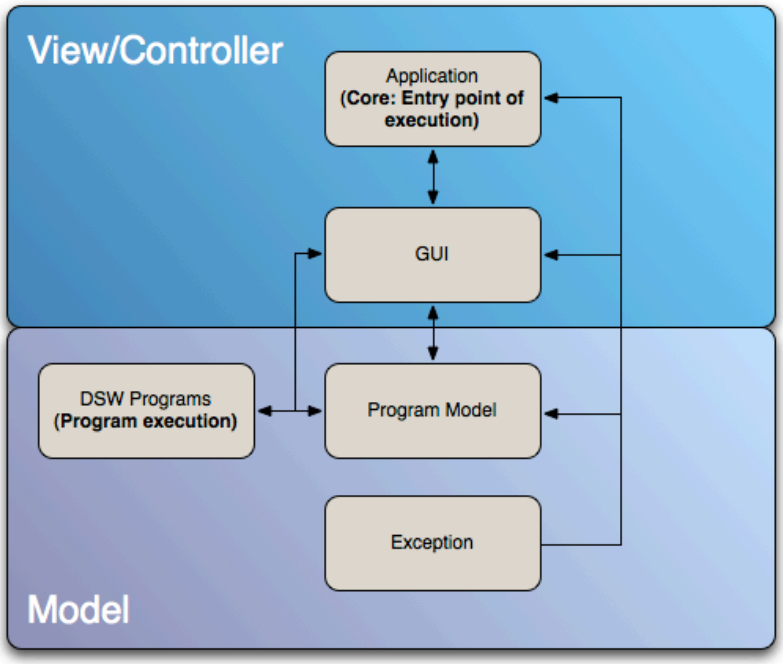


Figure 1SD

The Model

The model itself is further split into distinct components: The program model and program execution (seen in previous figures) as well as Workflow, Field Model, Fields, Update actions and constraints, and finally the program definition parser. The components within the model are also split into two layers with different functions, as described below (Figure 2SD)

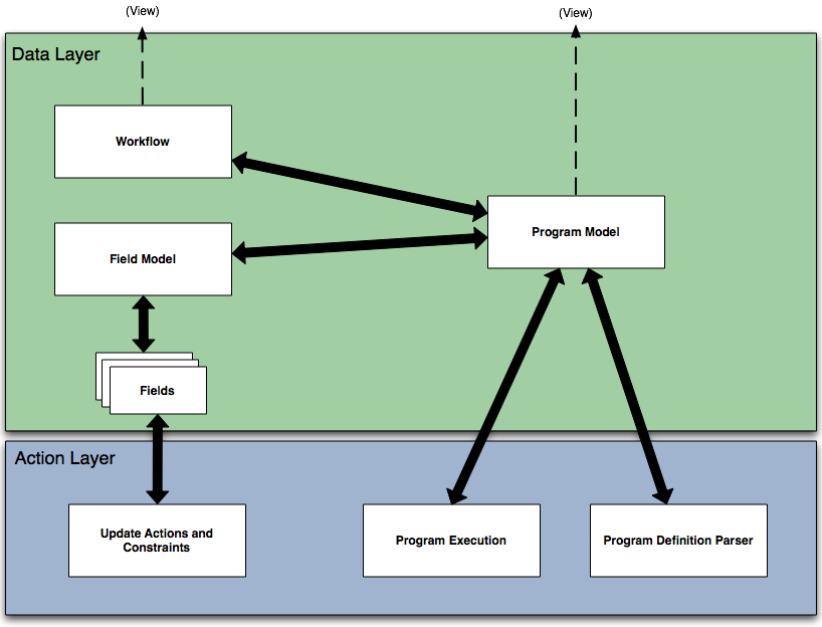


Figure 2SD

Data layer

The data layer contains exactly as the name implies: pure data. All of the components in this layer contain attributes or information but are incapable of 'acting' (whether to execute programs or otherwise) without another component to perform the work on their behalf. They *describe* actions (for example, what applications to execute and how) but do not perform them

- A workflow contains data regarding sequences of programs to run in order – that is, creating lists of program models and their associated fields and the sequence in which they must be run. *(Requirements 6.1.1.7 and 6.1.1.15– Creation of workflows as defined by requirements specification and storage of workflows. Also 'Desirable' brief requirement No. 2 – facilitating batch execution)*.
- The program model contains basic attributes about a wind tunnel program (the name of the program, the path to the executable) and contains a field model within itself (denoting the fields attached to the configuration file for the wind tunnel program in question).
- The program model makes use of the program definition parser to load its own attributes from a given filename (see Action Layer explanation)
- The field model contains a list of fields: components representing the individual parameters to be given to a wind tunnel program. The field model acts as a kind of proxy –high level actions can be performed on the field model, and it will sequentially alter each individual field to match the action performed (for example, validation can be performed on the field model – and the model will sequentially validate the inputs of each field)
- Individual fields contain values, documentation text and various other attributes. When requested (by the field model) fields can pass their values to an update action or a constraint

Action Layer

The action layer is responsible for executing actions (or loading data) for components in the data layer. The data layer components are used as containers or result objects for the action layer

- Update actions and constraints control how fields behave when the user attempts to input data (see later explanation of **Constraint**)
- Program execution uses the data contained within a ProgramModel to execute the wind tunnel applications (using the arguments and path names described)
- The program definition parser loads descriptions of a program model into memory from a text file (rather than the descriptions being hardcoded). *Fulfils Essential brief requirement No. 5 – ability to cater for future applications*

The View

The view (previously shown as View/Controller) is also split into a set of layers, just like the model. These layers and their interactions with the model are detailed below. Note that since the view is intended to be graphical in nature, it fulfils *Essential brief requirement No. 1 – A functioning and professional GUI*

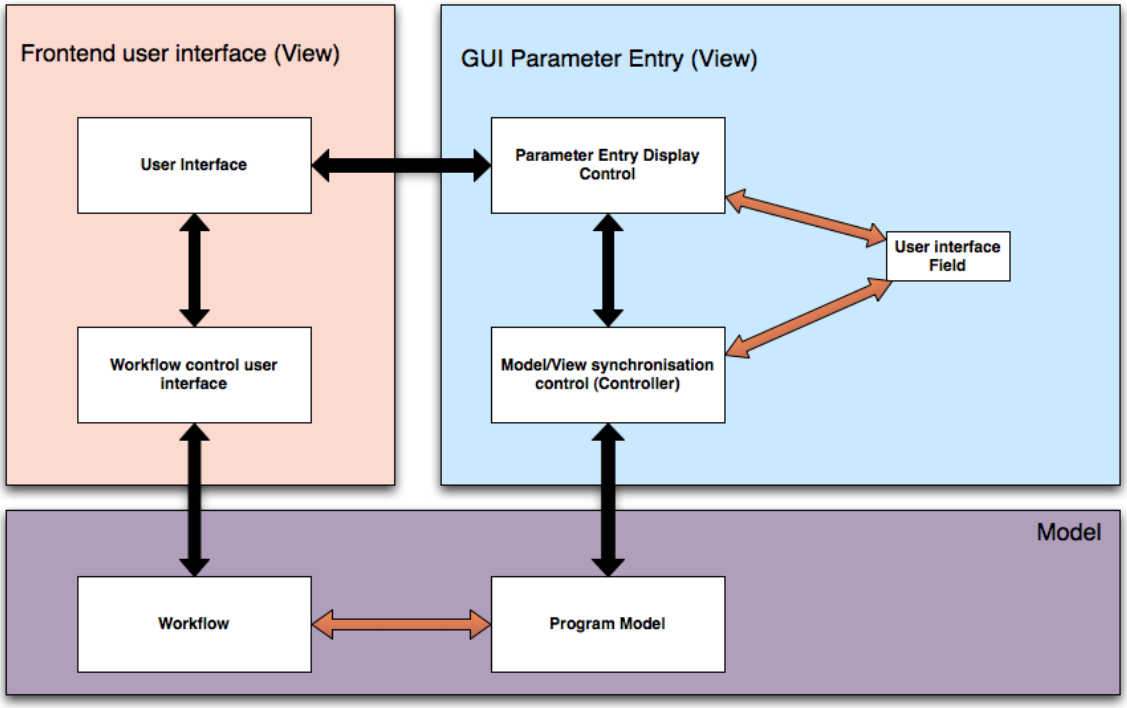


Figure 3SD

GUI (Graphical User Interface) Parameter entry

Based upon the user interface design (Section 2.1) the structure of the view has been designed so that the parameter entry (i.e. field entry) for wind tunnel programs is a separate autonomous component which can be embedded onto any other user interface (so it can exist as a 'child' of another user interface).

- The parameter entry display control (henceforth referred to as display control) is the main entry point for this component: when the parameter entry is started (to be embedded on another frame or window) it constructs the initial interface. It does this by requesting a synchronisation from the Model/View synchronisation control (controller)
- The model/view synchronisation controller is in charge of synchronising the elements being displayed with the fields described in the program model. It creates user interface controls and passes them to display control to manage. After the interface controls are built, the display control can periodically request a 'sync' between the values held in the model and the inputs held in the user interface (in either direction).
- User interface fields are objects which contain a visual presence (user interface elements) but also information about what field in the model they are associated with. The latter is a 'tag' so that when the Model/View synchronisation controller performs a 'sync' it is aware of what model field matches with any given user interface element

Frontend user interface (view)

The 'frontend' is the user interface sections other than the parameter entry for wind tunnel programs – for example, the main menu, the status screen, workflow control and logging window

- The user interface of the frontend communicates with the parameter entry (in the GUI parameter entry view). Since the parameter entry is in direct communication with the program model, it relays validation information (e.g. whether all the fields contain appropriate values) back to the frontend user interface (which can respond appropriately by displaying additional visual cues or preventing the user from continuing in a sequence of windows / dialogs). Note that this communication between two forms of user interface occurs because the parameter entry interface is designed to be **embedded within** another interface as described previously (and in this case, it will be embedded within part of the frontend user interface)
- The workflow control user interface is a section of the frontend specifically tasked with interacting with workflows and the workflow model (querying the status of workflows, commencing and terminating their execution and requesting additional parameters from the user when the workflow demands them). The workflow control interface also allows basic modifications to the workflow – program order changes and add/remove of programs from workflow *(Requirements 6.1.1.7, 6.1.1.8, 6.1.1.9 and 6.1.1.10 – Create workflows and additionally modify them – add/remove programs and change workflow order)*.
- The workflow model actually performs the actions required to create workflows – that is, creating lists (whether in files or in memory) of program models and their associated fields and the sequence in which they must be run.

3.1 Inter-module Dependencies

The Model: Data layer

The data layer classes are the most heavily used throughout the system due to the powerful validation and querying system enabled by the FieldDataModel. The layer is composed of Field, ProgramModel, FieldDataModel, Constraint and MultiValueField.

Figure 4SD

The Model: Action layer (Update actions and constraints)

The update actions/constraints component contains various different classes which act as 'plug-ins' or extensions to the field type (various different validation Constraints and UpdateActions). Note that Field, Constraint and UpdateAction are condensed on the following figure since they featured in full in the previous component.

Also, the following are constraints and update actions *available* to be attached to a field: they are not necessarily all used on every field. Whether they are used on fields depends upon the **ProgramDefinitionParser** which reads in the description of the fields from a file (see later section).

