A User Friendly, Type-Safe, Graphical Shell

Final Project Report

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

January 4, 2005

Contents

Chapter 1

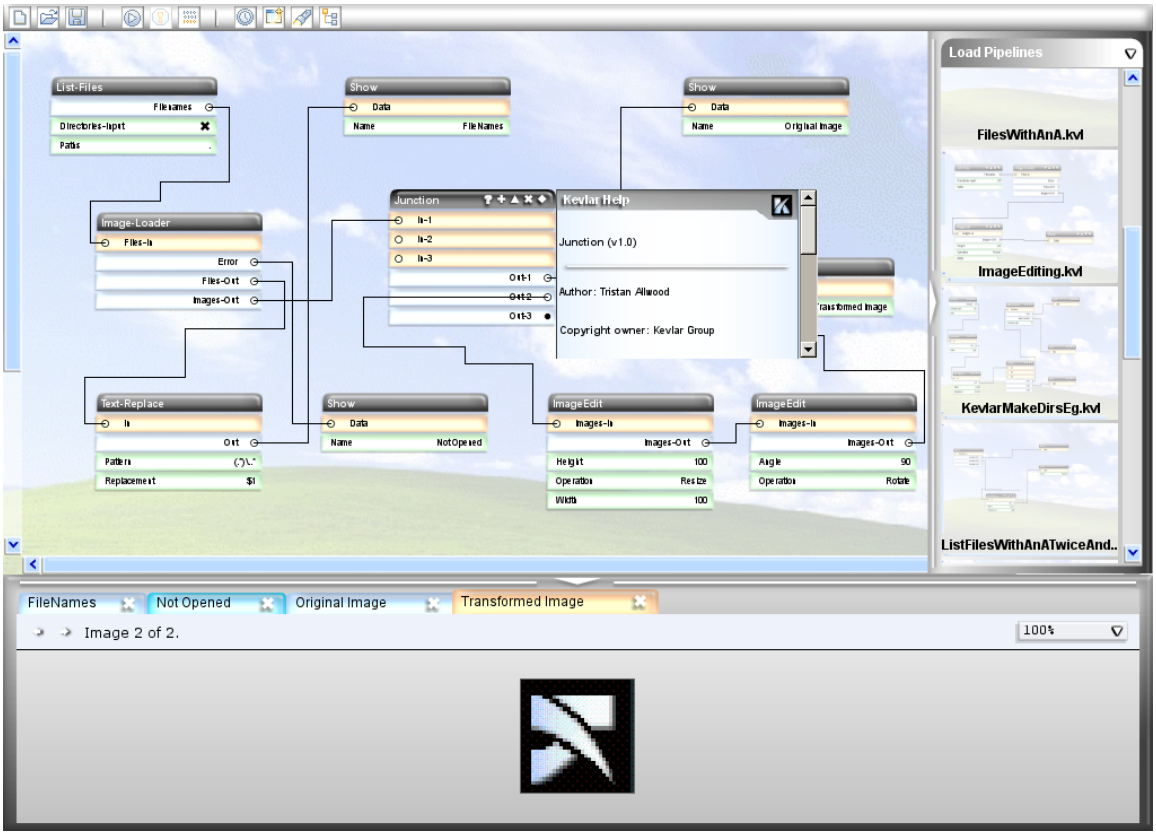
Introduction

This is the third and final report in relation to our 3rd year project, "A User Friendly, Type-Safe, Graphical Shell", or "Kevlar", which is the name given to system we have produced.

1.1 Achievements

We have produced a product that shows it is possible to present a functional way of interacting with the underlying operating system to a novice user which is intuitive, consistent and safe.

Figure 1.1 A Screenshot of the Kevlar System. An example of the Kevlar system performing some rather complex transformations to data. This is the solution to question 6 of our Usability Study (see Section 7.2) which was to display all the images in the current directory, and to transform them (by resizing to 100x100 pixels and rotating to the right 90 degrees), and also to print out their file names, without their extension.



We identified 7 major usability issues with existing shells, that makes learning and using shells hard. Our system, Kevlar, solves these problems.

USABILITY ISSUES SOLVED WITH EXISTING SHELLS

- **Consistent help.** When learning and using any system, having help available that is relevant is often crucial. One of the issues with the existing shells is the lack of a consistent model of help that the shell enforces. Currently there is an expectation that passing arguments such as `-h`, `--help` or `?` to a program on the command line *may* give some summary of help.

  Kevlar does not suffer this problem as there is a well defined API for programs to export help which is simple, and that the shell integrates into its GUI and makes easily accessible to the user.

- **Program discovery.** Traditional command shells provide the ability to run any program, however it is up to the user to manage the programs on their system, and to know which is the best program for any particular job. Migrating to a new shell system is therefore difficult as the user must learn the equivalent commands for their new shell.

  Kevlar categorises programs for the user, and also uses a sophisticated search algorithm to help the user find appropriate tools for a task based upon keywords exported by programs in their help.

- **Argument validation.** Many programs run from the command shell have a very rigid structure of valid arguments (e.g. switches or numeric inputs). Currently it is programs that validate their arguments once they are run. In many cases, there is no reason why the validation could not be done before execution.

  Kevlar uses the GUI widgets to make the process of setting arguments much easier. If an argument can only take one value out of a set of values, this is explicit as the argument is set using a drop down box. This makes setting arguments simpler, and less error-prone. Similarly, there is a checkbox widget for setting boolean arguments.

- **Flexible piping.** One of the most useful paradigms of a command shell is the ability to pipe output from one program into another. In the standard model there are 3 generally supported pipes (in, out and error), although some shells[1] allow redirection of other file descriptors, programs generally do not make the assumption this is available to them from the shell.

  In Kevlar however, programs are free to accept input from many different pipes, or to output to different pipes. The program is free to determine the number of inputs and outputs, and their arguments can be used to make some pipes conditionally available to the user.

- **Typed Pipes.** It is useful to have some notion of the 'type' of information being passed across a pipe. It is common for the shell to allow the user to print to the console (essentially piping to the user) the contents of a binary file, which can in turn change some options on the terminal driving the shell and make it unusable to an untrained user[2].

  Kevlar abandons using untyped binary data/text for output, and moves to a typed system. This makes the shell ideal for working with non-text data. For example, image and sound editing.

- **Spatial representation.** Since the traditional command shell is presented on a text interface, pipelines that are constructed have to be represented in one dimension. If a pipeline construct consists of many redirects in and out (particularly if standard error and out are being sent to different places), this can get very confusing.

  Kevlar introduces a 2nd dimension and allow the user to freely lay out programs to create the most natural view of a pipeline.

- **Program consistency.** Because the actual programs are generally targeted to the underlying operating system, even if the shell itself is portable, the programs run generally are not[3].

  Kevlar has been based upon Java™ technology, which renders it portable across the major Operating Systems (Windows™ and *nix). Programs written for Kevlar in Java are therefore also portable.

[1] For example the gnu bash shell <`http://www.gnu.org/software/bash/`>.

[2] Particularly if the shell is running in a VTxxx terminal and the user echoes the 'Shift Out' command (usually Ctrl-N).

[3] There are efforts to make command shells and their programs cross platform. One favoured by this group is the Cygwin project.<`http://www.cygwin.com/`>

We have also made our system usable to existing shell experts by providing a comprehensive and complete keyboard model for rapid development of pipelines, as well as being friendly to new shell users by providing a complete mouse model.

1.2 What is a command shell?

So far we have explained an overview of what we have achieved during this project. This section will put those achievments into the context of command shells at large.

A *command shell* provides a way for a user to interact with the underlying operating system of a computer. It is called a 'shell' because it provides a level of abstraction away from the operating system, hiding behind the shell's interface. Traditional command shells are implemented in a command line interface, where the user gives a command to execute, with appropriate arguments, and possible redirection of input and output. For example:

```
ls *.html > html_files.txt
```

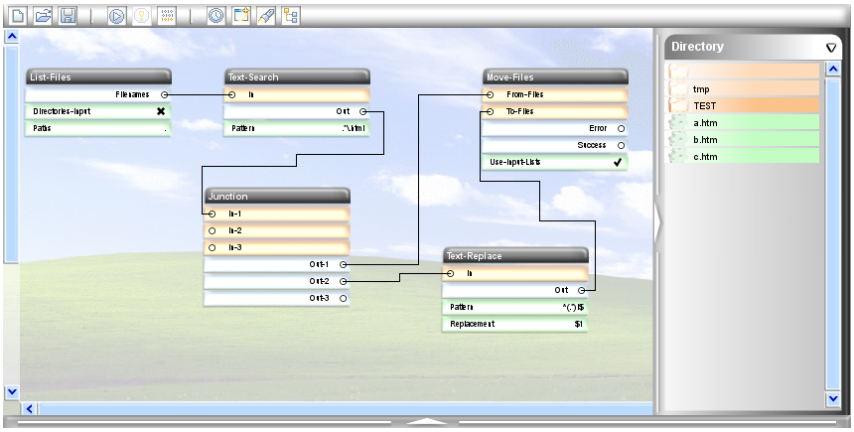
Here **ls** is a program to execute, ***.html** is an argument passed, > tells the shell to redirect the output of the program to a file named **html_files.txt**.

