# A User Friendly, Type-Safe, Graphical Shell

**Project Specifications Report**

Tristan Allwood, Daniel Burke, Marc Hull, Ekaterina Itskova, and Steve Zymler

January 4, 2005

# Contents

**1 Introduction** **3**
- 1.1 About the Group . . . 3
- 1.2 Relevant links . . . 3

**2 Specifications Introduction** **5**

**3 The GUI Specification** **6**
- 3.1 Minimum Specifications . . . 6
- 3.2 Extended Specifications . . . 6
- 3.3 Optional Specifications . . . 7

**4 The Programs and Types Specification** **8**
- 4.1 Minimum Specifications . . . 8
- 4.2 Extended Specifications . . . 8
- 4.3 Optional Specifications . . . 9

**5 The Framework Specification** **10**
- 5.1 Minimum Specifications . . . 10
- 5.2 Extended Specifications . . . 10
- 5.3 Optional Specifications . . . 10

**6 Architecture Overview** **12**

**7 The GUI Architecture** **13**
- 7.1 Role . . . 13
- 7.2 Functions . . . 13

**8 The Human Interface Abstraction Layer Architecture** **14**
- 8.1 Role . . . 14
- 8.2 Functions . . . 14

**9 The Programs and Types Architecture** **15**
- 9.1 Role . . . 15
- 9.2 Architecture . . . 15
  - 9.2.1 The API interfaces . . . 15

**10 The Framework Architecture** **16**
- 10.1 Role . . . 16
- 10.2 Architecture . . . 16
- 10.3 Functions . . . 16

**11 Proposed Schedule of Work** **17**

# Chapter 1

# Introduction

This is the first of three reports in relation to our 3rd year project *"A User Friendly, Type-Safe, Graphical Shell"*, or *Kevlar*, which is the working name we've given to the system that we will produce.

This report outlines what we believe are the criteria for a successful implementation of our project, the intended architecture of our project and the roles of the members of our group.

The command shell is a powerful tool that allows proficient users to perform complex tasks with minimal coding, however it is very difficult for new users to learn. The core aim of this project is to make an abstraction of a shell that is simple and intuitive, yet retains the power of the original. We aim to achieve this by using high-level constructs such as typed data that can flow over conventional pipes and providing dynamic help on programs and their arguments.

## 1.1 About the Group

The group is made up of five 3rd year computing students which are listed below with relevant details and their roles in the project.

- **Tristan Allwood.** <`toa02@doc.ic.ac.uk`> Tristan is *Group Leader*, and is additionally responsible for the build system of the project as well as the underlying framework.
- **Daniel Burke.** <`deb02@doc.ic.ac.uk`> Daniel is responsible for the core set of programs and types which will be used to demonstrate the project, and the middle layer which interfaces between the underlying framework and the GUI.
- **Marc Hull.** <`mfh02@doc.ic.ac.uk`> Marc is *Group Secretary*, and is additionally responsible for the GUI.
- **Ekaterina Itskova.** <`ei02@doc.ic.ac.uk`> Kate is responsible for the GUI and the interfacing of the GUI with the middle layer.
- **Steve Zymler.** <`sz02@doc.ic.ac.uk`> Steve is responsible for the framework, and its interfacing with the middle layer.

This project is supervised by *Dr Paul Kelly* (<`phjk@doc.ic.ac.uk`>). The group wish to express their thanks for his enthusiasm and interest in the project that has been evident from the start.

## 1.2 Relevant links

- <**`http://www.silicon-fusion.com/kevlar/`**>. Group Discussion and Organisation website, containing meeting notes, discussion forum and hours of working by group memebers.
- <**`http://www.doc.ic.ac.uk/project/2004/362/g04362341M/MavenSite/index.html`**>. Group Implementation website, containing reports, documents, API's and code metrics.