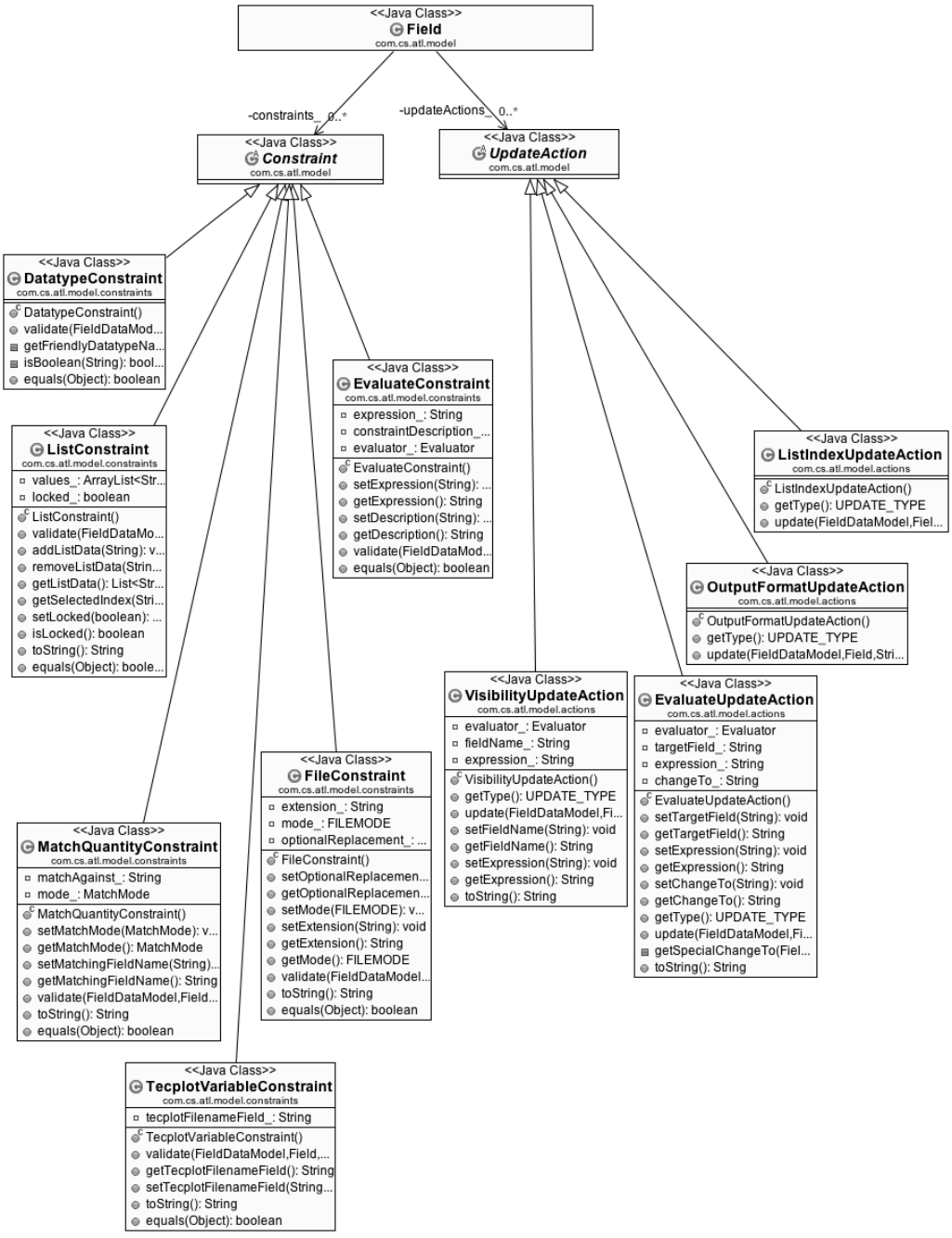


Figure 5SD

The Model: Action layer (Program Execution)

The execution component is part of the 'action layer' mentioned previously during decomposition: It deals with the actual execution of **ProgramModel** objects. Also, as will be explained later by inter-process dependencies, this class diagram demonstrates the disconnect between the queue of executions (ExecutionQueue) and an individual execution occurring (Executor) - a disconnect caused by the way in which the execution is threaded to avoid hanging the remainder of the system.

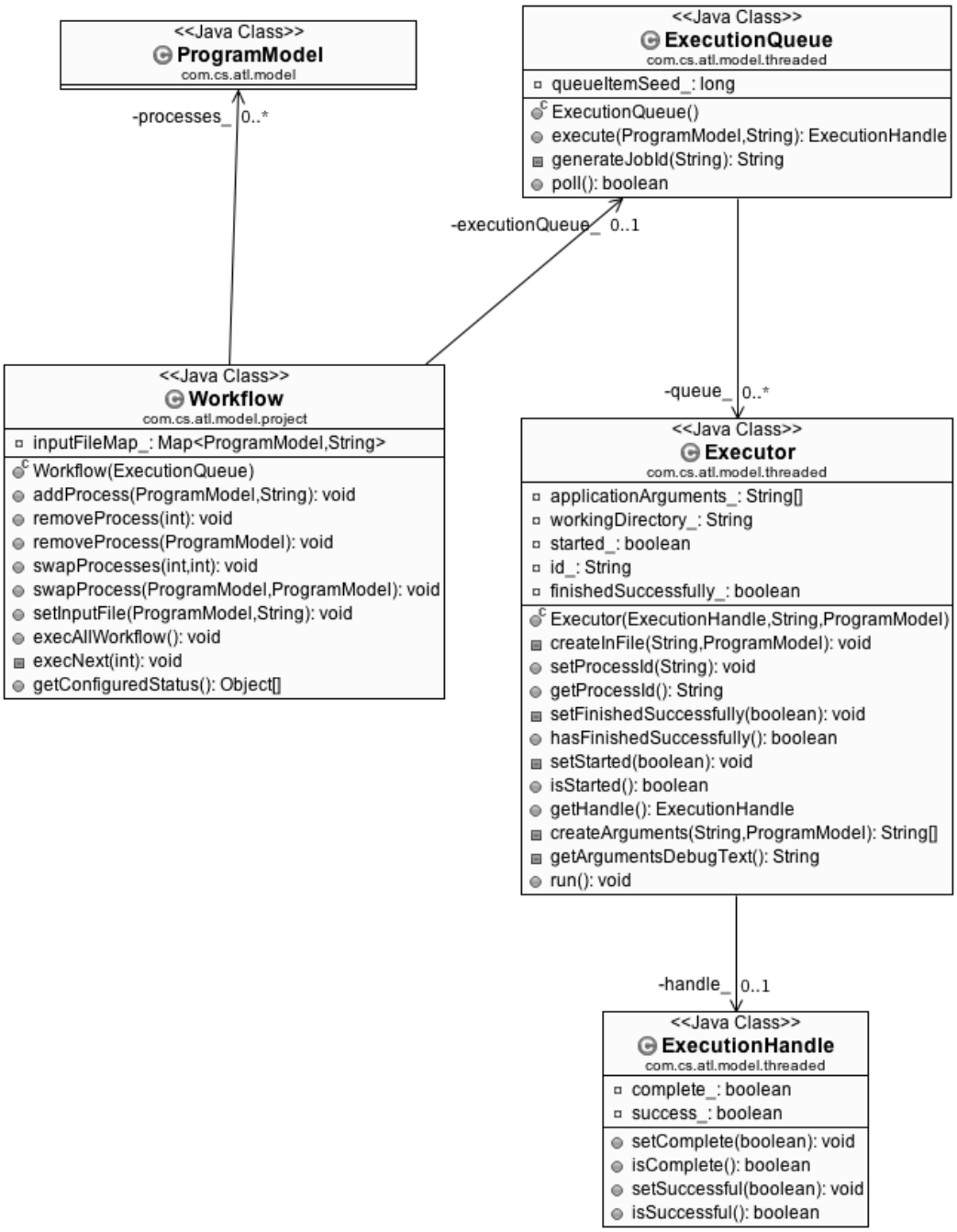


Figure 6SD

The Model: Action layer (Program definition parser et al.)

The classes within this component deal with parsing (of program descriptions and tecplot files) and generating .in files (the configuration files fed into the wind tunnel applications). Note that some classes in the following diagram are not directly connected to each other, but they are still within the same logical group as the others

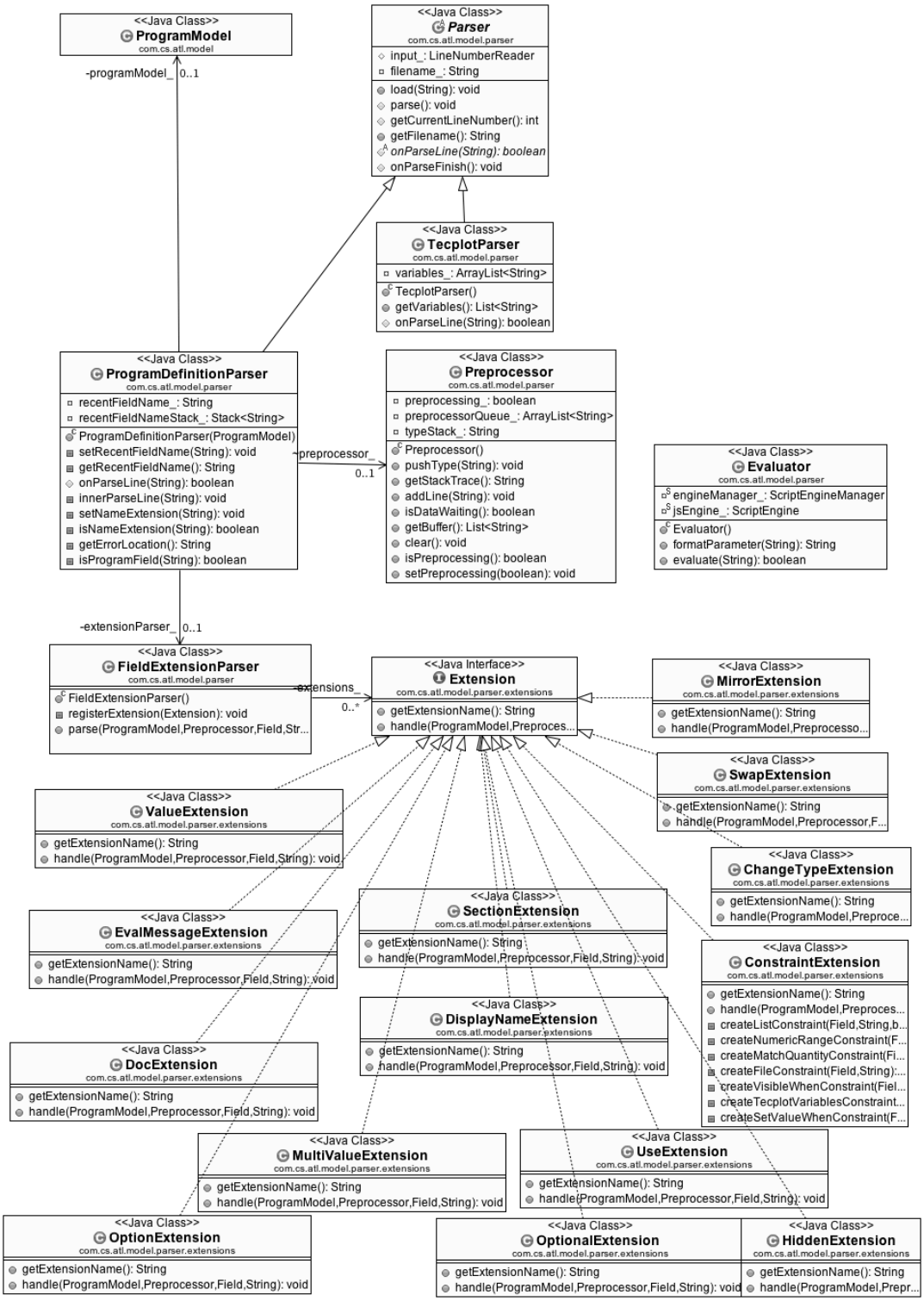


Figure 7SD

3.1 Inter-module Dependencies cont.

The View: Program (GUI parameter entry) component

The GUI parameter entry component deals solely with a single **ProgramModel** and is tasked with creating a user interface suitable for specifying the parameters required for the wind tunnel program to operate (the **ProgramModel** relating directly to a single wind tunnel application). As visible from the diagram, the parameter entry has to deal with a wide array of potential control types that could reflect the underlying program model fields.