Further, most command shell's support some way of passing the output of one program as input into another. This allows a 'pipeline' to be built up that uses simple programs to execute complicated tasks. For example:

```
ls *.html | sed 's/\(.*\.htm\)l/\1/' | xargs -ri mv '{}'l '{}'
```

In this example, any .html files in the current working directory are being moved to .htm files. For reference, the equivalent pipeline in Kevlar is shown in Figure 1.2.

Figure 1.2 A Pipeline in Kevlar.. This pipeline is the Kevlar equivalent of the shell pipeline **ls *.html** | **sed 's/\(.*\.htm\)l/\1/'** | **xargs -ri mv '{}'l '{}'**. This moves any .html files in the current working directory to .htm files.



For more details about command shells, we refer the reader to [WIKI].

1.3 Research

The following sections are a collection of projects which were designed before the Kevlar project. They are made by other development groups and universities. The reason we explain them is so that we can show that Kevlar borrows ideas from them, tries to extend on some of their features and add new ideas

to them. This research is a necessary step in the design process of Kevlar, because only by looking at advances and problems of previous projects can we progress with our own project.

1.3.1 Visual Shell Projects

This section focusses on the description of projects which are familiar to Kevlar in that they are all shells that try to mimic the classical UNIX command line visually.

1.3.1.1 The PURSUIT project

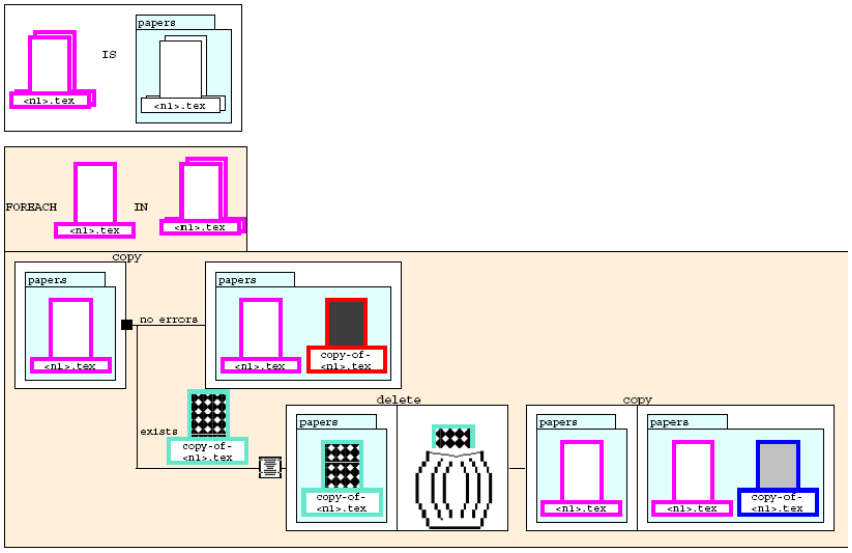
References See [PURSUIT]

Summary The pursuit project is based upon a thesis of Francesmary Modugno (Carnegie Mellon University). PURSUIT is similar to our project on the fact what they both try to resolve the same problem of minimizing mental context switches using the classical Command Line interpreters.

Although PURSUIT also uses a workflow visualization it has a rather different way of representing data. In fact in the PURSUIT system there are only four different types; file, set of files, folder and set of folders, so the PURSUIT system does have typed input output but not in the same degree as Kevlar which offers unlimited types through extensibility. The user can interact with the system in a form of very simple language (pseudo code like). The system will show Pre and Post states to explain what every execution node does. For example, to visualize the copy command that makes a copy of test.tex with name copy-of-test.tex in a folder (directory) called paper, the system will show a node split in two, with on one side a folder called paper which contains test.tex and on the other side (post), a folder called paper which contains test.tex as well as copy-of-test.tex. The connections between node do not represent typed pipes as our Shell will but represent control branches such as if exist copy-of-file.txt => delete. The PURSUIT system is clever enough to automatically detect if the user wants to define a loop over every file in a folder for example.

Conclusion of the thesis was also that the more intuitive way of representing command processes did in fact boost the users handling of the shell. To get a better idea what the description above means it is worthwhile to take a look at the paper with the link pasted above.

Figure 1.3 Snapshot of the PURSUIT System. The Pursuit script that copies each *.tex file in the papers folder. If the copy operation fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the copy operation. Users can see the other possible outcomes of the copy operation by clicking on the conditional marker.



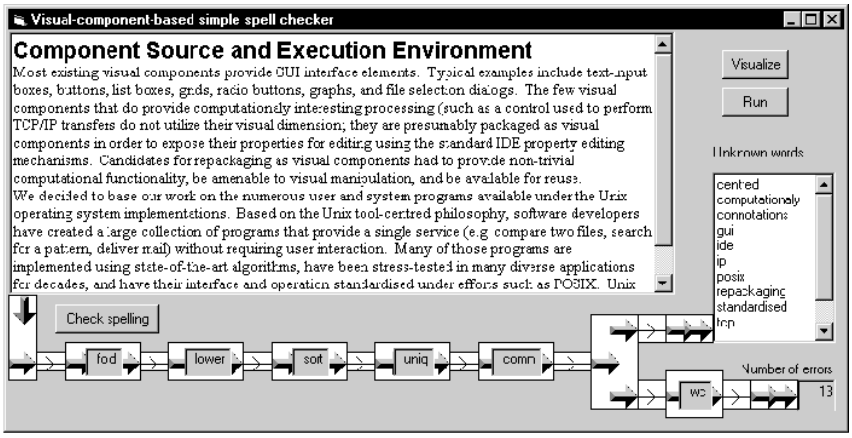
1.3.1.2 The VUFC project

References See [VUFC]

Summary The visual Unix-filter components (VUFC) project contains a lot of the functionality of the Kevlar project, except for the feature of typed pipes and therefore lacks a lot of the context sensitive help and type safety. It successfully managed to encapsulate Unix filter apps (like sort, wc, etc.) within COM components and display them on canvas connected by real UNIX pipes via ActiveX (in VB and ATL). The arguments needed by the encapsulated unix filter are described in their own language which is compiled into VB source so that the GUI can give information about them at build time. The automatic encapsulation of unix commands within their system is a benefit Kevlar does not have however - in Kevlar programs need to be made compatible for the Kevlar system. However, previousely written Java programs aren't difficult to integrate into Kevlar.

Interestingly they explained that typed pipes might be an ideal addition to their project. Information is given on the link cited, although no source or executables have been found showing the working process.

Figure 1.4 Snapshot of the VUFC System. A GUI-based spelling checker built using VUFC.



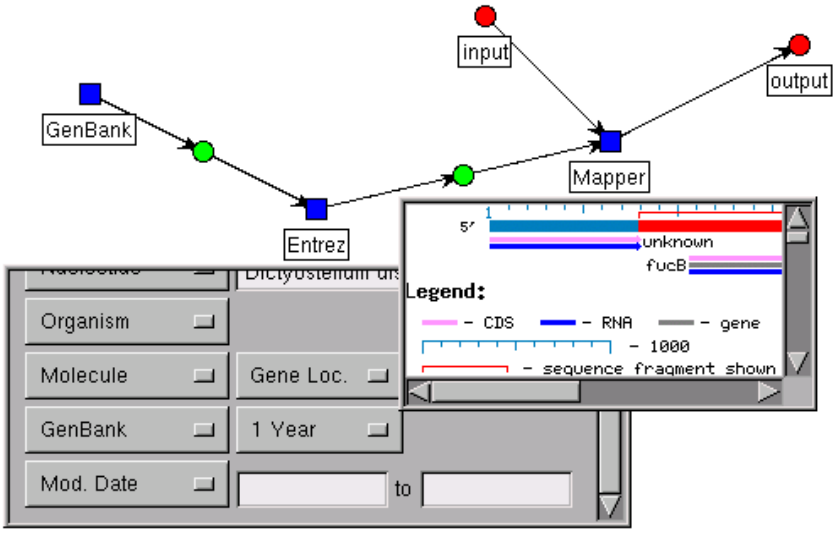
1.3.1.3 The Piper Project

References See [PIPER]

Summary Piper is a network-distributed system of clients and servers for data processing. Client types include programs that process data (perform analysis, translation and visualization). (These are not part of the system but come as extensions, making Piper independent of data-type and thus general-purpose.) Other clients include control structure (e.g., if and while) and user interface (UI) components. In the Pied UI, clients can be represented as the nodes of a "Work Flow Diagram" (WFD), joined by lines that depict network connections. The Pied/Piper system therefore provides a workspace for connecting and combining nodes, allowing Piper to function as a graphical scripting language. And, the network-distributed nature of Piper permits the user to deal with large data sets in a unique way: UI components will reside on a local workstation while compute-intensive, data-processing nodes execute remotely on high-performance computers. Linking nodes across the Internet can also be used to form world-wide collaboratives and provide access to an infinitely extensible set of tools for the user.