- <**http://www.doc.ic.ac.uk/˜ih/teaching/group-projects/proposals/sue1.html**>. The Part One describes the orignal project proposal.
- <**http://www.doc.ic.ac.uk/˜phjk**>. Dr Paul Kelly's home page.
- <**http://www.doc.ic.ac.uk**>. Department of Computing.
- <**http://www.imperial.ac.uk**>. Imperial College London.

# Chapter 2

# Specifications Introduction

In the following sections we will outline the specifications we believe we need to satisfy for a successful project.

- **Minimum Specifications.** These are requirements which must be completed in order to produce a working, usable product. If these specifications are completed in full, the group should gain a grade B for the project.
- **Extended Specifications.** These are requirements which are not essential to the product, but would improve its overall usability and should be achievable during the period of the project. If these specifications are completed in full along with the Minimum Specifications then the group should gain a grade A for the project.
- **Optional Specifications.** These are requirements that are not essential to the project, and that we estimate may not be completable during the period of the project. However, if completed, these features would improve the project's overall usability, and so we will attempt to implement them near the end of the period once all other specifications are completed.

# Chapter 3

# The GUI Specification

## 3.1 Minimum Specifications

- **Should provide a graphical representation of pipelines that is easy to understand.** New users of the system should be able to easily identify programs and pipes connecting them, as well as arguments and connection nodes.
- **Should allow the user to construct pipelines using the mouse.** The user should be able to drag-and-drop programs then click on connection points to connect them together to form a complete pipeline.
- **Should allow the user to construct pipelines using the keyboard.** To allow experienced users of the Linux shell to easily move to Kevlar, the keyboard should also be supported so that complete pipelines can be constructed without using the mouse.
- **Should gives the user feedback on construction errors.** Users should either be unable to construct incorrect pipelines (i.e connecting inputs to inputs or outputs to outputs).
- **Should restrict the user's ability to enter incorrect argument values.** The input method for arguments should be suitable to the format of argument being entered (i.e checkboxes for boolean values, drop-down lists for discrete selections) in order to limit the scope for error.
- **Should give the user feedback on invalid arguments.** Argument values should be validated and the user alerted if they do not match the specifications given by the program.
- **Should visualise certain pipeline output and errors.** The interface should provide tools for visualising simple program output such as text, and report program errors back to the user in a suitable way.
- **Should allow the user to save and load pipelines.** Once a pipeline has been constructed, an option should exist that allows it to be saved to disk and reloaded at a later date.
- **Should follow a design process that encourages high usability.** A design process should be used that produces prototypes of the interface for users to test and submit feedback on. This feedback should then be used to improve the design of the interface.

## 3.2 Extended Specifications

- **Should allow the user to search for programs to achieve a task.** In order to allow new users to learn which programs can be used to achieve which tasks, the human interface should provide a way that the user can enter keywords related to the task to be performed and search for programs relating to those words.

- **Should provide context-sensitive help based upon the current program, argument or node.** Upon selection of a program, argument or node, the user should be shown context-sensitive help on how to use the selected item in a way that does not interfere with the user's current task.
- **Should allow for construction of basic macros.** The user should be able to select a section of an existing pipeline and group it to form a macro, which is then represented graphically as a single program. This macro can then be saved and used in other programs. The macro should inherit any unconnected input and output nodes from the pipeline it contains and rename those with conflicting names. The macro does not need to inherit any arguments; those set inside the macro are fixed.