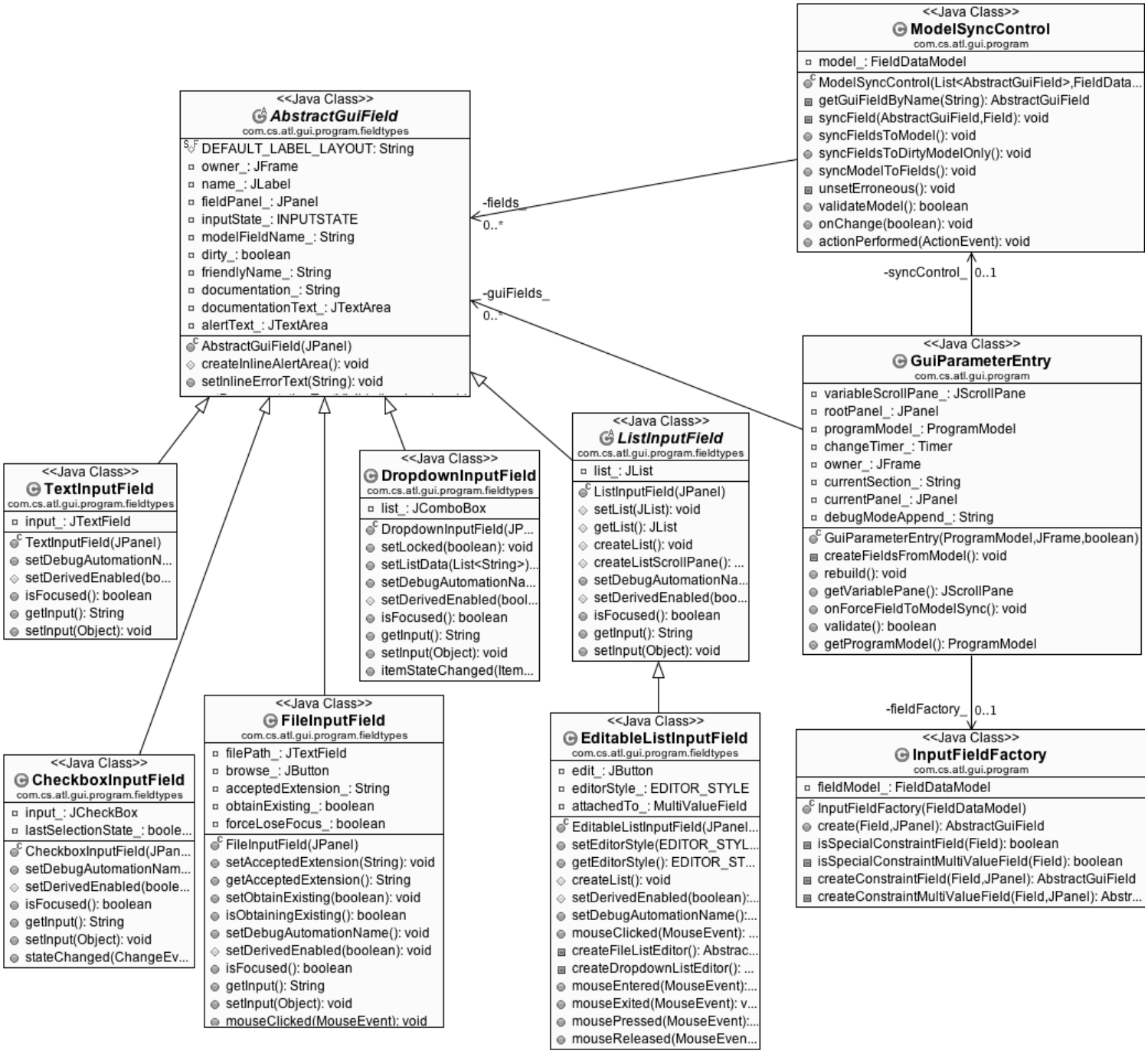


Figure 8SD

The View: Frontend component

The following all power the main user interface (distinct from parameter entry for ProgramModels) referred to as the frontend. This comprises the main screen, status screens and wizard dialogs. Note that whilst some classes are not directly attached / dependent upon each other in the following diagram, they are still logically related (and may still be instanced within other classes, but not in a fashion which renders them as a dependent or as a super class)

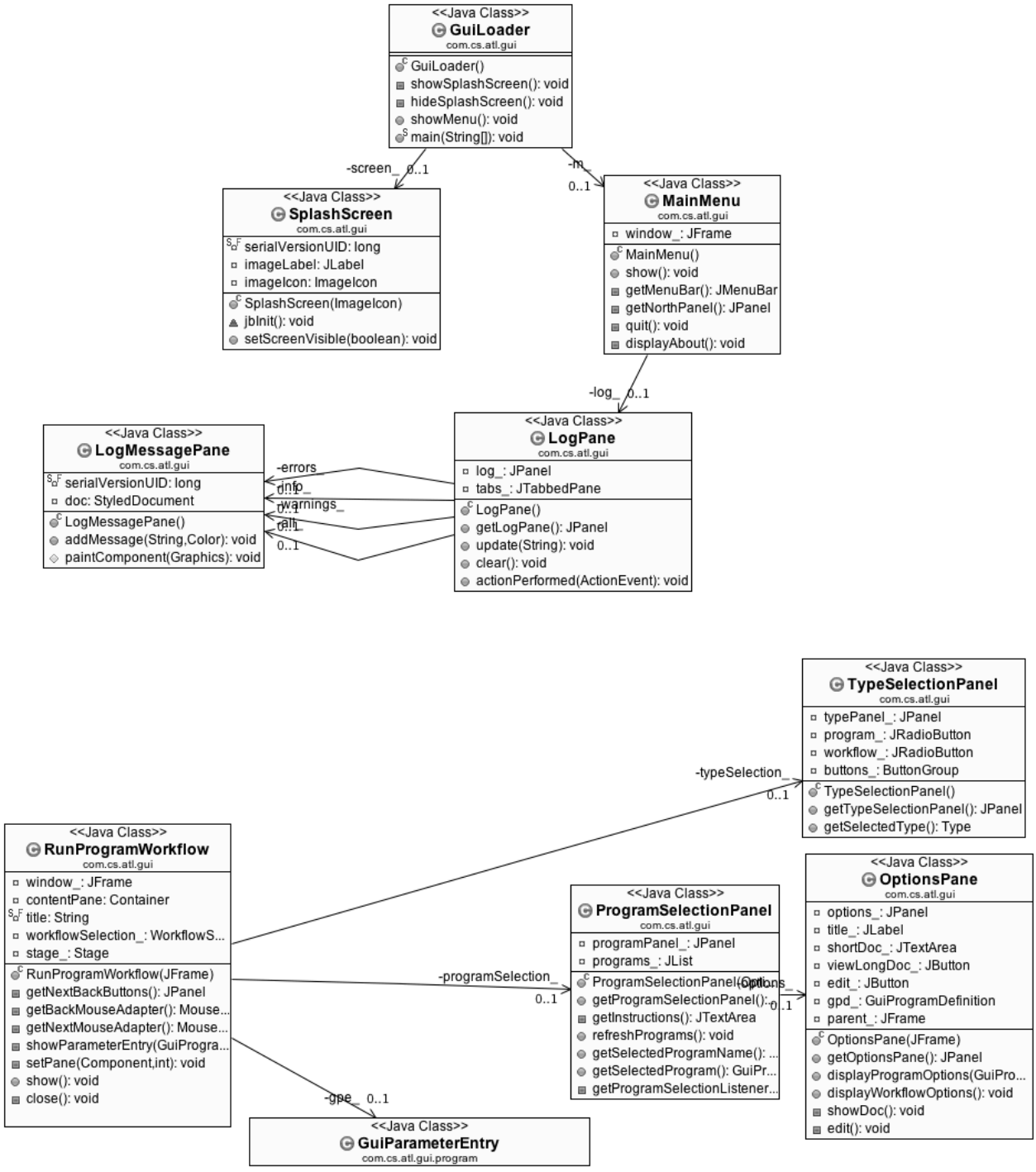


Figure 9SD

3.2Inter-process dependencies

The Model: Data layer

Class ProgramModel – As described earlier, the program model contains attributes of a single wind tunnel application. Within it is a single field model (**FieldDataModel**) as per figure 4SD. This field model contains the fields, which represent each input parameter to the program. The ProgramModel also depends upon the **Field** class, since some ProgramModel methods require iterating through the fields in the attached FieldDataModel (for example, `constructWorkingDirectory()` within ProgramModel relies on the values of fields to create the working directory path).

Preparation to generate .in files

When the action layer wishes to execute a program (by using the data provided inside the program model) it informs the program model that an .in file is about to be generated. The program model responds by constructing the working directory that the wind tunnel application will need. It obtains the working directory from the 'working directory source' – which is either the name of a field, which will contain a path, or the actual absolute path to use.

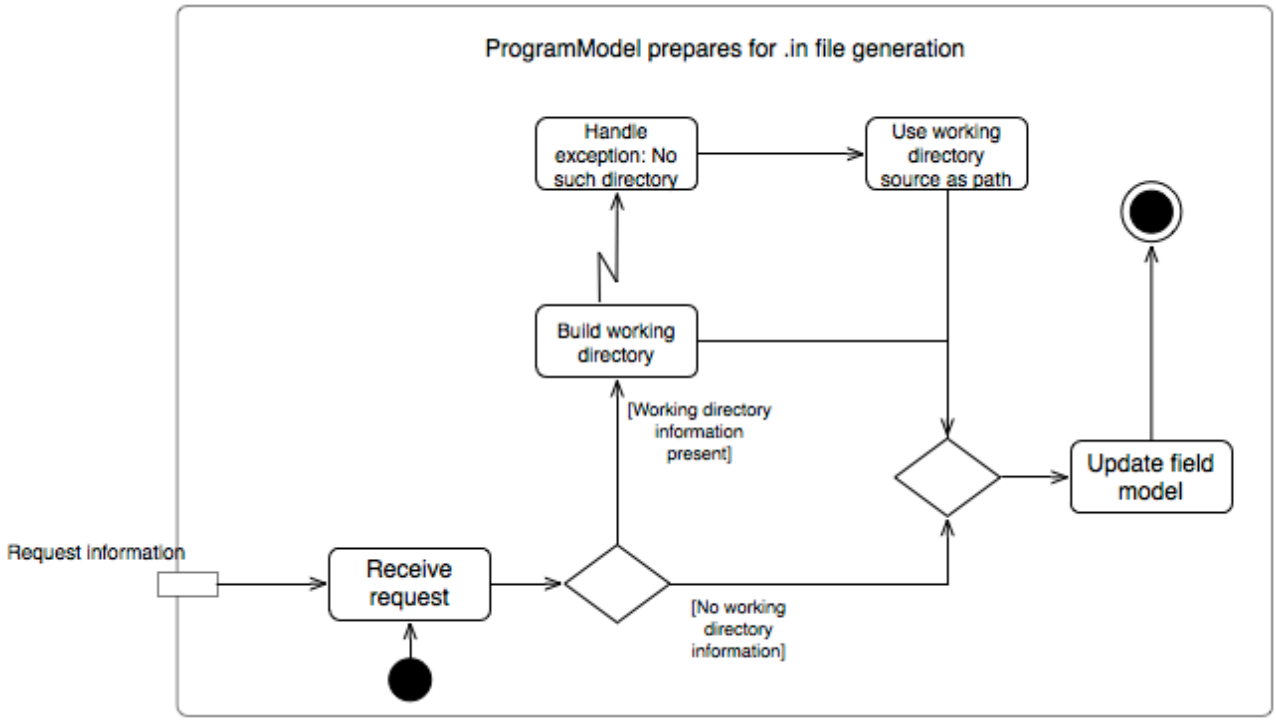


Figure 10SD

Notes for figure 10SD:

I. Request information is a status flag indicating whether .in file generation has started or not
II. The 'Build working directory' activity is the calling of `constructWorkingDirectory()` (see figure 4SD)
III. Working directory source is the variable `workingDirectorySource_` seen in the **ProgramModel** class diagram (Figure 4SD)
IV. The field model update taking place is **ONWRITE** (see explanation of the **Field** class)

The actual flow of execution (for when this activity occurs) is as follows:

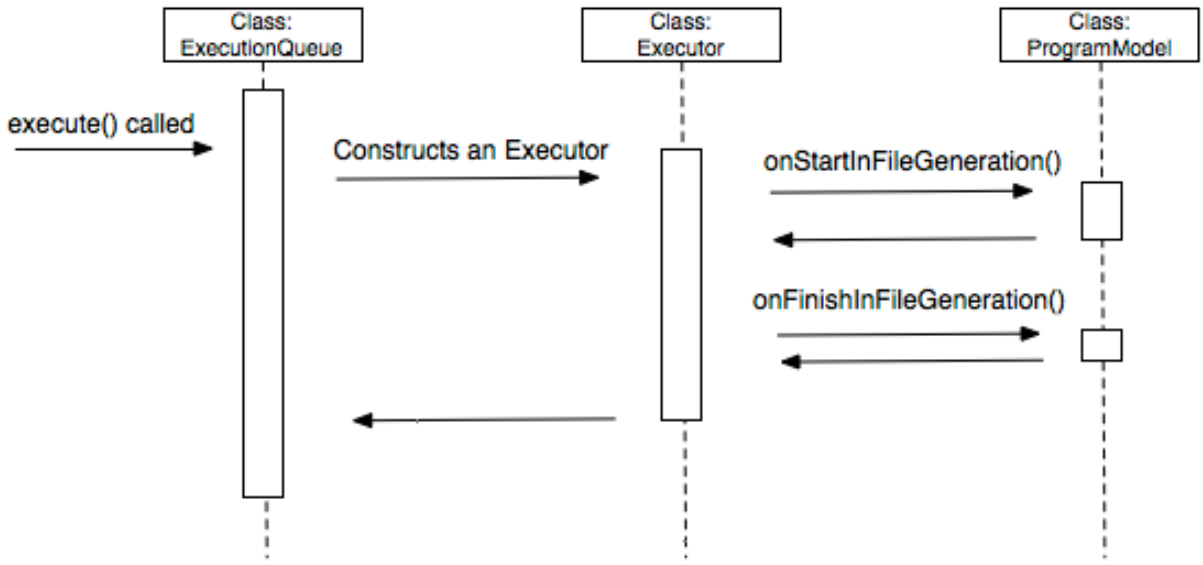


Figure 11SD

As shown in figure 11SD, the preparation for .in file generation is called from an instance of **Executor** (action layer). `onFinishInFileGeneration()` is called afterwards, which does nothing other than calling an update() of type **ONRESTORE** upon the field model (see Field class explanation)