As we can see from the description above Piper tries to establish the same idea as Kevlar - A graphical shell where nodes are process elements and we can link them to pipe them together so that we can stream over I/O results betwean nodes. There are some features Piper has that Kevlar hasn't; Piper doesn't require that the process elements are local, they can be anywhere on the web and Piper can reuse native UNIX commands without rewriting. Piper's description didn't mention typed pipes, so this is a great benifit Kevlar has over Piper. We also couldn't find any executables online, the only link we saw to a binary was not operational.

Figure 1.5 Snapshot of the Piper System. A Piper network connecting Bioinformatics Databases and running applications spread out over multiple sites.



1.3.1.4 Research Conlusions

In every one of the covered projects the conclusion was that a visual representation of a process flow did resolve usability issues with the standard Command Line.

Although the covered projects in this document contain some of the functionality that overlaps with the Kevlar project none of them contain the typed pipes feature that Kevlar has. And since the type system within Kevlar offers the user a lot more contextual help we believe that the Kevlar system boosts usability even more and thus fulfill the goal of minimizing mental context switches on user level even better.

1.3.2 Workflow Projects

The following projects are all familiar to Kevlar in that they all work around the creation of a workflow. These projects do not necesserally try to establish the same goals as Kevlar, but they all use a workflow to describe their work process. Note that all of the above described projects also create workflows.

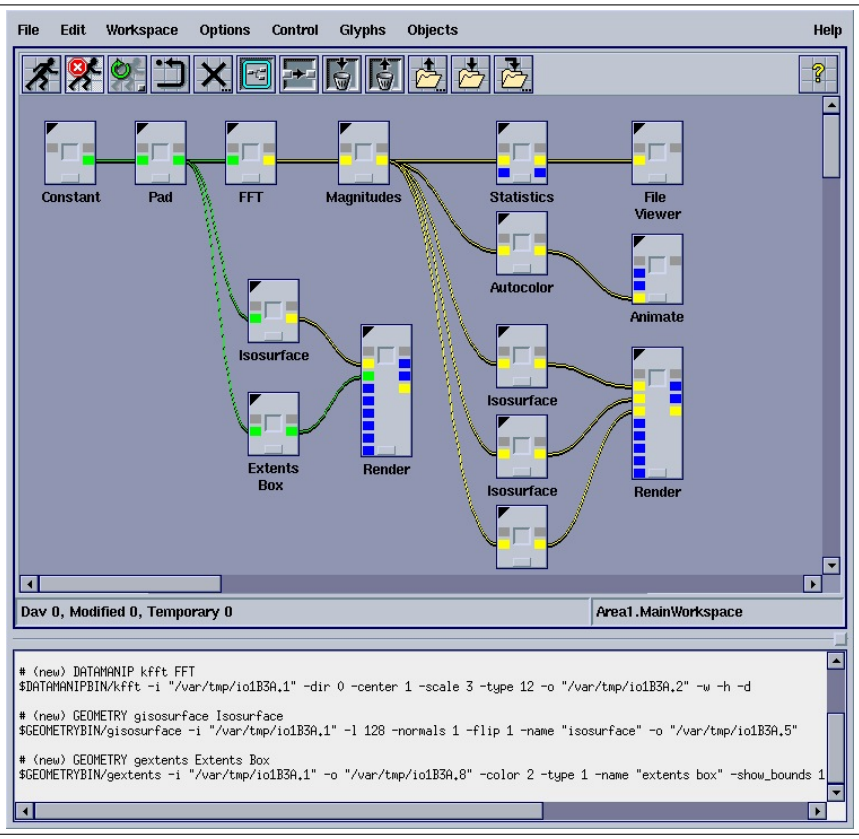
1.3.2.1 The VisiQuest Visual Programming Environment

References See [VISIQ]

Summary With VisiQuest, visual programs are created by placing glyphs (the rectangular icons in the image below) in the VisiQuest workspace. Glyphs represent operators, which are simply stand-alone programs written in C, C++, Java, or a scripting language. Each operator performs on an input image or dataset, producing an output image or dataset. Connections that represent data flow between the glyphs are created by clicking the mouse. Advanced programming language constructs such as loops, procedures, and control structures (i.e., if/else constructs) complete the visual programming capabilities. You may write visual programs that take advantage of the functionality offered by more than 300 operators included with VisiQuest, or you can use the VisiQuest software development environment to develop new operators of your own. Once a visual program has been created with VisiQuest, it may be compiled into a standalone program that can be executed without VisiQuest.

The workflow VisiQuest uses is quite similar to the one Kevlar uses. VisiQuest also offers the compilation of a pipeline into an executable, in Kevlar pipelines need to be executed within the System. VisiQuest's goal is more of a programming environment than that of a general Shell replacement like Kevlar is and is focused around image processing.

Figure 1.6 Snapshot of the VisiQuest System. A visual program used to perform rendering of 3D geometry data in VisiQuest.



1.3.2.2 The MEProf Profiling Framework

References See [HBK04]

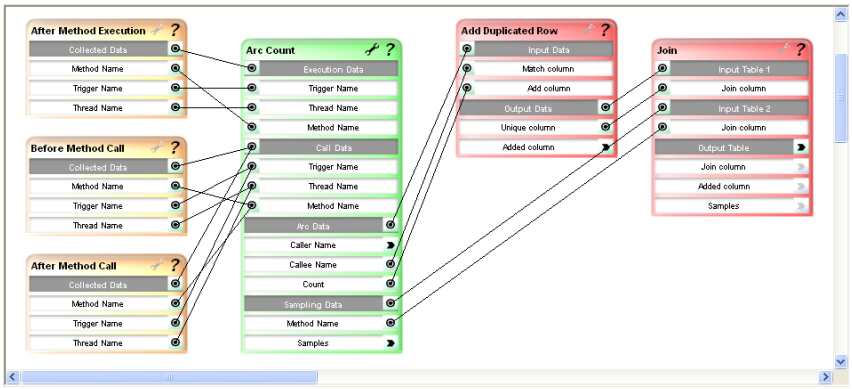
Summary MEProf is a profiling framework plug-in for the Eclipse IDE which allows the user to specify the type of profiling to be performed by building a visual workflow diagram. The plug-in contains a set of dynamically loaded profiling modules which perform a specific part of the profiling pipeline, such as capturing program events, gathering program information, processing information and visualising. Each module declares a set of input and output connection points that declare the names and types of the data they receive and produce. The user builds a profiling pipeline by adding modules from those available and connecting them togther by clicking on their input and output nodes. The framework then analyses the connections to make sure that the types of data passed between modules is compatible before allowing the pipeline to be deployed. When executed, part of the pipeline is woven into the program being profiled while the other half runs in a server, which receives profiling data and displays visualisations.

Although MEProf and Kevlar use a similar method for representing pipelines and both contain type checking, the two programs aim to achieve quite different things. Whereas MEProf simply uses pipeline visualisation as an interface for the user to declare the kind of profiling they wish to perform, Kevlar places much more emphasis on allowing the user to search for programs, executing programs safely and constructing pipelines quickly using the keyboard. MEProf also doesn't put any security restrictions upon its modules, since it trusts that modules are correctly written and will not crash the system. However, Kevlar cannot afford to make this assumption, so makes sure that programs are executed as separate processes so that the system can recover gracefully from crashes. Kevlar also allows users to specify more complex pipelines containing cycles, which are not allowed by MEProf due to the restriction on which modules can be connected to which other modules.

MEProf was written during Summer 2004 as a research project by Marc Hull, who is also one of the Kevlar team members. This means that the group is well aware of the shortcomings of MEProf's inter-

face and has been able to address them when designing Kevlar.

Figure 1.7 Snapshot of the MEProf interface. A profiling pipeline created in the MEProf plug-in for Eclipse.



1.4 The Group

The results of this project represent approximately 3 months work by five 3rd year MEng Computing students.