## 3.3 Optional Specifications

- **Should allow the user to choose which arguments and nodes should be visible in macros.** When constructing a macro, the user should be able to specify which arguments and nodes should be propagated up to the macro level, allowing them to be altered after the macro has been created.
- **Should give help and suggest corrections for invalid pipelines.** If a pipeline is invalid, the interface should be able to give detailed help explaining why it is invalid and suggest corrections where appropriate. For example, if the user connects together two nodes of incompatible types, the user could be shown a diagram of the type hierarchy with an explanation of why they are incompatible. Additionally, the interface could suggest programs that might convert between the two types.
- **Should give help about invalid argument values.** If an argument value supplied by the user fails validation, the user should be told why the value is incorrect and given suggestions on how to correct the value if appropriate.
- **Should allow macros to be expanded and edited.** Once a macro has been created, the user should be able to expand it back to its original pipeline, edit it and then re-macro it if required. These changes should only apply to the particular instance of the macro being edited and not any unexpanded instances.
- **Should allow macro changes to affect multiple instances.** If a macro is edited, the user should be given the choice of whether or not the changes are made to all other instances of that macro.
- **Should allow the user to construct temporal scripts.** The user should be able to link together multiple pipelines temporally, so that the pipelines are executed sequentially. The execution of a pipeline may also depend on the successful termination of the previous pipeline in the sequence.
- **Should provide a plug-in-able architecture for visualising different types of output.** Aside from displaying standard text output, a framework should exist that allows new visualisations to register themselves with the system and associate themselves as the default handlers for certain types of output. For instance, a plug-in could be added to the system which allows image data output from a pipeline to be visualised.

# Chapter 4

# The Programs and Types Specification

## 4.1 Minimum Specifications

- **Programs can define input and output nodes that are named and typed.** A program must have the ability to connect to other programs in order to exchange data if necessary. This should be done by programs declaring input nodes, from which they can accept data from other programs, and output nodes, from which they can send data to other programs. These nodes should be given descriptive names to help the user, and also be typed so that only nodes with compatible types can be connected together.
- **Programs have arguments, some of which may change its IO function.** A program must have the ability to be customisable through arguments that the user may define. These arguments may also add, remove, rename or change the types of the program's input and output nodes.
- **Programs can validate their arguments before runtime.** A program should be able to check whether the given arguments are valid before the pipeline is executed.
- **A core set of programs should be provided to demonstrate the system.** These programs must provide all the neccessary functionality to demonstrate all the features of the console.
- **Third parties should be able to write programs for the framework.** Programs should be easy to add to the framework. Support should be provided to convert existing programs so that they may be integrated into the framework.
- **Programs should have human-useable names.** Programs in the system should not simply use the names of their Unix-shell counterparts, but instead provide more intuitive names where applicable.
- **Types should be in a type hierarchy.** Programs should be allowed to accept a range of different types by declaring acceptance of a supertype, so that all sub-types will then be accepted.
- **Users should be able to define their own types.** Types should be easily added to the system using a plug-in based architecture.
- **A basic set of types should be provided for demonstration.** The system should be provided with a basic set of types that allow the core features to be demonstrated.

## 4.2 Extended Specifications

- **Programs should export help information.** A program should provide its own help on its function, arguments and the input and output nodes it contains.
- **Programs can accept and produce templated types.** A program should be able to define how the output types relate to the input types without having to specify the exact types used, only bounds upon those types.

## 4.3 Optional Specifications

- **Programs should notify the system when they are waiting for input.** To solve potential deadlock in a pipeline that contains cycles, programs must notify the system when waiting for input, so that the pipeline may be closed if all programs are detected as waiting.

# Chapter 5

# The Framework Specification

## 5.1 Minimum Specifications

- **Can connect programs together using pipes between their input and output nodes.** The Framework should allow programs to be connected together using pipes in order to pass data between them.
- **Pipes should have a flow direction.** A pipe should only be connected from output nodes to input nodes, signifying that data flows out of the output and into the input. Any other combination should not be allowed.
- **Pipes should be type checked.** The Framework should check that connected pipes are type-safe by analysing the type outputted from the output node and the types accepted by the input node.
- **Should run valid pipelines.** The Framework should be able to execute valid pipelines after they have been checked for type-safety.
- **Should handle program crashes.** The whole system should not crash if an individual program crashes during its execution. The Framework should also notify the user of any irregular behaviour of programs during execution time.
- **Should pick up newly registered types and programs from a central repository.** New programs and types should be easily added to the Framework. This should be done by simply allowing users to drop JAR files into certain directories which will be periodically checked and the files dynamically loaded.