Class Field – a single field represents a single parameter input to the **ProgramModel**. It has a value, a type and also a name and various other attributes. A field also contains a list of constraints (of type **Constraint** shown on figure 4SD). Constraints represent conditions on the value of the field which the field must fulfil in order for it to be considered 'valid'. Hence constraints enforce the validation of the field. Field also depends upon **OutputFormatUpdateAction** (see next section) which is run during an update (see below) when writing to a file

Important activities a Field instance performs during its lifetime

- Validation (iterating each Constraint applied, and checking the value of the field conforms to the requirements of the Constraint)
- Updating. Three special types of update can occur during the lifetime of a field:
    - **ONCHANGE** – something else in the field model the field is attached to has changed. The field should react and possibly change its own value if applicable. (This functionality allows real time updating of field values)
    - **ONWRITE** – The field is about to be written to a file. Since the value of a field is written as-is, the field it being told it needs to reformat itself (for example, adding quotes if it is a string). This update is performed in figure 10SD for example
    - **ONRESTORE** – The field has been written to file, now its value needs to be reverted (ready to be displayed in the user interface again)

Validation

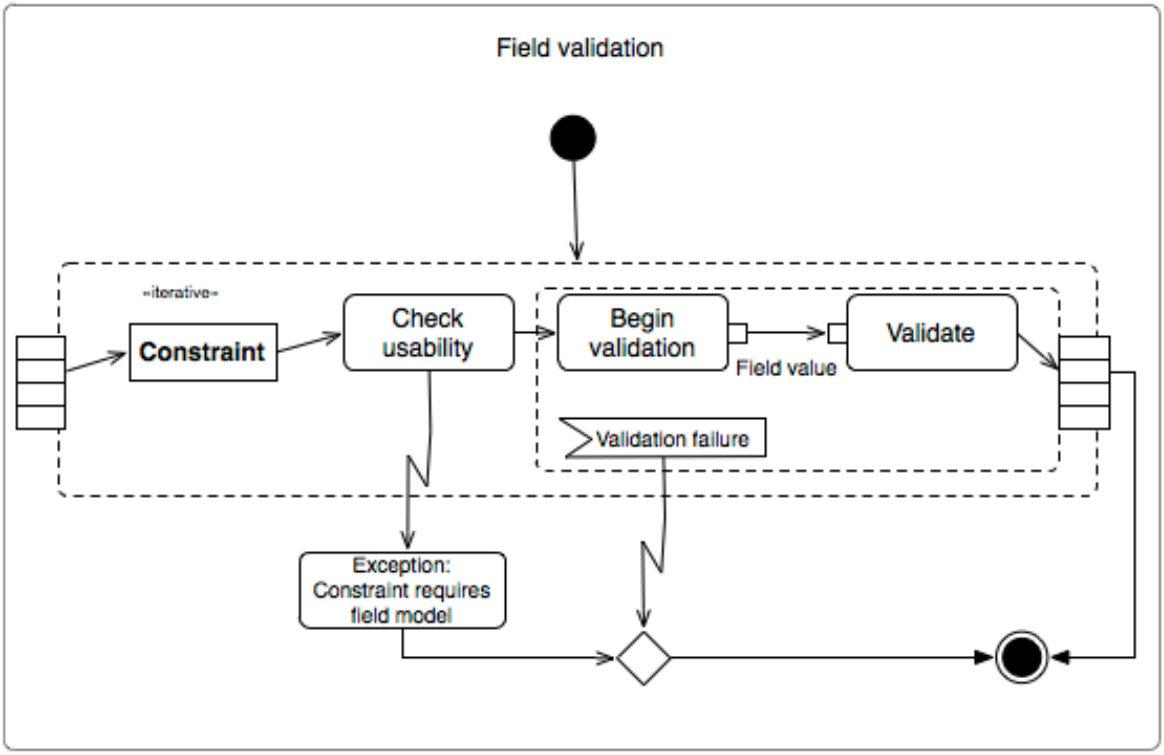


Figure 12SD

Notes for figure 12SD:

I. 'Check usability' pertains to `constraintUsabilityCheck()` in the Field class. Some constraints require access to the entire field model (for example, some constraints

check the value of the current field against another field). However, it is possible to call `validate()` on a field **without** passing the field model. Hence this check occurs and throws an exception if the constraint *requires* the field model but does not have access to it

II. If a validation failure occurs, the iteration does not continue: validate immediately terminates and returns failure. If no failures occur, the iteration eventually finishes and terminates with successful validation as a result

III. The exact mechanism of 'Validate' depends upon the Constraint being run (Constraint is an abstract class). Irrespective of the constraint, however, the field always passes its current value to it for checking, and receives a boolean result.

Update

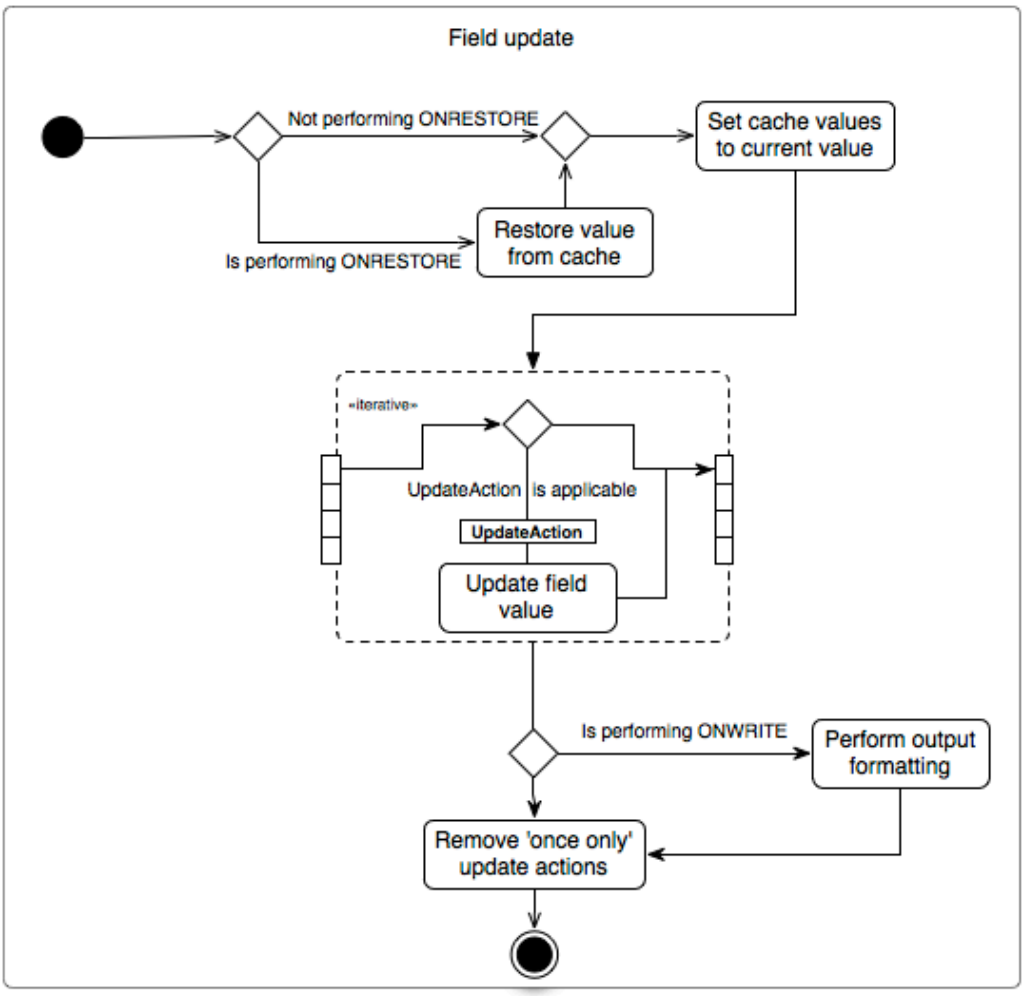


Figure 13SD

Notes for figure 13SD:

I. When an update occurs, the caller selects to either perform ONRESTORE, ONWRITE or ONUPDATE (which is why several conditions are run on the update type being performed)

II. Output formatting for ONWRITE is performed by an instance of **OutputFormatUpdateAction** (see figure 5SD).

III. Much like validation, the actual work performed for each update depends upon the update types attached to the field (UpdateAction is an abstract class)
IV. The cache referred to in the diagram is a copy of the field value which allows the field to switch between the value intended for output to a file (ONWRITE) back to the original value (during ONRESTORE)

Class MultiValueField– is an extended type derived from Field. Unlike Field, this class is designed to be a single parameter to the wind tunnel application, which holds several distinct values at once (so it is a single parameter with a list of values). It does this by reading from and writing to the same `value_` member as a normal Field, but encoding a list within it using a separator character (`$`)

Note (from figure 4SD) how MultiValueField depends upon Constraint: this is because MultiValueField implements a different method of constraint validation compared to the regular Field type (it runs through each of the values within itself and validates each one separately)

(Abstract) Class Constraint – as described above, a constraint is a condition on the value of a field that the field must fulfil in order to be considered valid. *(Essential brief requirement No. 4 – Provision for constraints upon user selection)*. Given a field value, the constraint returns a **ValidationResult** object with a boolean success flag and possible an error message (if applicable). Constraints also depend upon the FieldDataModel (an argument passed to them) so that they can look up other field values to use as a condition for validation. The exact mechanism of the Constraint depends upon the derived implementation of `validate()`

Class FieldDataModel– Contains a set of individual fields. Can perform bulk actions upon all the fields contained within itself. Specifically, the field data model can validate and update all fields:

Validation

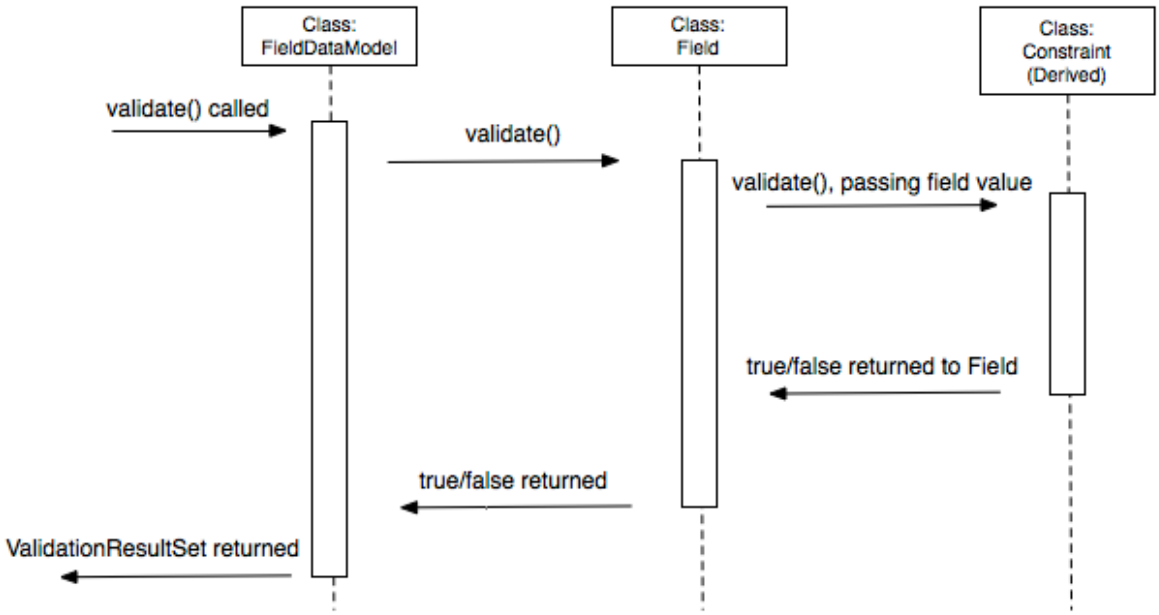


Figure 14SD

Notes for figure 14SD:

I. The field model performs `validate()` on a **series** of Field instances, each of which runs `validate()` on a **series** of possible constraints
II. `validate()` in FieldDataModel returns a **ValidationResultSet** which links the field name of each field to the true/false of whether they validated correctly
III. Constraint is abstract, so the Constraint shown would be assumed to be any class derived from Constraint