- **Tristan Allwood (Group Leader).** <toa02@doc.ic.ac.uk>
- **Daniel Burke.** <deb02@doc.ic.ac.uk>
- **Marc Hull (Group Secretary).** <mfh02@doc.ic.ac.uk>
- **Ekaterina Itskova.** <ei02@doc.ic.ac.uk>
- **Steve Zymler.** <sz02@doc.ic.ac.uk>

This project has been 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.

1.5 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/project/2004/362/g04362341M/study/**>. Group Usability Study Website.
- <**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

Design Overview

2.1 Technologies And Tools

This section will outline the primary technologies and tools we used during the development of this project.

2.1.1 Programming Language

The primary language chosen for the development of this project is Java (J2SE 5.0)[1]. This is because it was a language we knew and are confident with, however the choice to adopt J2SE 5.0 also meant we had the opportunity to learn the new Java features (particularly generics and enumerations).

Additionally, Java has a large API which has many features that where useful for the project. An overview of some of these is below, along with a few extensions to the API that have been used.

THE JAVA J2SE 5.0 API (AND EXTENSIONS) AND THEIR USAGE IN THE PROJECT

- **Reflection.** For dynamic loading of types, and to validate that programs extend a certain class.
- **XML DOM.** For XML parsing and validation. This is useful for parsing XML files that describe programs and for the saving and loading of pipelines.
- **Serialization.** User-defined programs are executed in separate processes. Serialization is used to allow first-class objects to be communicated to these processess across standard IO pipes.
- **Processes, IO, Threads.** For the actual location of loadable programs, the execution of programs and communication with them, Java's IO libraries have simplified many of the already complicated procedures. Also Java's ability to create daemonized threads, pool them and perform inter-thread-communication via the new concurrent libraries has proved valuable.
- **JUnit[2].** An extension tool for Java that allows flexible unit testing of code. This has been adopted by the group as a way of checking that code, contracts and algorithms work correctly without having to wait for other users to find bugs. The unit testing of the code will be covered in more detail below.
- **SWT[3].** The Eclipse Project's Standard Widget Toolkit provides a way of creating graphical-user-interfaces which exploits native platform optimisations. SWT has been adopted as the base for the GUI, as its programming model is more flexible and powerful than the standard Java AWT/Swing.

A final reason for the choice of development using Java is that it is platform independent. Although the choice of using SWT limits the project to only a Windows, Linux or Mac platform for deployment, the GUI has been designed to be a modular component, so creating a separate GUI that uses the rest of the

[1] <http://www.java.sun.com/j2se/1.5.0/download.jsp>
[2] <http://junit.sourceforge.net>
[3] <http://www.eclipse.org/swt>

system should not be impossible if it is so desired. Windows™ and Linux where the major targets for the project as those are the operating systems the group members (and DoC) primarily support.

2.1.2 Project Code

For the actual development of the code, it was decided to use Eclipse[4]. Since this is a Java based system (which is based upon SWT), it is available for both Linux and Windows, and the interface on both systems is consistent. Eclipse also has support for the Java J2SE 5.0 features.

Because work is being done by group members both at home and in DoC, the source for the project should be easily accessible and updatable via a content management system. The choice for CMS has been CVS[5] (over ssh, using the provided group project space as the repository). Eclipse also features first class support for CVS which was a factor in the decision to use it.

Eclipse also has the feature that build and run targets can be shared by the group. The consequence of this is that only one person needs to set-up the running of the main program / unit tests / building of jars, and then on the next CVS update all other members of the group can see this target under an accessible menu. This is elaborated upon further below.

2.1.3 The Build System

One of the requirements for this project is a plug-in-able archticture of Types and user-defined Programs. As will be described in the 'Programs and Types' section (see Chapter 3.1, "Program and Type Interfaces"), groups of Programs / Types need to be put together in jar files containing manifest information, and descriptors (xml files).

Eclipse provides support for the compilation of all source code into class files, however to release an actual 'product' there needs to be a way of building a jar file for the main program, with satellite jars for each of the Program and Type groups. Also, to actually run the main program (even for testing) the Program and Type jars must be generated, as the loading mechanism requires them to be in the format summarised above.

The group wanted to minimize the amount of time members spend doing jobs other than working on their code, so it was decided to create a flexible, platform independent way of building the main and satellite jars, and to perform any other useful tasks the group wanted (e.g. generation of group documentation). To achieve this, the Ant[6] build tool, and Eclipse's configurable Run and Build targets are used so that any group member can generate all the jar files with no more than two mouse clicks.

The way in which the satellite jar files are generated is outlined below:

1. **Normal Source Code Compilation.** The Ant build script checks that all the source files have been compiled. As Eclipse is setup to continuously build the source for the project, very few files are ever compiled in this step.

2. **Locating Types and Programs.** A custom written Java program that is held in the project source tree is invoked by the main Ant build script. This program iterates through the predefined directories in the source tree where Programs and Types that should be dynamically loaded are stored.

   Each sub-directory in these locations corresponds to a group of Programs or Types. Each class file in these sub-directories is then loaded using reflection and analysed to see if is a valid Program or Type. If it is, this information is noted.

   The program finally outputs an XML description of the groups of Programs and Types it has found.

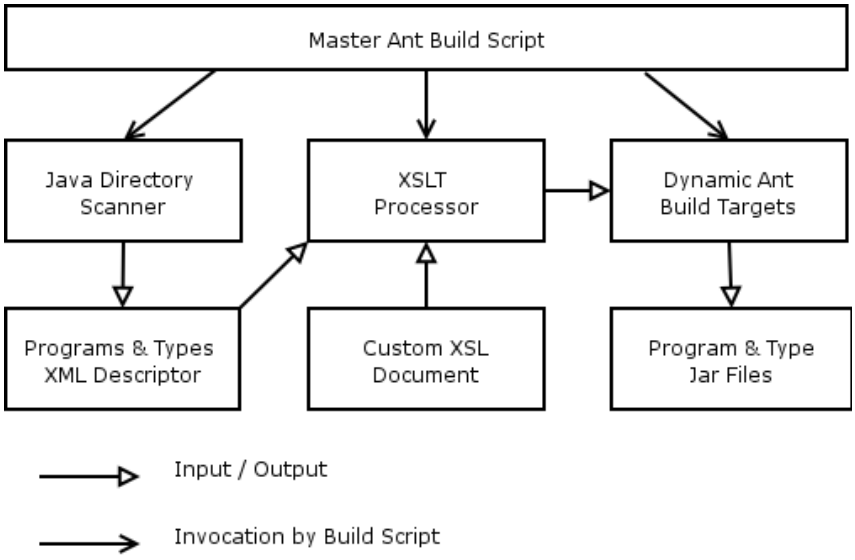
3. **Dynamic Ant File Generation[7].** The main Ant build script then invokes an XSL transform upon the XML descriptor using a custom written XSL document. This creates a new XML document

[4] <http://www.eclipse.org>
[5] <http://www.cvshome.org>
[6] <http://ant.apache.org>
[7] As explained in the Apache Ant document 'Ant in Anger' (<http://www.ant.apache.org/ant_in_anger>) the use of this technique places our group "on the bleeding edge of technology".

that represents an Ant build script that has targets to create each of the satellite jar files, with the correct dependencies and correct manifest information.

4. **Satellite Jar Generation.** Finally, the main Ant build script calls a target on the newly generated build script to produce all the jar files and place them in the correct place for the defined run targets to use them.

Figure 2.1 Program and Type satellite jar generation by the Build System



2.1.4 Unit Testing

As mentioned above, the JUnit framework has been employed in this project to allow for quick and easy testing of code. JUnit has first class support in Eclipse, so its integration and use has been simple.

The principal motive for using JUnit has been to facilitate rapid development and checking of the underlying components of the project (type-checking, dynamic loading and xml parsing to name a few), before they are needed. This helps ensure that when integration does occur it is smooth as most code will have been tested before being depended upon.

As the design is very modular and component-based, the use of unit testing is facilitated as each component can (and in most cases has) been tested as it has been written. Also, since JUnit allows the unit tests to be quickly re-run (and an Eclipse run target shared by all users has been set up to facilitate this), the group is able to check that changed parts of the code do not have major consiquences on others.

2.1.5 Maven

As mentioned in the previous report, the group has a *Group Implementation Website* located at <`http://www.doc.ic.ac.uk/project/2004/362/g04362341M/MavenSite/index.html`>. This website is automatically generated for the group by the apache tool Maven[8].

Although not an integral part of the project, this website has proved valuable for the group as it provides summaries of unit test results, the CVS changelog, interesting statistics on file changes and source code metrics, as-well as all of the group JavaDoc and a central place to keep all of the group reports.