## 5.2 Extended Specifications

- **Pipes should have the fuctionality of being Templated.** The Framework should accept pipes that are templated. This means that the type of the pipes will be inferred dynamically by the Framework.

## 5.3 Optional Specifications

- **Should support temporal and conditional temporal options (cf. && and ; in Unix based operating systems).** The Framework should incorporate temporal execution of pipelines. This means that instead of the default behaviour of running all programs at once, a program could for example only execute if the program before it executed without errors.
- **Should support for and while loops.** The Framework should have the functionality of inserting while and for loops in a pipeline.

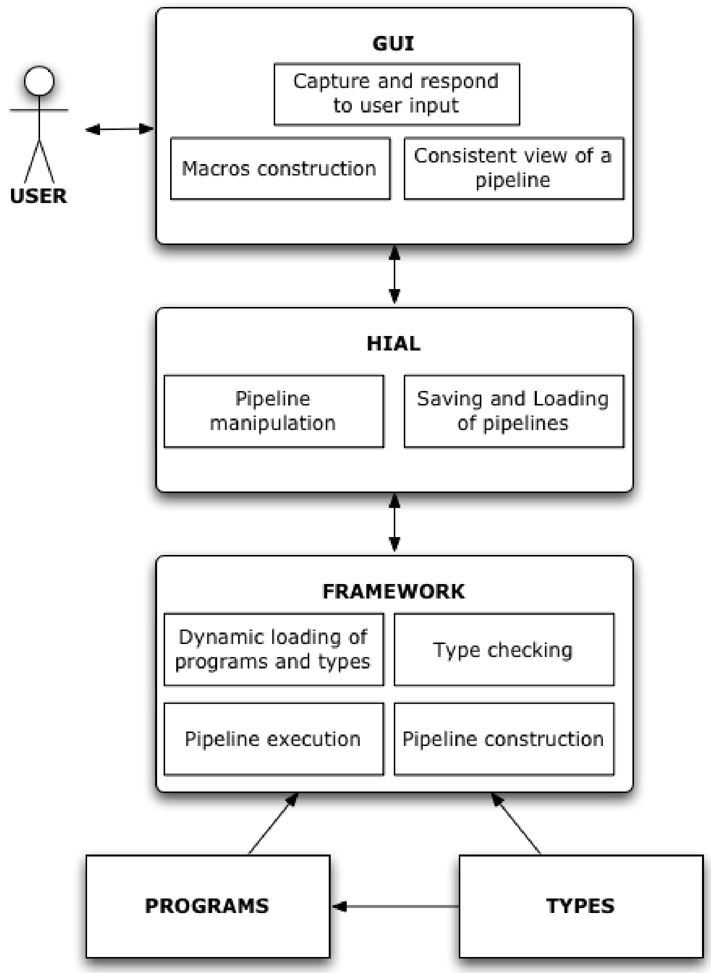
- **Security managers should control what programs do at runtime.** The implementation of a Secutity manager within the Framework should provide a security scheme to filter out unwanted program behaviour.

# Chapter 6

# Architecture Overview

In the follwing sections, we will outline details relating to the architecture of our system. This will include the Role of each section, a comment architectural implementation decisions, and the functions each section should provide.

**Figure 6.1** Architecture Overview. An overview of the architecture of the Kevlar System, showing the roles and communication between the three core layers and the plugginable types and programs.