Update

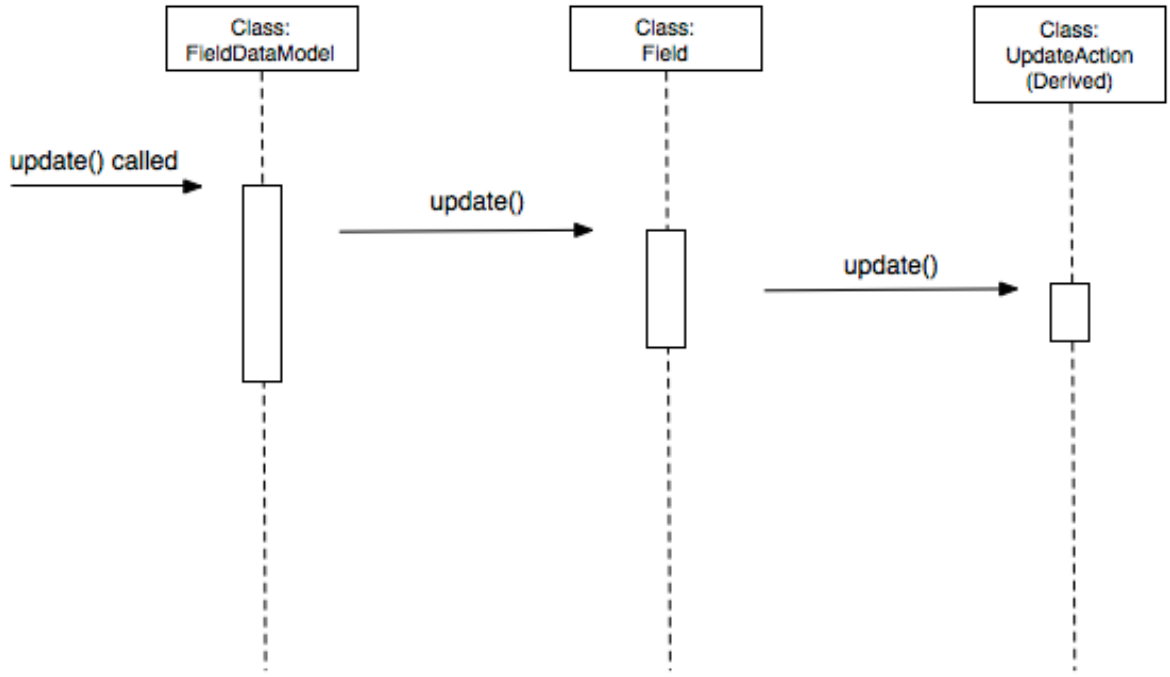


Figure 15SD

Notes for figure 15SD:

I. The field model performs `update()` on a **series** of Field instances, each of which runs `update()` on a **series** of possible update actions
II. `Update()` in all instances shown above does not return a value
III. UpdateAction is abstract, so the UpdateAction shown would be assumed to be any class derived from UpdateAction
IV. The returning arrow in the above diagram is not shown because all instances of `update()` do not return a value

(Abstract) Class UpdateAction (not pictured on 4SD due to a heavy quantity of dependencies) – Update actions are small self-contained actions which are run on the field after certain events (as seen above, they are run during ONUPDATE on an individual Field). Update actions depend upon the Field type, the FieldDataModel, and in some cases Constraint and MultiValueField (depending upon the derived implementation)

The Model: Action layer (Constraints and update actions)

Class DatatypeConstraint– a validation constraint attached to a field to ensure it fulfils the data type indicated (e.g. string, integer, floating point number). This constraint is **always** present in the field (fields add this constraint upon construction) hence the Field class has a dependency upon this class.

Class ListConstraint– a validation constraint, which ensures the field value, can only be from a predefined list of valid values (mimicking the behaviour of a dropdown list). This constraint being attached to a field is also a 'hint' to the user interface to display a drop down box.

Class MatchQuantityConstraint– this validation constraint requires the current field (assumed to be a **MultiValueField**) to have a matching quantity of list items to another field (referring to as 'matching field name'). This class depends upon the FieldDataModel passed to it by Constraint's validate method, since it looks up the other field in the FieldDataModel in order to get the quantity of items it contains

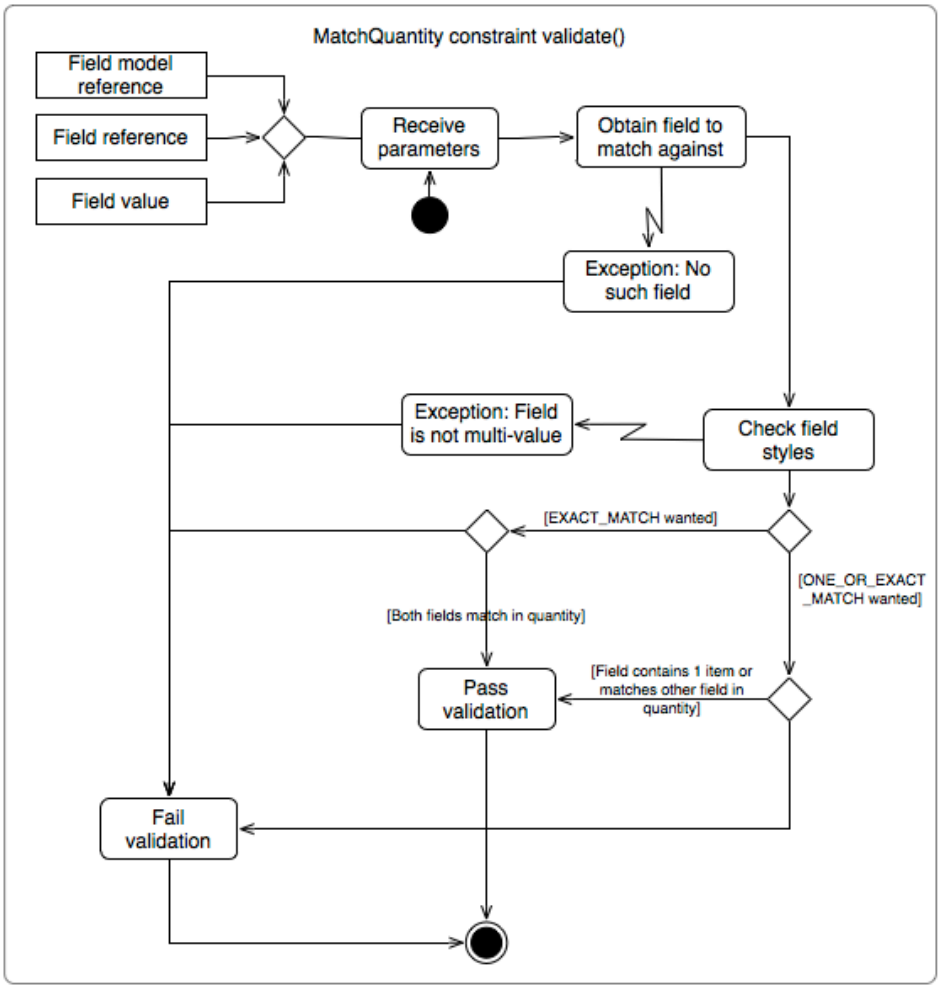


Figure 16SD

Notes for figure 16SD:

I. Only fields which hold multiple values (**MultiValueField**) can be matched in quantity (for obvious reasons)

II. There are two distinct matching modes: EXACT_MATCH and ONE_OR_EXACT_MATCH. The former is self-obvious, but the latter enforces a rule that the current field must **either** contain exactly one item, or exactly match the other field in quantity

Class TecplotVariableConstraint– a validation constraint that ensures the field contains the name (or names) of a variable from within a Tecplot data file (.tec). This depends upon **TecplotParser** in the 'Parser and .in file generation' component. It instances the parser, loads the specified file and ensures that the `getVariables()` list from the parser contains an item with the field's value

Class FileConstrunt– this validation constraint enforces a rule that the field's value must contain the filename of either a file to write (so a file with the same name must not exist) or a file to read (the file must exist). Which rule is applied depends upon the 'mode' of the constraint. This constraint being on a field is a hint to user interface to display a 'Browse...' button next to the field.

Class EvaluateConstraint– this constraint evaluates an expression in order to verify whether the field contains valid data. The expression may be a rule on the field's value (e.g. >2 &&<24) or possibly on another field's value as well (e.g. this field <= other field). Depends upon **Evaluator** in the action layer and also the **FieldDataModel** passed to it via validate.

Class VisibilityUpdateAction– When the field is updated (when the **FieldDataModel** is informed of the user having changed a user interface field by the view/controller) this update action can change whether the field is visible (to the user interface) based upon the evaluation of an expression. This is intended to allow hiding of fields, which are not appropriate given some combination of other options.). Depends upon **Evaluator in** the action layer and also the **FieldDataModel** passed to it via validate.

Class EvaluateUpdateAction– When the field is updated, this update action evaluates a given expression, and changes the field to a fixed value if the evaluation is true. Depends upon **Evaluator** in the in the action layer and also the **FieldDataModel** passed to it via validate.

Class OutputFormatUpdateAction– When a field is written to a file, it is formatted differently depending upon its type. This update action performs this formatting (e.g. surrounding strings with quotes).

Class ListIndexUpdateAction– Fields with ListConstraints (see above) may wish to be written as numbers rather than the text within the list (depending on how the wind tunnel application expects input). This update action transforms the list item text into an index number. Depends upon **ListConstraint** and also the **FieldDataModel** passed to it via validate.

The Model: Action layer (execution)

Class ExecutionQueue– the central queue for program models which are waiting to be executed. The queue delegates actual execution to a separate thread (an instance of **Executor**) and periodically checks the execution state (via the poll method). Whilst the execution is threaded (so wind tunnel applications can execute without blocking the rest of the system until they are complete) the ExecutionQueue exists to ensure only a single execution takes place at once (otherwise output data from the wind tunnel applications would be interleaved due to threading and timing issues, making the data difficult to decipher and interpret). Fulfils *Requirements 6.1.1.23 and 6.1.1.24 – program execution and echoing of console output* and *'Desirable' brief requirement No. 3– display of console output for runtime interaction*

Queuing process to be executed in ExecutionQueue

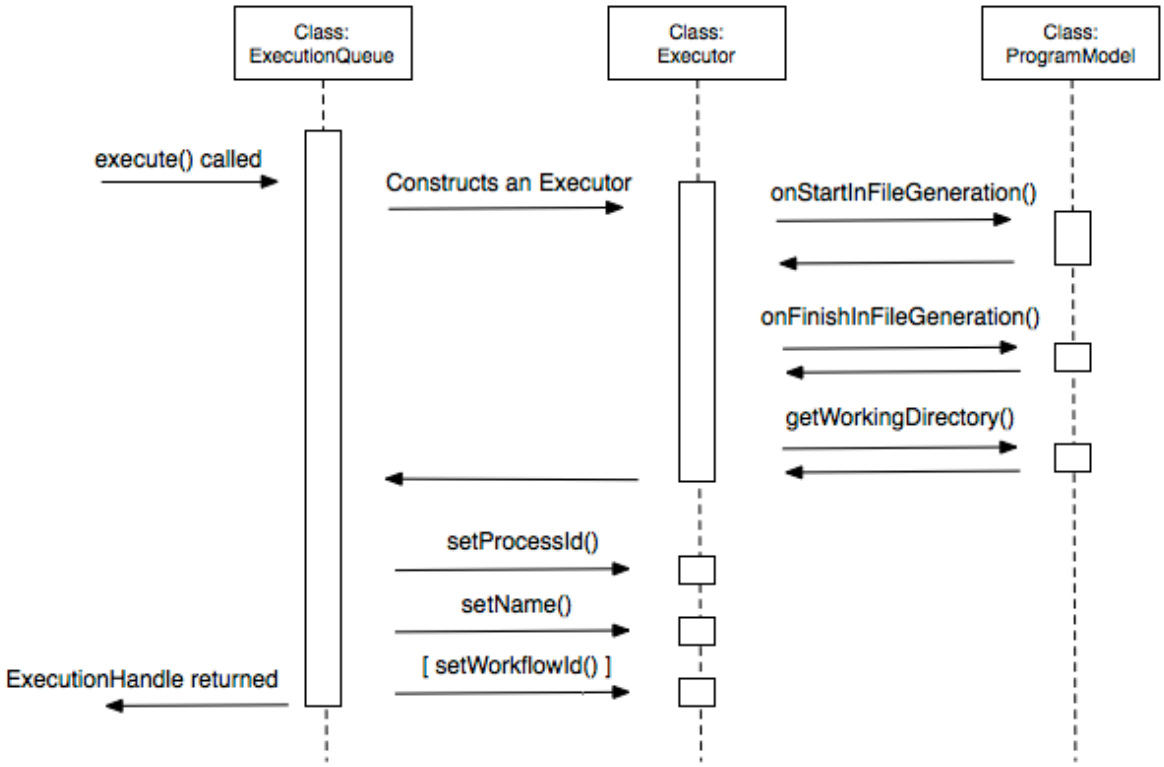


Figure 17SD

Notes for figure 17SD:

I. An Executor represents a single thread – however, the thread is not started during the call to `execute()`. It is added to a list of processes, as it is to be started later

II. `setWorkflowId()` is an optional call, performed only if a workflow ID was specified in the arguments to `execute()`

Starting and ending execution in ExecutionQueue

Since ExecutionQueue is designed to only allow a single execution at once, it needs to be polled regularly (approx. Every 500 milliseconds) so that it can invoke the next process in line after the current executor has finished

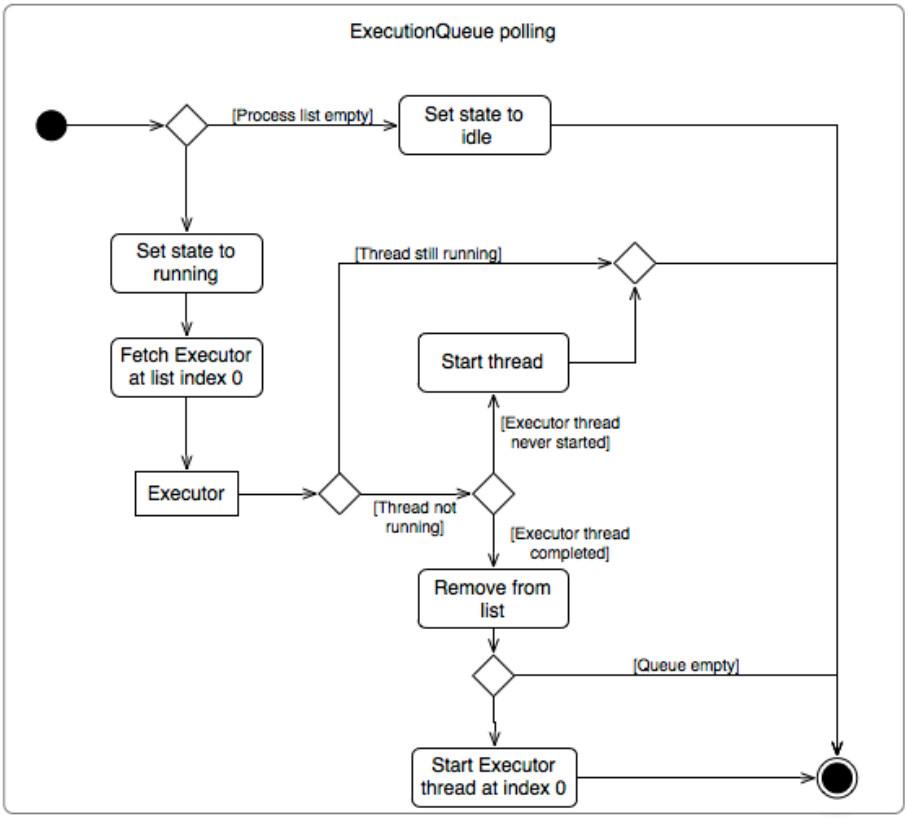


Figure 18SD

Notes for figure 18SD:

I. The ExecutionQueue will only start another thread provided the currently running thread has already finished. This enforces the 'one execution at a time' rule.

II. As soon as the current thread finishes, provided the queue isn't empty, the topmost item is then started (The list is treated like a queue, by always using the topmost item – index 0, and removing it once complete. This shifts the other items upward)

Class Executor – represents a single wind tunnel application being executed. The executor exists as a separate thread and periodically provides **ExecutionQueue** with status information upon request (as to whether execution has finished / whether it was successful). Depends upon **ProgramModel** (a program model must be passed during construction) and **ExecutionHandle** (the same instance returned to the caller of `execute()` must be passed during construction – see below).

Class ExecutionHandle– an instance of this is given to any calling code of `execute()` inside of **ExecutionQueue.** When the **Executor** tasked with the execution finishes, it will update the same instance of the object that the caller was given. This way, the caller of `execute()` can receive status and completion information about the program without having to query the ExecutionQueue. Self-contained with no dependencies.

Class Workflow – contains an editable sequence of **ProgramModel**s. The order and content of the sequence can be manipulated, and eventually executed as a batch operation (where each **ProgramModel** will be executed in the same order and sequence). Depends upon **ProgramModel** (for its internal list of processes) and an **ExecutionQueue** which must be passed to the constructor (to which it delegates actual execution of the **ProgramModel**s)

The Model: Action layer (program definition parser et. Al)

Class ProgramDefinitionParser– Loads **ProgramModel** descriptions from a file by parsing in text. *(Requirements 6.1.1.5 and 6.1.1.6 – Storage of program descriptions. Also fulfils Essential brief requirement No. 5 – ability to cater for future applications)*. The text file being loaded contains descriptions of each field (which will end up inside the FieldDataModel) and descriptions of the program itself (.EXE name, working directory to use and various other properties).

Basic mechanism

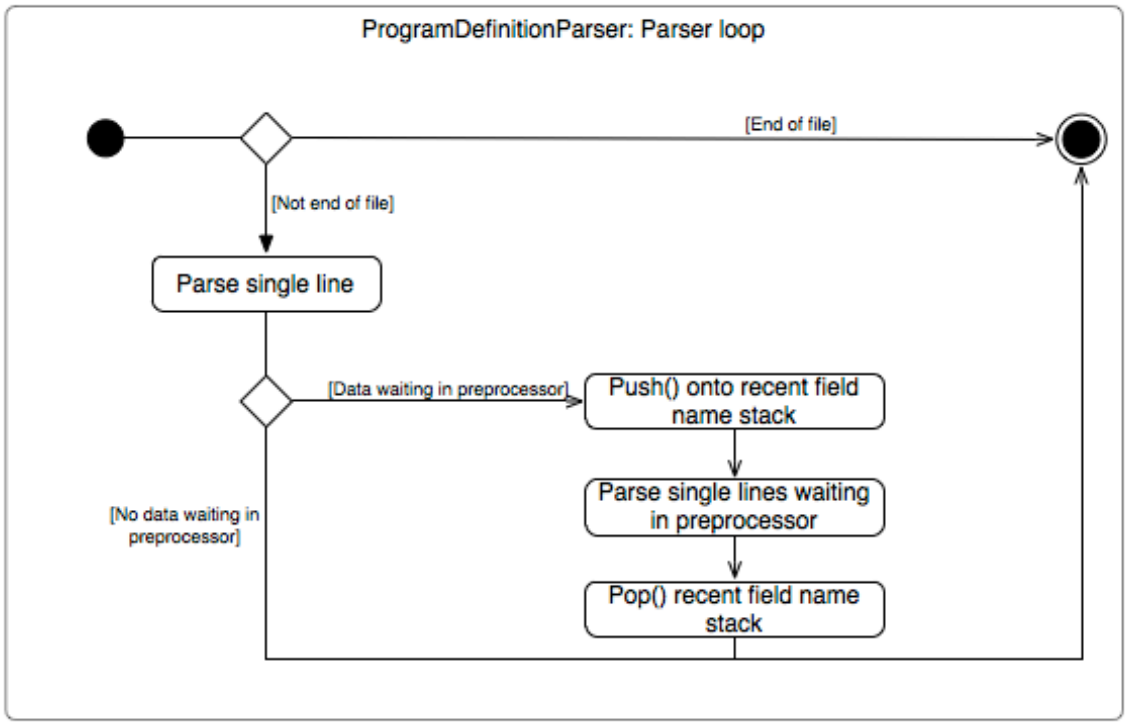


Figure 19SD

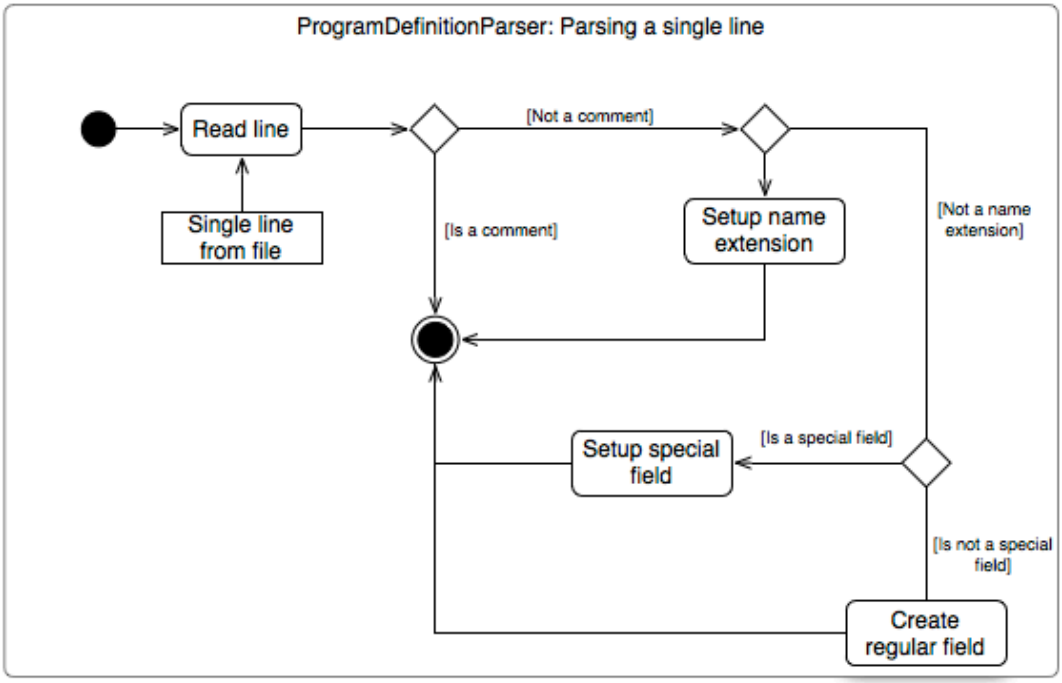


Figure 20SD

Notes for figure 19SD & 20SD:

I. The parser features a pre-processor, where certain commands can automatically expand to multiple lines of additional instructions (which themselves can also contain pre-processed instructions). The steps to check for pre-processor data are checking a string buffer for any content. The push/pop of recent field name is used for error reporting: Since pre-processing can be nested, the stack is required to keep track of the most recently loaded field

II. Parsing a single line requires checking for four distinct scenarios:
   a. The line is comment. Silently discarded
   b. The line is a 'name extension'. Words on their own in the file being loaded are assumed to be field names[2]. A name extension uses an underscore to denote a property of the field. For example: **MyField_documentation**. This is the situation the name extension check is catering for.
   c. The line is a 'special field'. Some field names are reserved words, which reflect properties, which are set directly inside the **ProgramModel, which** the parser is loading into. An example is **ATLProgram, which** is a reserved field name for defining the name of the application.
   d. The line is just an ordinary field, in which case the parser creates a **Field** object, gives it the appropriate name, and adds it the **ProgramModel**'s field model.

[2]There is insufficient space to document the entire syntax of the files being loaded – however, hopefully this simplistic example is enough to explain the workings of the parser itself

Parsing name extensions

Name extensions are not dealt with from within ProgramDefinitionParser. They are delegated to **FieldExtensionParser**.