[8] <`http://maven.apache.org`>

2.2 Design Challenges

During the planning phase for Kevlar, we identified some key challenges and risks that would need to be solved or mitigated as part of the project. A summary of these is below:

KEY CHALLENGES AND RISKS FOUND DURING THE DESIGN OF KEVLAR

- **Type Checking.** One of the features our project aimed to present was a way of type-checking the input and output pipes of programs, and disallow type-unsafe pipelines from being constructed. We additionally wanted to allow templated types, and discussed the possibility of allowing multiple inheritance in the type tree.

  The mitigation of this risk was early on in the project, when much work was spent on planning and testing the solution algorithm to this problem. It was decided that multiple inheritance would be beyond the scope of this project, and is not necessary for most user needs.

  The implementation of the type checking algorithm can be found in Section 4.4.

- **Execution.** To have a complete shell, we needed to be able to execute programs from within it. Originally we planned to execute programs in Kevlar's JVM, but research into this area (see Section 3.3) showed that this could not be done with any security, so an alternative way of forking processes and managing them was needed.

  It was known that this problem was solvable (and there was at least one group member with experience of dealing with creating Java processes from within Java and performing IPC with streams). The actual mitigation of this risk came mid-way into the project, (partly because the underlying frameworks that it required were not present until that time) with a prototypical implementation to allow the rest of the project to carry on, before a more thought out solution.

  The implementation of the core of the execution system can be found in Section 4.6.

- **GUI Widget Set.** Our project was aimed around providing a user-experience that was tailored to building pipelines. As part of this, we realised it would be necessary to use our own custom widget set, as standard ones are not designed for this type of application.

  We realised the need for custom widgets early on in our design, and planned to create widgets for representing programs, pipes, nodes and arguments. Although we originally intended to use existing SWT widgets for standard components such as text boxes and scrollbars, this meant that certain optimisations could not be applied to our drawing system. In the end, we decided that the extra performance provided by the optimisations was more important, and wrote our own set of standard widgets too. The design and implementation of these widgets can be found in Section 6.2.

- **Fast Keyboard Model.** In the specifications for this project, we wanted to allow users of existing shells the ability to transfer to ours. The primary method of using existing shells is the keyboard, and to ease this transition (and to allow power users speed when working on our shell) we wanted to provide a complete and fast keyboard model for all actions possible in the shell.

  The ability to construct pipelines using the keyboard was stated as a requirement of our project, but early on we realised the need to make common actions fast. We designed the auto-complete section to mirror the tab-completion feature of most standard consoles and implemented a number of methods for quickly navigating between programs and selecting nodes. We tested a number of key combinations and compared them to the number of keypresses required to create semantically equivalent pipelines in the Linux console. The design and implementation of the keyboard model is described in Section 6.4.

- **Layout Algorithm.** It became clear during our use case analysis, that complicated pipelines could become very hard to visualise, and that support at some level had to be given to routing pipes and automatically laying out programs. This became even more of an issue as users should be able to interact using only the keyboard if they wish.

  Since automatic layouts was not mentioned in our initial requirements document, we considered the interfaces required for it early on but did not plan implementation until quite late on in the project. Since the interfaces we well defined, we were able to work on this section completely separately from the rest of the graphical interface, and simply replace the default test layout with

the new layout algorithm to integrate it into the rest of the system. The layout algorithm used is explained in Section 6.3.

- **Macro pipelines.** During the design and development of use-cases for our project, it was recognised that a way of grouping common pipe-lines into 'macros' to be reused would be a useful feature. It was also recognised that this would require architecture frameworks in place to support it, and there were also discussions as to whether this abstraction should something that is throughout the entire project, or just in the GUI.

  Initially we spent a long time disucssing what we wanted macros to achieve and whether we could link them together temporally to create Bash-style scripts. This highlighted a number of difficult problems, which forced us to move the majority of the macro requirements into the optional specifications. Since the only compulsory requirement for the project was to implement basic macros that could not be temporally linked and did not allow the user to choose propagated arguments, we decided to implement them solely in the GUI and translate any operations upon them into operations upon their component programs before contacting the HIAL. However, the possibility of expanding macros to support temporal linking was considered in the design of the framework in case we wanted to add it at a later stage. The implementation of the compulsory macro requirement is covered in Section 6.7.

2.3 System Design

The overall design for the Kevlar system was strongly based upon our requirements.

We needed a plug-in based architecture for Programs and Types, and ways of loading these plug-ins in. We also needed ways to construct pipelines and type check then. These are very ”low level” requirements, completely abstracted away from users or graphics.

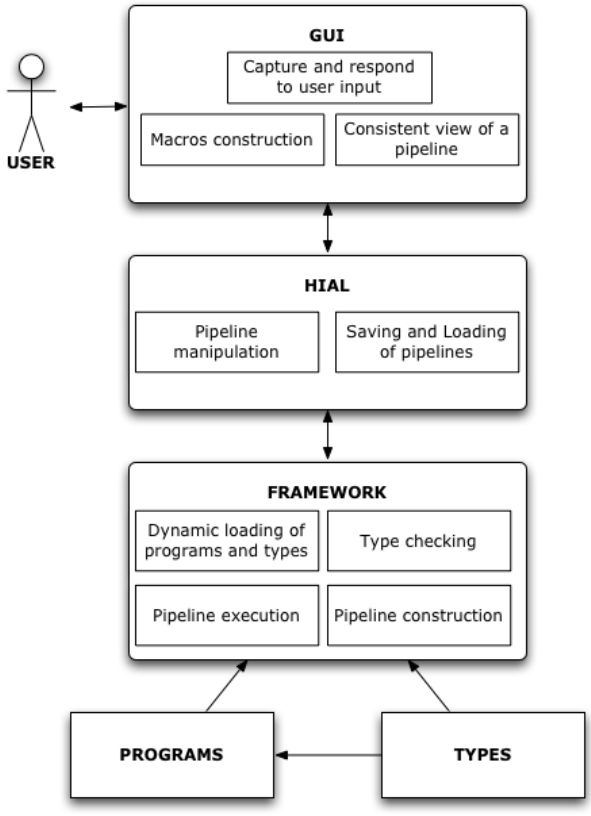
Additionally, we had to visualise and capture interactions with pipelines, but this was not concerned with how programs are loaded, or how type-checking is performed. These are very ”high level” requirements.

We decided to have two separate components, one for each extreme. The ”high level” became the “GUI” component, and the ”low level” became the “Framework”.

However there was still a 'gulf' of functionality missing between these two components. To bridge this, we created a mediation layer, known as the “Human Interaction Abstraction Layer” (HIAL). Its role was to remove common functionality that any GUI could require, and to provide primitives for manipulating pipelines made in the framework.

The following chapters will cover the different components, and the main parts of their design and implementation.

Figure 2.2 Architecture Overview. An overview of the architecture of the Kevlar System, showing the roles and communication between the three core layers and the plug-in-able types and programs.



Chapter 3

Program and Type Interfaces

As part of the requirements for this project, we needed to create an API that external developers could use to write programs and types for Kevlar. This chapter discusses the different parts of the design and implementation of these interfaces, and how they reflect back upon the end user of Kevlar.

3.1 Overview of programs

3.1.1 User's view of programs

From the user's point of view, a program is a functional unit that can be put into a pipeline and executed.

Programs take arguments, and have a set of input and output nodes. Arguments are name-value pairs that allow the program to be configured. Input and output nodes are connection points for pipes for linking programs together in a pipeline.

A type is assigned to each input and output node of each program. To the user, these are the possible types of information that can flow between programs over pipes.

3.1.2 Developer's view of programs

When a program is run and it can take input from its input nodes, process the incoming data and then output new data on its output nodes. The arguments will be used to control what input and output nodes are available and how the program processes the input it receives.

In addition to these features, programs also provide a map converting program exit codes into messages for the user, and a set of basic information about the program such as its name and version. Figure 3.1 details the features of programs that a developer uses.