USER
GUI
Capture and respond to user input
Macros construction
Consistent view of a pipeline
HIAL
Pipeline manipulation
Saving and Loading of pipelines
FRAMEWORK
Dynamic loading of programs and types
Type checking
Pipeline execution
Pipeline construction
PROGRAMS
TYPES

# Chapter 7

# The GUI Architecture

## 7.1 Role

The GUI handles how the user interacts with the system, both in terms of capturing user input and visualising the pipeline on the screen in a human-readable form. It is the responsibility of the GUI to inform the HIAL when the user wants to make changes to the current pipeline. It must also to listen to and respond to changes in the pipeline reported from the HIAL so that the user always sees an accurate representation of the pipeline stored in the Framework.

## 7.2 Functions

- **Conveying the user's intentions to the HIAL.** The GUI acts as a thin layer between the user and the HIAL, providing an understandable way for the user to perform tasks using well-known metaphores such as dragging-and-dropping. It then identifies the user's intention and invokes the relevant methods in the HIAL's interface to indirectly make changes to the Framework's representation of the pipeline.
- **Provide the user with a consistent view of the pipeline.** Since the Framework itself holds an internal representation of the pipeline, the GUI must make sure that the pipeline being displayed to the user semantically matches that of the Framework. It must do this by subscribing to changes to the Framework's model and updating the visual representation when they occur. Additionally, the GUI will maintain type-incorrect pipes which do not exist in the Framework, in order to alert users to their mistakes and suggest corrections.
- **Help to user to find and understand the available programs.** The GUI is responsible for helping users to quickly create pipelines to perform their intended tasks. This involes alterting them of which programs are available and providing facilities to search for programs that perform a specific function. The GUI should also allow for easy access to unintrusive help for each program, and if possible display the available options to the user via an auto-complete system.
- **Provide the user with the ability to contract sub-pipelines into macros.** The GUI should allow the user to create macros from pipelines which reduces on-screen clutter and allows repeated sub-pipelines to be controlled more manageably. It is then up to the GUI to translate these macros into the complete pipeline when calling interface methods in the HIAL.
- **Separate user interaction from the rest of the system.** The GUI should act as a separate layer from the rest of the system, communicating to the other layers using well-defined interfaces. This allows the underlying system to be completely decoupled from way the program captures user input and presents output, allowing other GUI implementations to be added. For instance, it would be possible to write a user-interface using other forms of input such as voice recognition, and also other forms of output, such as rendering the pipelines in 3D.

Chapter 8

# The Human Interface Abstraction Layer Architecture

## 8.1 Role

The HIAL (Human Interface Abstraction Layer) is the mediator between the GUI and the framework components. It will factorize out all of the GUI functionality that would be common to all GUIs that used the framework component. For example, if we were to make a 3D version of the GUI, the HIAL would be completely reusable in that version.

## 8.2 Functions

- **Saving and loading.** The HIAL will have functionality to persist complete and partial pipelines to a file. The GUI component will be able to add additional GUI specific information to the file such as the display co-ordinates of a program. This file can be implemented as an XML file with a flexible schema that permits this extra data to be integrated.
- **Pipeline manipulation.** The HIAL will be capable of altering pipelines to translate between the feature set the GUI will work with and the raw primitives that the framework component will provide according to a set of rules. An example, is connecting an error visualizing application to all outputs with the name 'error'.
- **Searching program information.** The HIAL will contain a library to query the set of installed programs based on a set of keywords. It will also record the frequency with which programs are used. This will enabled the GUI component to prioritize the display of programs in a selection list.
- **Communication mediator.** The HIAL will act as a mediator for messages between the framework and GUI components.

# Chapter 9

# The Programs and Types Architecture

## 9.1 Role

The programs and types are user-side components of the system that the user provides to accomplish their tasks. For demonstration purposes, we will also have a set of programs and types that a user might want to use.

## 9.2 Architecture

Programs and types will take the form of Java jar files. These jar files will reside in the specified directories for programs and types. These can then be loaded by the framework component. If the user wishes to use non java programs, they will use a wrapper tool that generates a java wrapper for the program.

There is only one necessary type in the system, IType. This type can be cloned and is serializable, which is necessary to allow for splitting and joining of pipes. The jar file for each type contains an interface class that must extend one or more interfaces that are either IType, or interfaces that meet this requirement.