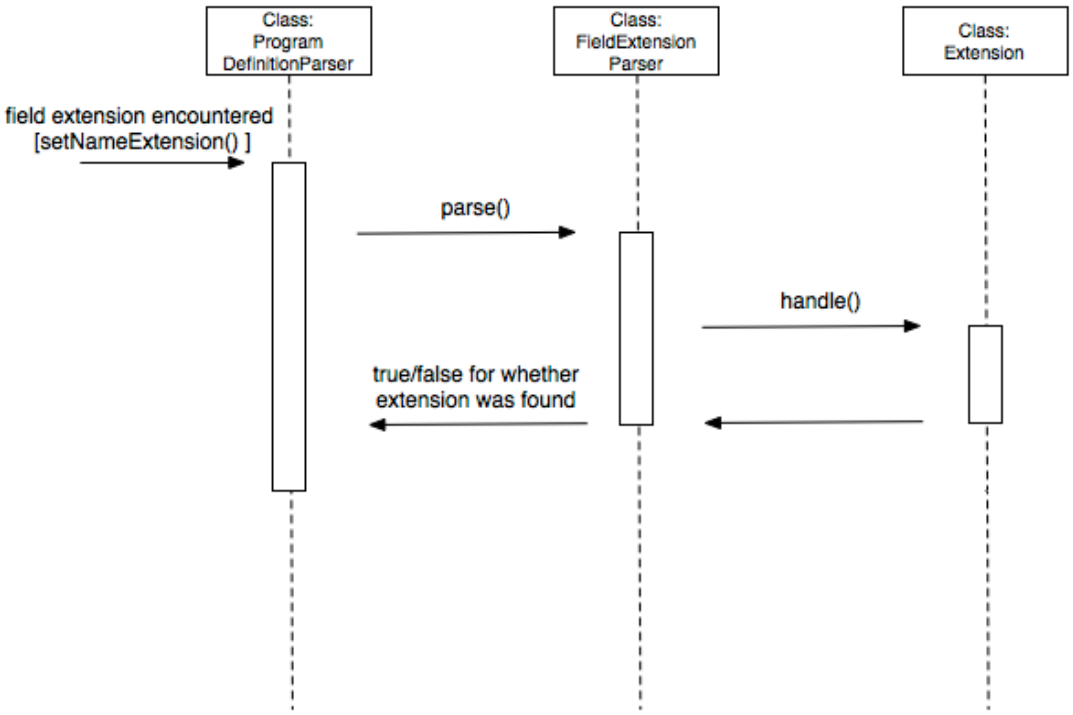


Figure 21SD

Notes for figure 21SD:

I. When the parser encounters a name extension, it splits the extension text from the field name, and passes it to `parse()` in FieldExtensionParser
II. FieldExtensionParser iterates its own internal list of Extension classes. If the appropriate extension is found to handle the extension name, it calls `handle()` and passes the extension information to the Extension.
III. If the FieldExtensionParser finds an appropriate Extension, it returns true in `parse()`. Otherwise it returns false.
IV. 'Extension' on the diagram represents any classes registered to FieldExtensionParser which are derived from Extension (abstract)

(Abstract) Class Parser – Implements basic routines for reading single lines from a file, as well as filtering commonly used structures (such as comments). Also tracks line numbers to aid in error reporting. Derived classes must implement `onParseLine()` which the Parser class calls each time a line is read from a file

Class FieldExtensionParser – as already covered this class handles 'name extensions' and their delegation to Extension instances. Depends upon **Extension**

Class Extension – an abstract class from which handlers for certain name extensions should be derived.

Class TecplotParser– Provides functionality for parsing Tecplot files (.tec). *(Requirement 6.1.1.19 – .tectecplot file parsing).* The parsing obtains the variable names from the file, and stores them within an array (accessible via getVariables()). This class is used for validation (**TecplotVariableConstraint**) and also with the use interface (when the user selects an input .tec file for a wind tunnel application, this parser is used to allow visual selection of variables in the file). **Parser** is the super class, otherwise no direct dependencies.

Class Evaluator – This class exists solely to serve **Constraint** validation – it is used by various constraint types to evaluate expressions. Evaluator does this by instancing the Java Virtual Machine's internal Javascript parser (ECMAScript) and using this to evaluate boolean expressions (returning the result to the caller). The first time the Evaluator is instanced, the Javascript engine is instanced and stored statically so that subsequent instances of Evaluator need not acquire it again (engine acquisition poses a performance penalty)

Class DswInputFile– Controls the writing of parameters to an .in file (configuration file for a wind tunnel application). This half of the .in file writer mainly deals with creating the file and preparing it for writing. It is directly dependent upon **DswInputFileHeader** which actually performs the writing. *(Requirement 6.1.1.18 – creation of .in files)*

Class DswInputFileHeader– the 'header' of the .in file is in fact the parameters stored within it. Parameter lists are added to this class, and the appropriate form for writing to file can be obtained via `toString()`. *(Requirement 6.1.1.18 – creation of .in files)*

The View: Program (GUI) parameter entry

Class GuiParameterEntry– the main component, which initiates the construction of a user interface for the wind tunnel application (based upon a single ProgramModel). An instance of this is *embedded* within another interface (it cannot be used on its own).

Is dependent upon **ModelSyncControl** (it uses an instance of this to synchronise the model and the user interface at a fixed interval) and **InputFieldFactory**

Creation of the user interface

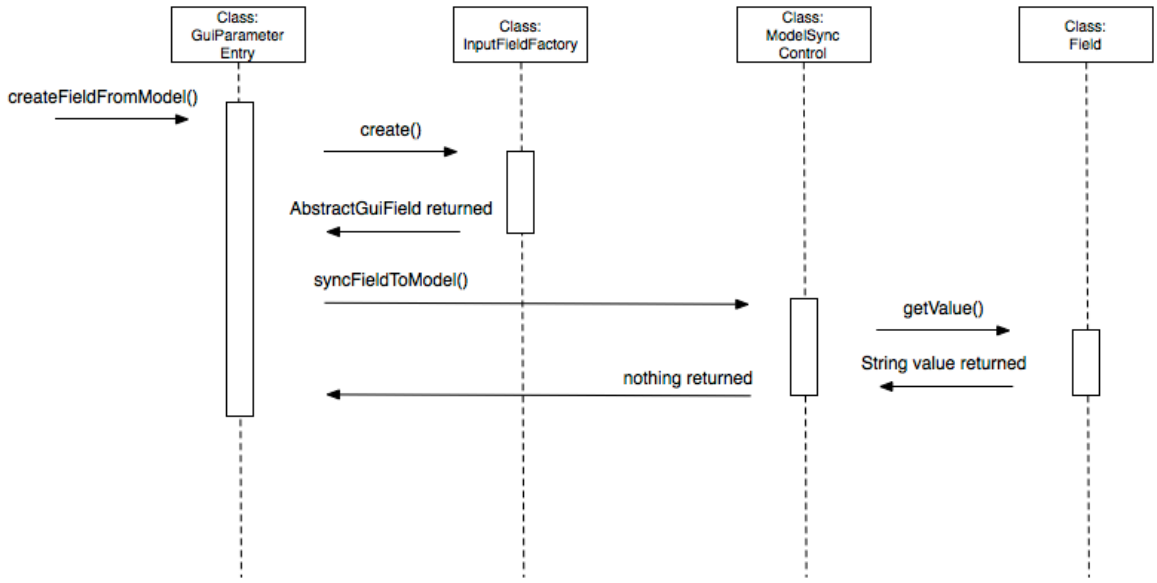


Figure 22SD

Notes for figure 22SD:

I. The `create()` method is called many times (for as many fields as there are in the model) not just once. `syncFieldsToModel()` on the other hand, is called only once (but cycles through multiple fields)
II. `syncFieldsToModel()` synchronises all the user interface fields (AbstractGuiFields) to the values held in the model (the Field object in the model that the user interface field is paired with)
III. Each individual synchronisation involves retrieving the value of the model field (Field) via `getValue()` and setting the user interface field to the same value.

Validation of the user interface

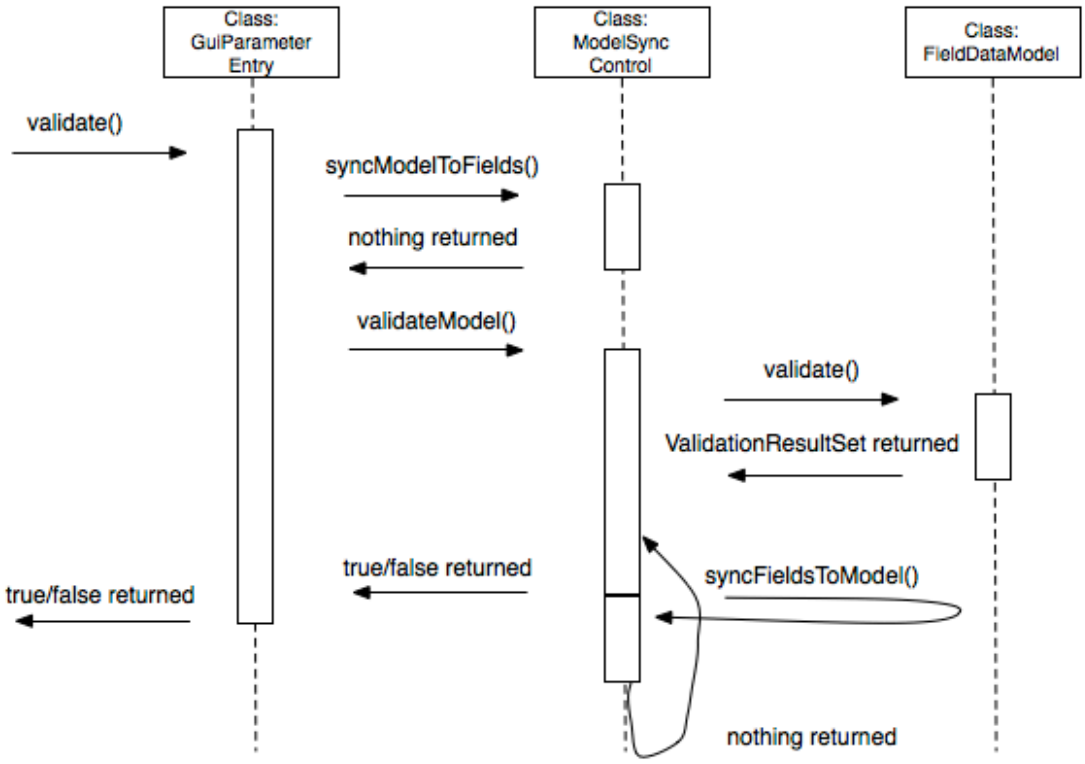


Figure 23SD

Notes for figure 23SD:

I. The `syncModelToFields()` synchronises the underlying model to the values held in the user interface (as opposed to the inverse, `syncFieldsToModel()`)
II. `validateModel()` causes the validation of the underlying FieldDataModel.
III. ModelSync control calls `syncFieldsToModel()` inside of itself, after validation is complete.
IV. `validateModel()` returns a true or false flag for whether validation succeeded
V. `validate()` returns the same true/false flag given by `validateModel()`

Class InputFieldFactory– creates user interface fields (**AbstractGuiFields**) when given a **Field** from the underlying model. It decides the user interface field to create based on the datatype of the model field, as well as its style (single value vs. Multi value) and the validation constraints in use (e.g. FileConstraints prompt the factory to create a filename input with a browse button)

Class ModelSyncControl– performs synchronisation between the user interface fields and the model. This can be done in either direction. Also contains a trigger for informing the model as to when the user has changed values on the user interface (some fields can dynamically hide and show themselves based upon these 'OnChange' events)

Class AbstractGuiField– the super-class for all types of user interface field. Contains basic methods to associate a user interface element with a model field, as well as default settings for creating labels and preset layouts

Class TextInputField– a user interface element which accepts a single line of text as an input. Created in response to any model field which is single value. Derived from **AbstractGuiField**

Class ListInputField– a user interface element which accepts multiple values of any item (displayed as a list). By default an 'Edit' button is also provided. Created in response to any model field which is multi-value. Derived from **AbstractGuiField**

The View: Frontend

Class GuiLoader– in charge of creating the main screen, this class is dependent solely upon **MainMenu** which it instances. Also displays a splash screen and initiates the look and feel of the user interface prior to its creation. Has no interaction with the model or the **Program** package/component

Class MainMenu– represents the main user-facing screen of the interface (Section 2.1). Main responsibility is to construct the distinct 'regions' of the main screen (top and bottom, for example). Depends upon **LogPane** which it instances as the bottommost region of the interface. Has no interaction with the model or the **Program** package/component

Aero Tunnel Lab user interface lifetime (initiated by GuiLoader)

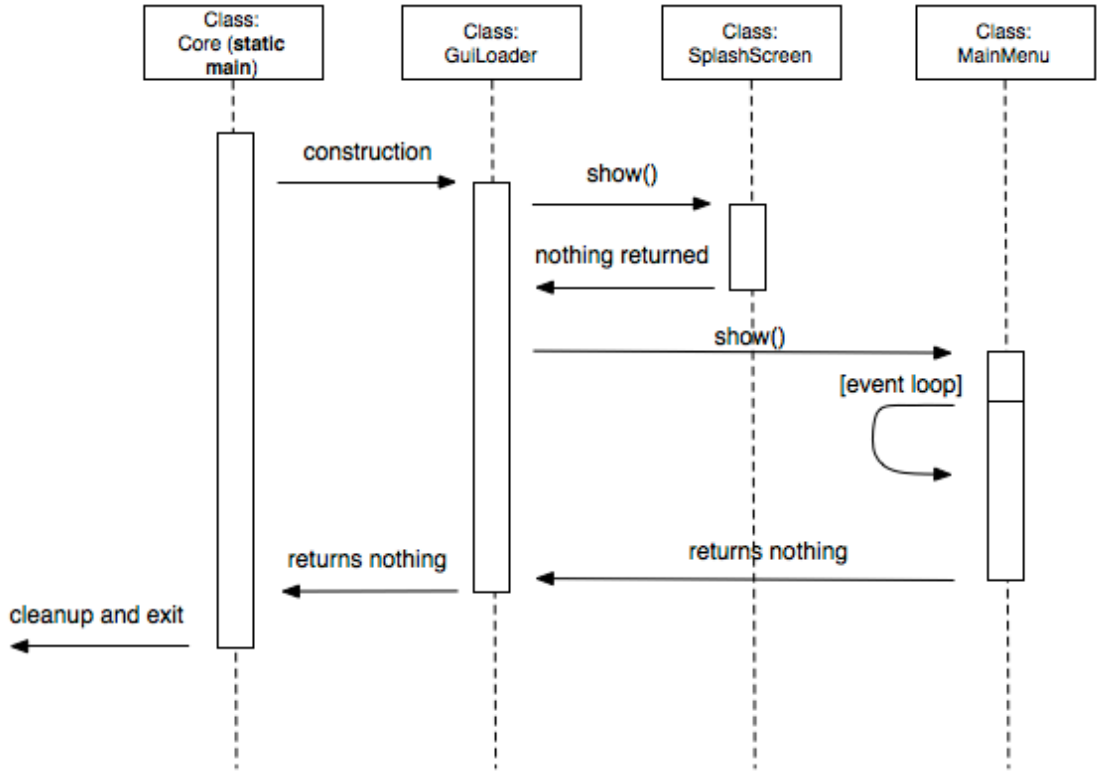


Figure 24SD

Notes for figure 24SD:

I. Core is a class which houses only a static `main()` method (and simply instances GuiLoader)
II. '[event loop]' inside of MainMenu is a user interface event loop which runs until the user quits Aero Tunnel Lab. The main menu itself can invoke actual execution of programs during its lifetime

Class LogPane– displays log messages in the user interface. This works via a subscriber pattern, whereby the LogPane subscribes to receive log events. The log itself is visually updated every 500 milliseconds (every half-second). The application log is within a static class entitled **Core** which this class is dependent upon

Class RunProgramWorkflow– a wizard-like dialog in charge of guiding the user through the process of running either a single wind tunnel application or a workflow (batch execution of multiple applications). When the user requests a specific wind tunnel application be run, an instance of **GuiParameterEntry** is embedded within the user interface – hence invoking interaction with the model layer (and the **GuiParameterEntry** acting as a proxy between the frontend user interface and the underlying model). **RunProgramWorkflow** waits until **GuiParameterEntry** returns positive for field validation until it allows any application to be executed

Invoking the display of a parameter entry

(So that the user can run a wind tunnel application)

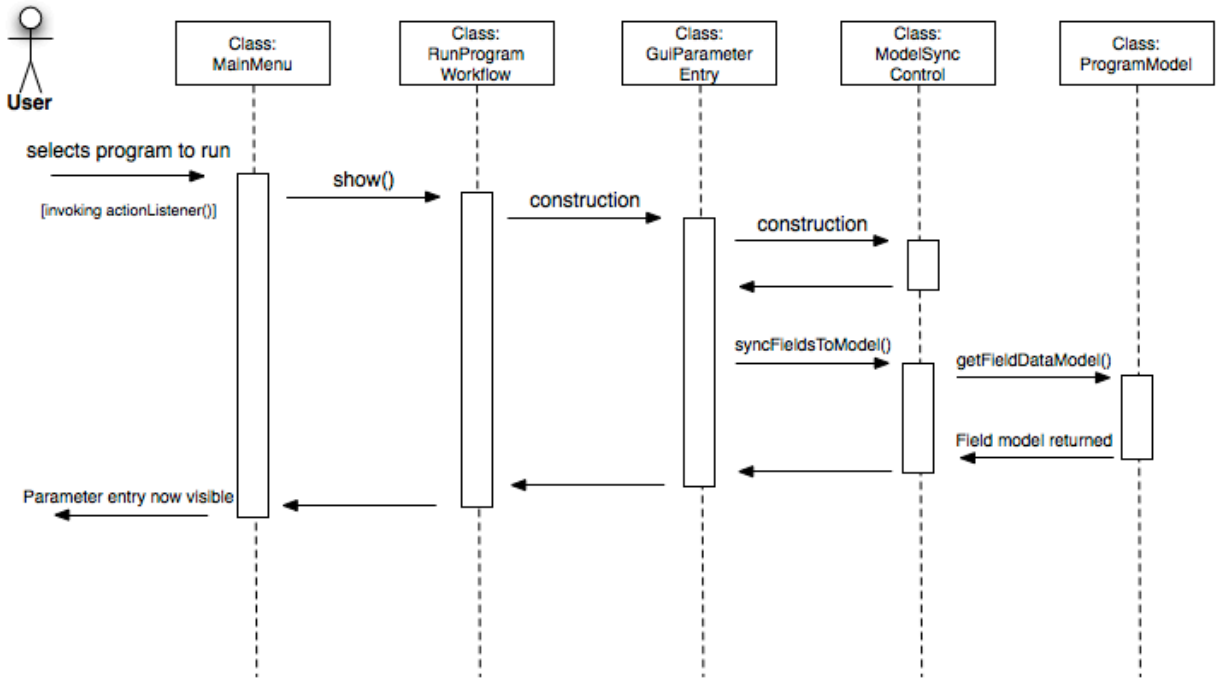


Figure 25SD

Notes for figure 25SD:

I. The field model is acquired from the ProgramModel so that the ModelSyncControl may iterate the fields within it and synchronise the UI to each field

Validation within the parameter entry

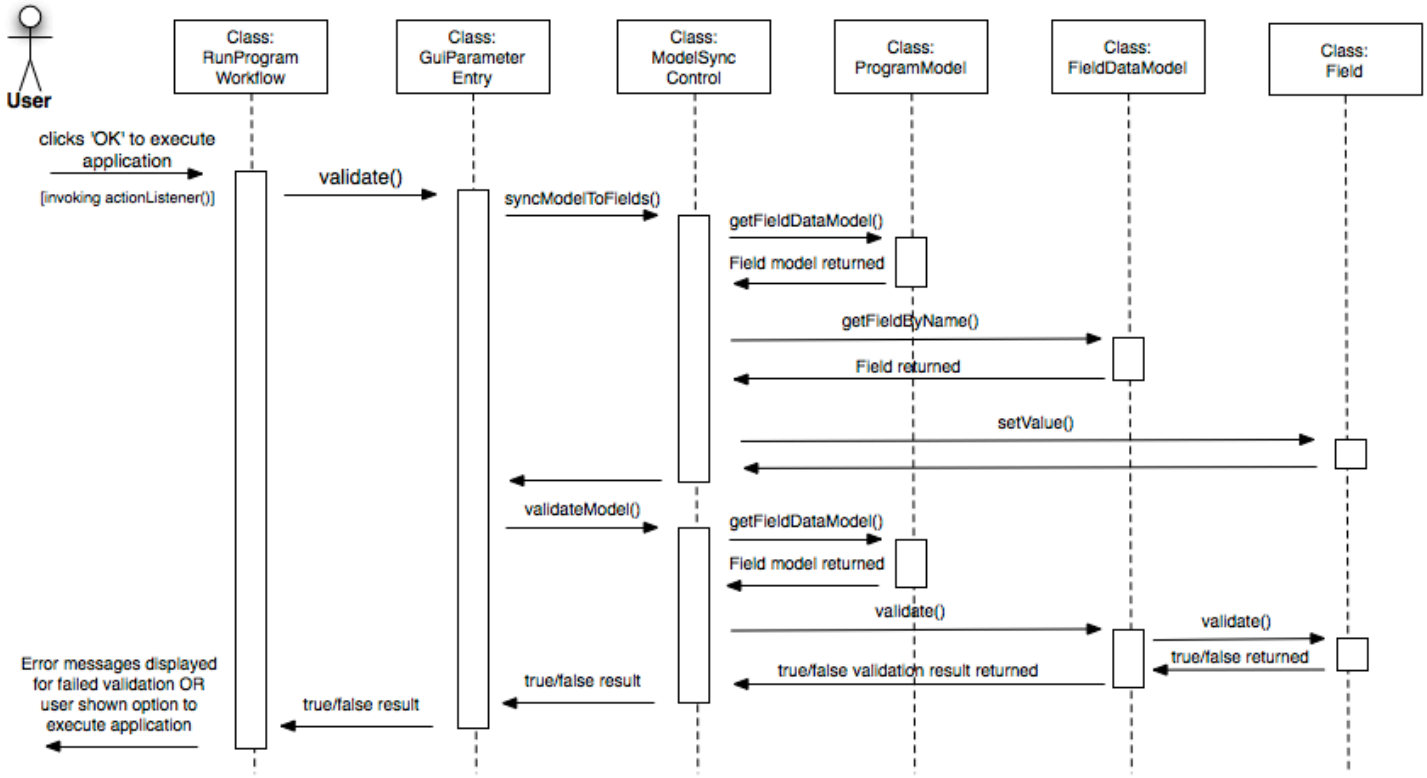


Figure 26SD

Notes for figure 26SD:

I. Validation occurs when the user attempts to run the application via 'OK' (or Finish as per Fig. 2.1.4)

II. Validation synchronises the model to the user interface fields and then goes about validating each model field in sequence (the Field `validate()` shown would be performed for every Field)

III. If the validation is successful, the wind tunnel application will be executed (see Fig. 27SD) otherwise error prompts will be displayed to the user alerted them to the problems in the field values

Execution of a program from the main menu / RunProgramWorkflow dialog

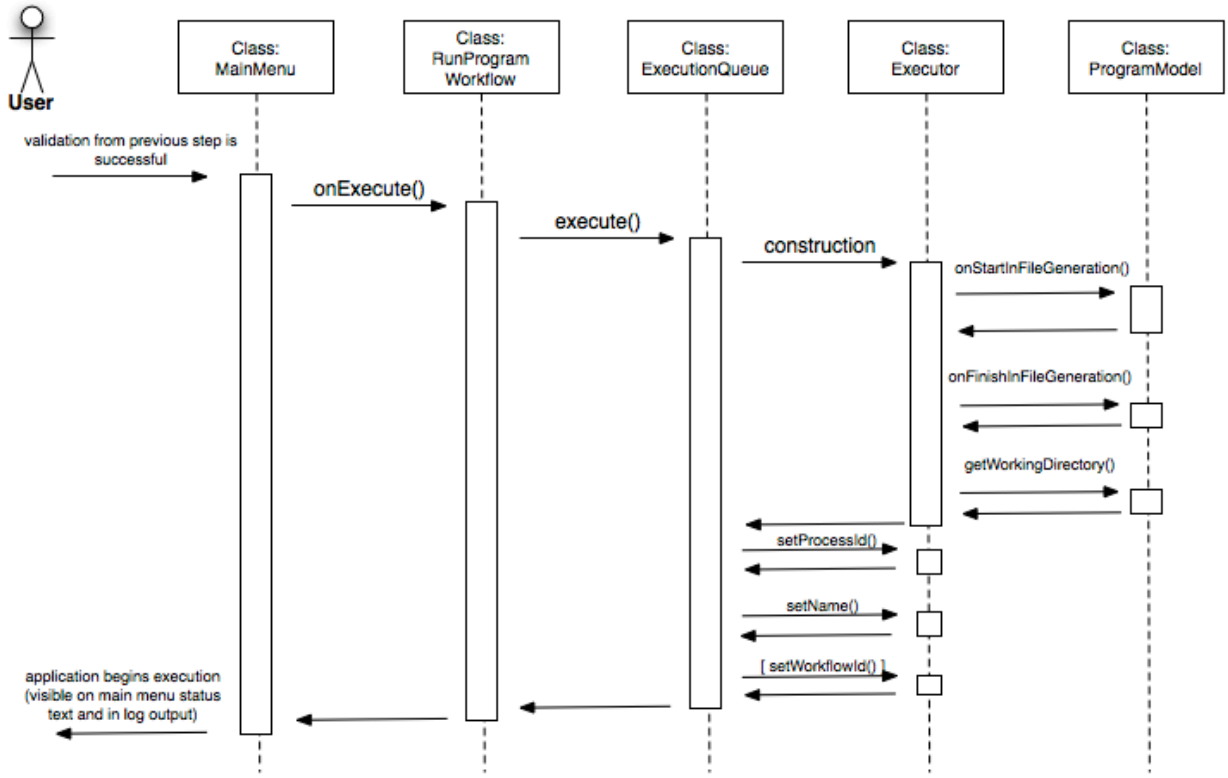


Figure 27SD

Notes

I. The Executor is created based upon the ProgramModel provided by GuiParameterEntry
II. ExecutionQueue is persistent – it can be accessed anywhere (it is a singleton and will only be instanced once throughout the entire system)
III. Actual execution will not begin until the next call to `poll()` inside ExecutionQueue (which is attached to a timer inside MainMenu to be called every half second [500 milliseconds])