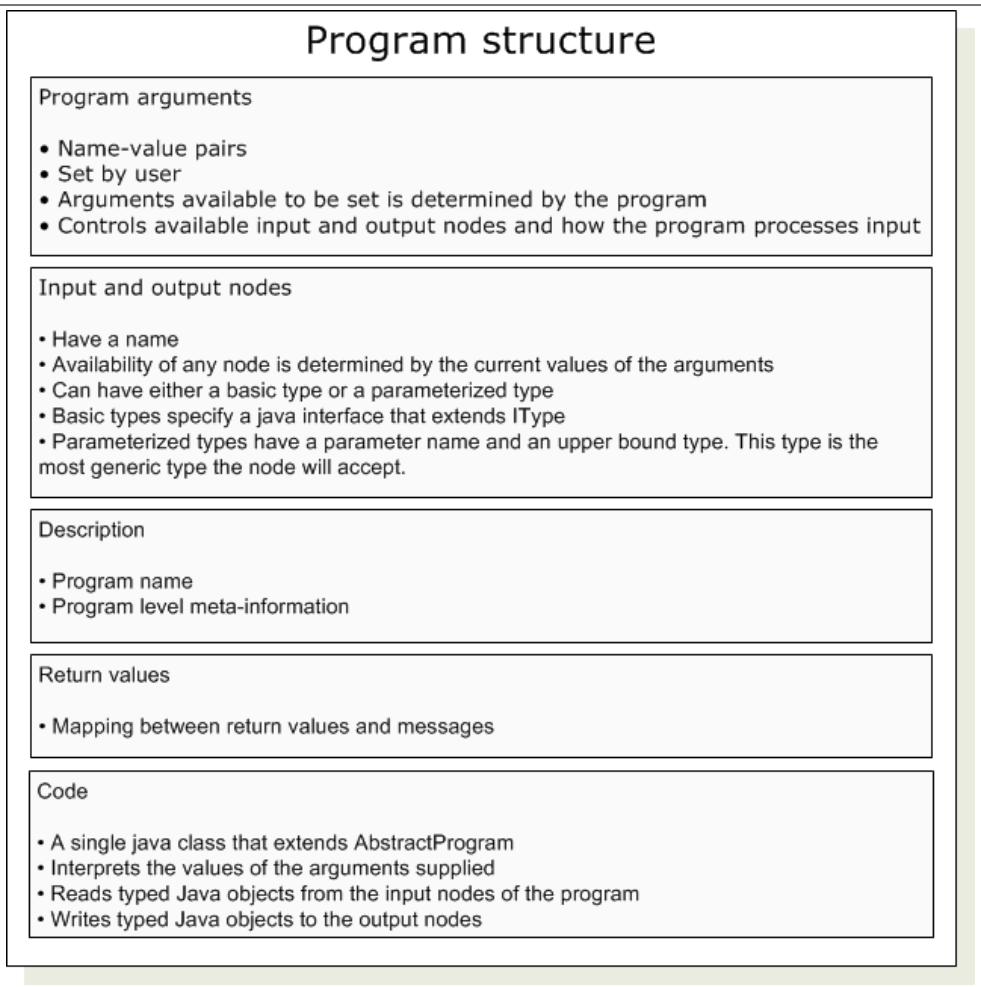
3.2 Program meta-information

3.2.1 Meta-information design

To meet the goal of creating a user friendly shell, a rich set of meta information needs to be available for each program. Early on was identified particular meta information which would be necessary or would be useful to the shell.

- **Basic identification information.** The program should have two names. One is the package identifier which can be assumed to be unique due to the official derivation rules for package names. The other name would be a friendly name that meets the specification for having names that are meaningful to the user.

Figure 3.1 Overview of the structure of a program



To allow different versions of programs to be distinguished from one another, version information should be available for all programs. This would standardize the method for getting the version information from a program.

The user should be able to discover where a program came from. Therefore the program needs to export a form of contact information; A website URL, for example.

- **Arguments.** Traditional shells allow programs to have arbitrary argument models and, as a result, there are many different ways to specify arguments for UNIX programs.

  In order to have a more usable approach, a standard model for presenting arguments needs to be provided. Originally a dynamic model was planned in which only arguments and input/output nodes that were relevant to the current program configuration would be shown. Several design possibilities were considered.

  – **Arbitrary argument specification.** The original design was to allow programs to specify currently available arguments and input/output nodes which could change arbitrarily with the input pipes connected and the current arguments set by the user.

    This design was rejected since there would be no consistent model for the user; small changes could cause unexpected effects to the pipeline which would not be user friendly.

  – **Argument tree model.** The preferred design would restrict the ways in which program arguments and the set of available input/output nodes could be specified by the programs.

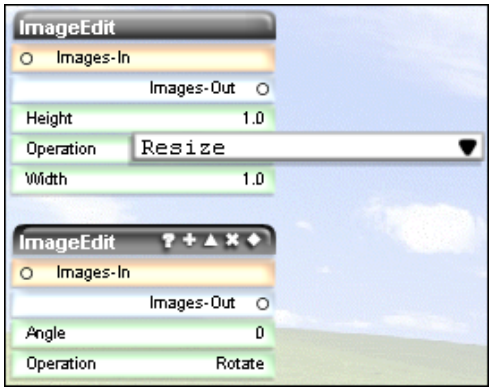
    The number of argument types was restricted to four types. These types are; singleline text, multiline text, boolean and a discrete selection set type. These could be represented in the GUI as textlines, textboxes, checkboxes and combination boxes respectively. Having fixed

types ensures a consistent method for setting arguments between programs.

Programs must specify an argument tree. There is a set of root arguments which is always available for the user to set. When a boolean or set argument has its value altered, the program may specify which arguments will become available as a result. This provides a predictable model for arguments, since the availability of any non-root argument is completely dependant on the discrete value set for exactly one other argument.

As an example, in a program for manipulating an image, the 'Operation' argument would always be available for the user to set. The 'Operation' argument would be of the 'set' type, which means it might be represented in the GUI as a dropdown menu. If the user chooses 'Resize' for the operation, the 'Width' and 'Height' arguments become available for the user to set. Figure 3.2 shows this example.

Figure 3.2 Example of argument trees. When the 'Operation' argument is set to 'Resize', the 'Width' and 'Height' arguments are availalble. When 'Rotate' is set, the 'Angle' argument is available.



- **Program input and output.** Traditional shells mostly provide one fixed input and two fixed outputs for a program. (Standard in, out and error.) Since Kevlar is breaking away from this model, the program meta data needs to reflect this by specifying the input and output possibilities.

  In order to give the best user model, the available input/output nodes of a program and their corresponding types needs to be determined exclusively by the discrete valued arguments of a program. Programs will, therefore, be able to export mappings from argument values to input/output node sets. Figure 3.3 shows an example of this.

- **Validation.** Arguments should be able to be validated before the program is run, the program should be able to specify validation rules for arguments so that the user can be alerted of mistakes immediately, instead of when they run the pipeline. Figure 3.4 shows this validation meta-information in use.

- **Context sensitive help.** To meet the specification of having context sensitive help, the programs need to be able to export help for each of their individual components, which includes their arguments, and each input and output node. Figure 3.5 shows the help meta-infromation in use.

- **Return value to message mappings.** Existing programs return a numeric exit code. In this system, these can be mapped to a predefined set of return value messages.

  These messages could then be presented to the user as feedback on the completed execution of a program. Unfortunatly, due to time constraints, this feedback on the result of a program's execution is not represented in the GUI. However, this feature would be useful to tackle the complaint regarding lack of feedback from programs in the command line console.

3.2.2 An extract from an example XML meta-information file for a program

```
<!-- XML meta data file for a String searching program like grep -->
<program>
```

Figure 3.3 Example of argument trees. The 'Move-Files' program has an 'Use-Input-Lists' argument. When this is off, a 'From-File' argument and 'To-File' argument are available. When this is set on, the arguments dissappear and a 'From-Files' input node and 'To-Files' input node becomes availalble. This mapping is set in the meta-information for the program.