### 9.2.1 The API interfaces

- **Information interface.** This interface contains methods to export help information on the program's role and usage. The system can use this information when searching for a program based on keywords.
- **Argument and IO function interfaces.** These interfaces supply the information on the programs' arguments and IO pipes needed by the framework component. This information includes an argument tree that depicts the relationship between arguments. For example, setting the task argument of an ImageEdit program to 'Resize' would cause 'Width' and 'Height' to become available. Each time an argument is changed, the program is queried to find out which of the child arguments should be visible or hidden.

  These interfaces also provide means to validate argument values. The program can start validating it's arguments as soon as they are provided and report back the result.

  Finally, for any given set of arguments, these interfaces are queried to find which IO pipe nodes may be connected, and what type those nodes have. In summary, the entire argument and IO layout of the program is dynamic.
- **Code interface.** This interface allows the program to be run and supplied with all the connecting pipes and the list of arguments provided.

# Chapter 10

# The Framework Architecture

## 10.1 Role

The Framework is the core backend to the project. It is there to hold a representation of the pipelines while they are being constructed, to validate and type check them, and to execute them safely, protecting programs using the Framework library from internal crashes.

## 10.2 Architecture

The Framework is modular, with modules to do each of the core functions. This modularity will be expressed through strong use of interfaces and java packages to keep the framework parts highly cohesive, so for example the TypeChecking module will not need to know about the Dynamic Type Loader module.

## 10.3 Functions

- **Dynamic Program Loading.** Programs in the repository will be loaded dynamically by the Framework at run time. This will enable the user of the system to easily create and add programs that will be used by the Framework as plugins.
- **Dynamic Type Loading.** The Framework will load the types in our system at runtime and dynamically recreate the type tree that will be used by the Framework's type checker. This will enable the user to easily add types that the Framework will use to type check.
- **Pipeline Construction.** The Framework will build up an internal representation of a pipline. This will enable to inform the GUI about possible type errors and prepare the pipeline for runtime execution.
- **Type Checking.** The Framework will perform type checking on connected programs within the pipeline. It will inform the GUI about type errors and IO function changes due to argument changes. It will also support templated type and perform type inferrences on the parameters.
- **Pipeline Execution.** The Framework will be able to execute a type safe pipeline at runtime and inform the GUI of possible runtime execution errors within the programs. The Framework will be robust enough to handle crashes of programs within the executed pipeline and handle them correctly.

# Chapter 11

# Proposed Schedule of Work

Here we outline an initial schedule for our project; this is subject to change as the project evolves.

- **13th October.** Initial allocation to the project.
- **22nd October.** Submission of this report.
- **23rd October.** Internal Milestone One.

  For this milestone, we plan to have the core build system for this project working, a visible GUI, dynamic loading of programs, and passing of this information between the GUI layer and the Framework layer via the HIAL.
- **30th October.** Internal Milestone Two.

  For this milestone we aim to have a small core set of programs and types avaialable, program arguments can change the IO function of a program and this is reflected back in the GUI.
- **6th November.** Internal Milestone Three.

  For this milestone we would like to have program pipeline construction, and be making progress towards executing programs in the Framework.
- **12th November.** Submission of Report Two: Project Design/Implementation.
- **20th November.** Internal Milestone Four.

  The targets for this milestone, and the following two are very dependent on progress made, and will be assesed at that point. By this milestone programs and pipelines should be able to be executed in the system.
- **27th November.** Internal Milestone Five.

  Hopefully all Minimum Specifications are met, and most of the Extended Ones. A fairly usable system should be evident.
- **4th December.** Internal Milestone Six.

  Project feature freeze. Code should now only change for bug-fixing. The shell system should be a usable product.
- **10th December.** Final Release.

  Tested and stable product should have been realised.
- **4th January.** Submission of Final Report.