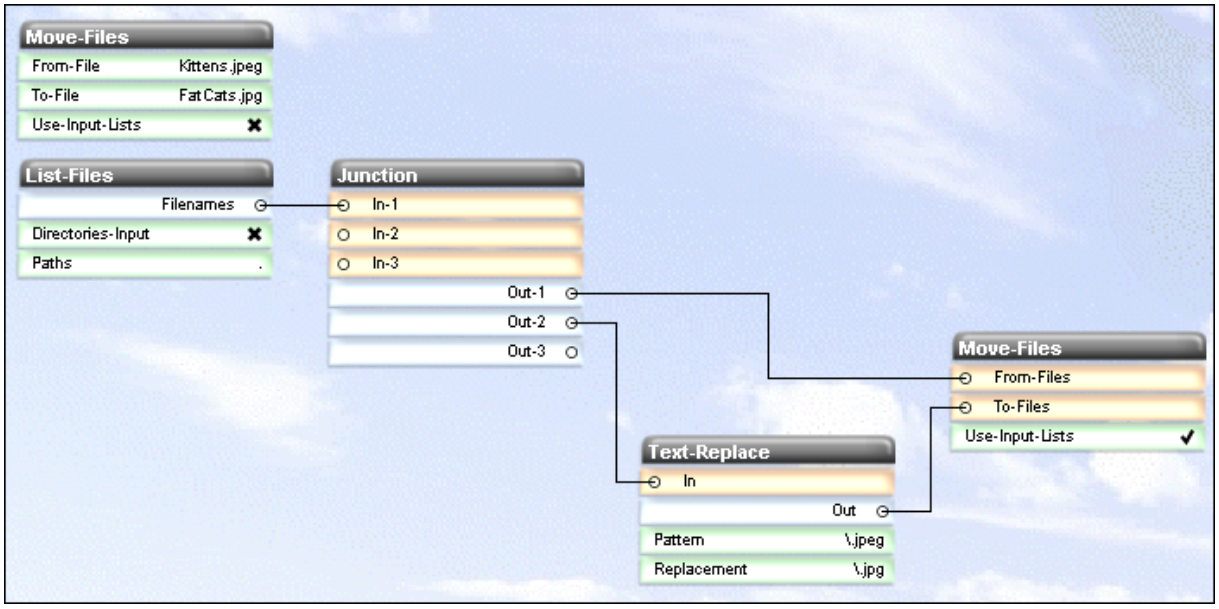


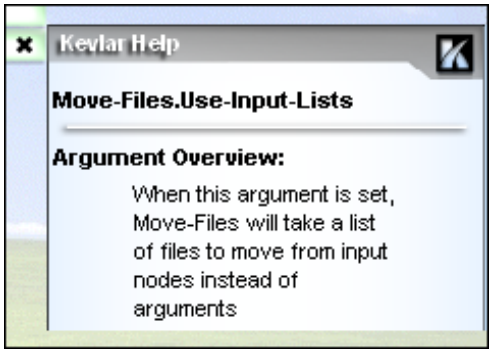
Figure 3.4 How validation meta-information is used. When an argument value set by the user does not match the pattern in the meta-information, the argument turns red. The user can right click to get this message: "Must be an integer number of milliseconds.". This message is taken from the meta-information.


```
<!-- specify the both the arguments available and
     how they map onto input and output nodes -->
<argument-model>

  <!-- First specify the arguments -->
  <arguments>

    <!-- specify a compulsory argument called
         Pattern which is initially empty -->
    <argument>
      <name>Pattern</name>
      <help>
        <overview>Specify a regular expression
            to search for</overview>
      </help>
      <is-compulsary />
      <singleline-argument>
        <initial></initial>
      </singleline-argument>
    </argument>
```

Figure 3.5 How the help meta-information is used. With the Move-Files example from Figure 3.3, a user will not know what the 'Use-Input-Lists' argument does without being told. The argument and input node help meta-information is essential for helping the user understand arguments.




```
  <!-- specify a non-compulsory boolean argument -->
  <argument>
    <name>Case-insensitive-search</name>
    <help>
      <overview>Turn on or off case sensitivity
        in the pattern matching</overview>
    </help>
    <boolean-argument />
  </argument>

  <!-- .. etc - more arguments below -->
</arguments>

<!-- specifies which argument values lead to which
     input output nodes. When arguments are updated,
     each mapping is tried in turn to see if it's
     conditions matches until a match is found -->
<mappings>
  <mapping>
    <!-- each mapping has conditions, if all the
          conditions are true, the specified inputs
            and outputs are available. Otherwise the
           next mapping is tried. -->
    <conditions>

      <!-- This condition requires that the
           Verify-match-only argument is set to false.
      <condition>
        <target>Verify-match-only</target>
        <boolean-false-condition />
      </condition>
    </conditions>



    <!-- Specify one input node, make it generic so the program can
         be used for a filename grep too for example -->
    <input-pipes>
      <pipe>
        <name>Text-lines</name>
        <type>uk.ac.ic.doc.kevlar.core.types.ITextType</type>
     <pipe-help>Lines to be matched against the pattern</pipe-help>
        <param-name>T</param-name>
```

```
        </pipe>
      </input-pipes>

      <!-- Specify two output nodes, one for positive matching lines
           one for negative matching lines -->
      <output-pipes>
        <pipe>
          <name>Matching-lines</name>
          <type>uk.ac.ic.doc.kevlar.core.types.ITextType</type>
       <pipe-help>Lines from the input that
                matched the pattern</pipe-help>
          <param-name>T</param-name>
        </pipe>
        <pipe>
          <name>Negative-matches</name>
          <type>uk.ac.ic.doc.kevlar.core.types.ITextType</type>
       <pipe-help>Lines from the input that did
                not match the pattern</pipe-help>
          <param-name>T</param-name>
        </pipe>
      </output-pipes>
    </mapping>

    <!-- ... etc ... more mappings would go here -->

  </mappings>

</argument-model>



<!-- Specify program meta information -->
<description-model>
  <name>MatchLines</name>
  <version>
    <major>0</major>
    <minor>1</minor>
    </version>
  <description>Extract lines of text that
      match a regular expression pattern</description>
  <author>Daniel Burke</author>
  <website>http://www.doc.ic.ac.uk/project/2004/362/g04362341M/MavenSite/</website>
  <copyright>
    <name>Daniel Burke</name>
    <year>2004</year>
  </copyright>
  <keywords>
    <keyword>grep</keyword>
    <keyword>match</keyword>
    <keyword>text</keyword>
    <keyword>find</keyword>
    <keyword>pattern</keyword>
  </keywords>
  <program-help>TODO: Write MatchLines help</program-help>
</description-model>

<!-- Specify how program exit codes map onto messages -->
<return-model>
  <return-mappings>
```

```
      <return-mapping>
        <value>0</value>
        <message>No errors</message>
      </return-mapping>
      <return-mapping>
        <value>1</value>
        <message>Unexpected error</message>
        <is-good />
      </return-mapping>
    </return-mappings>
  </return-model>
</program>
```

3.3 Program Representation Interfaces

Having described and motivated the meta-information we required from programs, this section discusses the API that programs to be loaded must adhere to, and what it gives them.

3.3.1 Initial API

The initial implementation required each program to come with a class that implements a Java interface for extracting the program meta-information. This simple approach allowed the programs to specify what their meta-information is through well-defined methods of the interface.

However, this requires that all programs loaded into the shell obey a provided contract, and essentially become trusted. Although Java has been built to implement a strong security model, this is at the level of securing a single JVM via policies. While there are ways to secure parts of individual pieces of running code, there is nothing to stop a loaded program from synchronizing on a system class and not releasing the lock, from accessing any other (non-system) thread, or from creating its own threads. Although there is potentially a future API coming out to solve this [JISO], it is not available yet. For a further discussion of these problems, I refer the reader to [HS01].

As a shell is a fundamental usage tool of a computer, it did not seem correct to trust a program before the user actually decides to execute it. Security for the user should be guaranteed and not assumed.

3.3.2 Secure API

The simplest way to protect the user from untrusted code is not to execute any. For this reason program argument validation and help exportation is statically defined by an XML meta-information file.

Programs still need to be executed, however it was decided that instead of executing them inside the shell's JVM, a new JVM should be created for each one. To allow programs access to the high-level discriptions of their arguments, and their (possibly many) named input and output pipes, there is a requirement that they extend a class named `AbstractProgram`, which will invoke the program at an entry point with arguments corresponding to each of the above. This is comparable with the requirement that Java Applets extend `java.applet.Applet` to be loaded into a browser.

The `AbstractProgram` class handles the communication with the shell across the standard in and out pipes, and Java's serialization mechanisms are used to communicate the high-level objects that travel across the pipes to the child processes and back, this is described further in Section 4.6

3.3.3 AbstractProgram

This is the class that programs must extend to be executed in Kevlar. Implementations must provide a zero-argument constructor (since the class is to be loaded by reflection). `AbstractProgram` provides

an entry point method with the signature:

```
public abstract void run(IArgumentInstance args,
                         Map<String, IInputPipe> inputs,
                         Map<String, IOutputPipe> outputs);
```

Here the argument instance is the programmatic representation of the argument values given to the program, as specified in its argument model. The maps of Strings to Pipes relate the named input and output pipes again as specified its argument model.

3.3.4 Packaging

To be successfully loaded into Kevlar, programs must be jar'd up, and the class file must have a mirror .xml file specifying its meta information in the same directory as the class file. Also the meta-information to the jar file must contain a name-value pair with the name "Programs", and a comma-separated list of fully-qualified class names which are the class (and xml) names of the files to load.

3.4 Type Interfaces

Types in Kevlar are much simpler than programs from the end-developer point of view. There are simply a few contractual points to be adhered to.

- **All types must descend from `IType`.** This is to ensure that types remain in a tree.
- **All types must only descend from one other type if they do not descend directly from `IType`.** This is to ensure that the resultant type-tree has only single inheritance.
- **All types must be interfaces.** This is so there is a clear distinction drawn between a type, and an implementation of a type.
- **All type implementations must be completely in the jar file that they implement.** This is so that the Kevlar shell can find the definitions as well as all children.
- **All type implementations must be Serializable.** This is so that the instances of types can be passed across streams.
- **Types must be placed in jars, with manifest annotations.** The jar files containing types must have a manifest attribute 'Types', with a comma-separated list of the fully-qualified class names of types in the jar.

3.4.1 Mutable types

Our contracts, however, do not specify that the implementation of types must be immutable. This allows for generic types to be operated upon by programs. A program can declare that it accepts inputs as any sub-type of a type, and outputs the same sub-type it recieves, but will change the value of the instances of the type passing through it by using the appropriate setter or state-changing methods. For example `ITextType` exports a method `setText( String newText ) throws IllegalArgumentException`. This allows for the text-replace program to be able to apply a regular expression to any type with text-type rooted in its heirarch (e.g `IFilenameType`) and still return the sub-type.

3.5 The 'Show' program

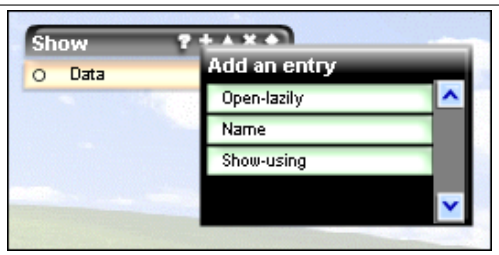
The show program is a special program in the shell used for visualizing the output of a pipeline. Unlike other programs, Show interacts with the Shell and the GUI of the shell.

Since a 2-dimensional pipeline can have many outputs, the user will place a show program wherever they wish to visualize the output of a program.

Show is used like a normal program. The data input node is connected to the output of another program and its arguments are set.

The Show program does not contain any code to display the data. Instead, it passes the data it recieves to the GUI, which selects an appropriate method to display the data. The method used by the GUI to display the data is explained in Section 6.12.

Figure 3.6 . The show program and its available arguments. The data input node takes an output of the pipeline and visualizes it.



SHOW'S ARGUMENTS

- **Name.** The GUI can use this name as the title of the visualizer window it uses to display the output.
- **Show-Using.** This argument instructs the GUI which specific visualizer to use for the data. This issue is covered in detail in Section 6.12.
- **Open-Lazily.** This argument instructs the GUI to not open a visualizer window until the first item of data comes through into Show. This is useful for Show programs attached to Error outputs.

When a pipeline is run, our Shell has some rules that relieve the user from adding many 'Show' programs.

- If that pipeline is a single program with one output, a Show program is automatically connected to that output. This makes it very easy to make small pipelines, such as one to list directory contents. This should improve the speed at which the shell can be used.
- Also, since a user should not be concerned with specifying that they want to visualize errors of programs, if a program has an unconnected output called "Error", a Show program will automatically be connected with the 'Open-Lazily' argument set on.

3.6 Programs Written For Kevlar

In order to demonstrate the Kevlar system, we have written programs so that all features of the system can be demonstrated.

- **CurrentWorkingDirectory.** This program outputs a representation of the current working directory.
- **Junction.** This program allows the multiplexing of up to 3 input pipes into (up to) 3 output pipes.
- **Split.** This program converts a single input-pipe into two output pipes.
- **Merge.** This program merges two input pipes into a single output pipe.
- **SysExec.** This program allows the execution of a system-native program, allowing passing in of standard in, and extracting of standard out.
- **TimeOutRepeater.** This program repeats any of its input upon its output, and exits after a certain quantity of time has passed with no input.
- **GetFileContents.** This program reads in file names on its input stream, and returns the textual contents of its inputs.

- **ListFiles.** This programs outputs the name of each file in directories specified by its inputs.
- **MakeDirectory.** This program takes names of directories on its inputs and creates them, outputting the names of the directories it successfully created.
- **MoveFiles.** This program takes source and destination names for files and moves the source file to the destination file.
- **ImageLoader.** This program takes the names of image files to load and outputs the images, along with their file names and any files that failed to load.
- **ImageEdit.** This program takes images as input and applies a transform to them to produce altered images on its output.
- **Show (Built-in).** This program provides either a textual or graphical representation of its inputs.
- **TextReplace.** This program allows the user to apply a regular expression replace to some text-based input to alter its value.
- **TextSearch.** This program filters input text, only outputting those inputs that match a given regular expression.
- **TextSplit.** This program takes text inputs, and splits them into multiple text-outputs

3.7 Types Written For Kevlar

In order to demonstrate the Kevlar system, we have written types so that all features of the system can be demonstrated.

- **IType.** This is the root type that all other types must descend from.
- **ITextType.** This is a basic type representing an arbitrary length string of text.
- **IFilenameType.** This is an extension of `ITextType` where the text-string represents a file path/name.
- **IImageType.** This represents image data.

Chapter 4

Framework

4.1 Overview of the Framework Design

The framework is the core of the Kevlar project; it manages the compilation and manipulation of the pipelines. It contains the components to validate the pipelines for type safety and to execute them. Furthermore the framework uses dynamic loaders to pick up user defined types and programs in real time.

The following description gives a brief overview of the components the framework is composed of.

- **Dynamic type and program loaders.** The dynamic types and program loaders load the user defined types and programs into the framework. They are able to pick up the types and programs the user has added to the system in real time, even when the Kevlar system is already running.
- **Program Repository.** The loaded programs are stored and managed within the Program Repository module of the framework. It is the access point for programs in the Kevlar system and is used by the HIAL to propagate the programs up to the GUI. The Program Repository also attaches unique identifiers to each program.
- **Type Checker.** The type checker component contains a library of functionality used to anaylse pipelines for type safety. It uses the loaded types to construct a type tree and exposes a set of functions for performing operations on types such as testing for type compatiability, and finding the least upper bound of a set of types.
- **Contexts.** The framework manages a collection of contexts. Contexts are containers for pipelines. They hold the data structure that represents the pipeline. A context has functions for adding new programs to the pipeline and for connecting them with pipes. It contains the part of the type checking algorithm that is tightly coupled with the pipeline data structure. It uses the methods of the `TypeChecker` to check user manipulations of the pipeline for type safety. Furthermore, it invokes the real time execution of the pipeline.

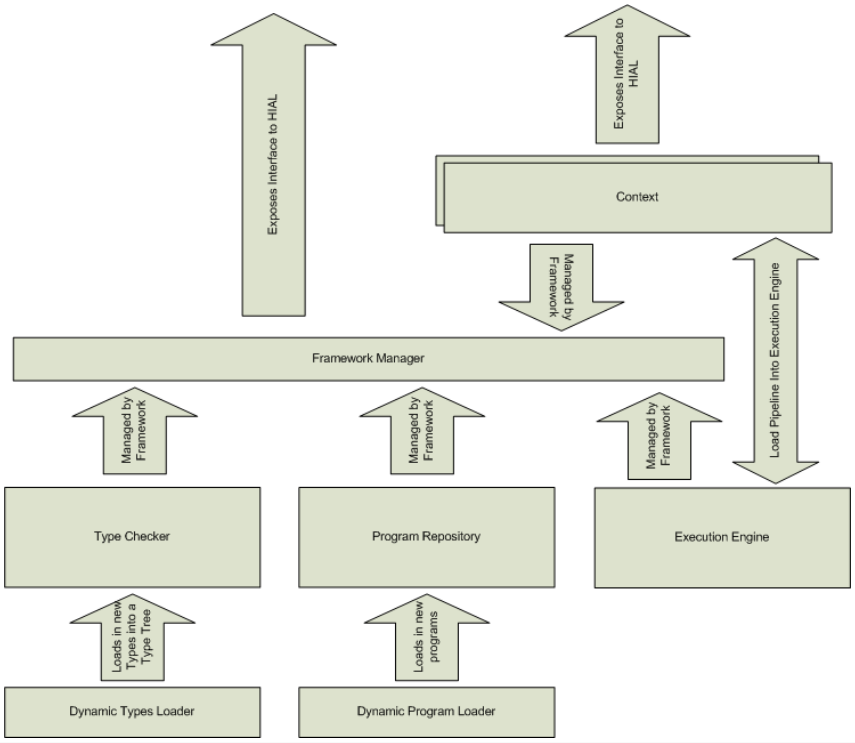
4.2 Program And Type Loading

One of the requirements for this project was a way of dynamically loading new programs and types into the system. Also, an optional extension to the project was the dynamic loading of visualisation plug-ins to the Kevlar shell.

Seeing that there would be a need to periodically scan the file system for updates, and that this was functionality common to all the potential loaders, it was decided to abstract it away into its own component, and make the loaders for types and programs subscribe to it.

The following sections describe the `IDirectoryWatcher`, `ITypeLoader` and `IProgramLoader` classes, which perform the dynamic loading of programs and types.

Figure 4.1 Block diagram showing the framework components interaction. The main components of the framework



4.2.1 IDirectoryWatcher

4.2.1.1 Design Notes

As mentioned already, the `IDirectoryWatcher`s role is to abstract away the interactions with the file system so that the other loader components can stay focused on managing programs / types.

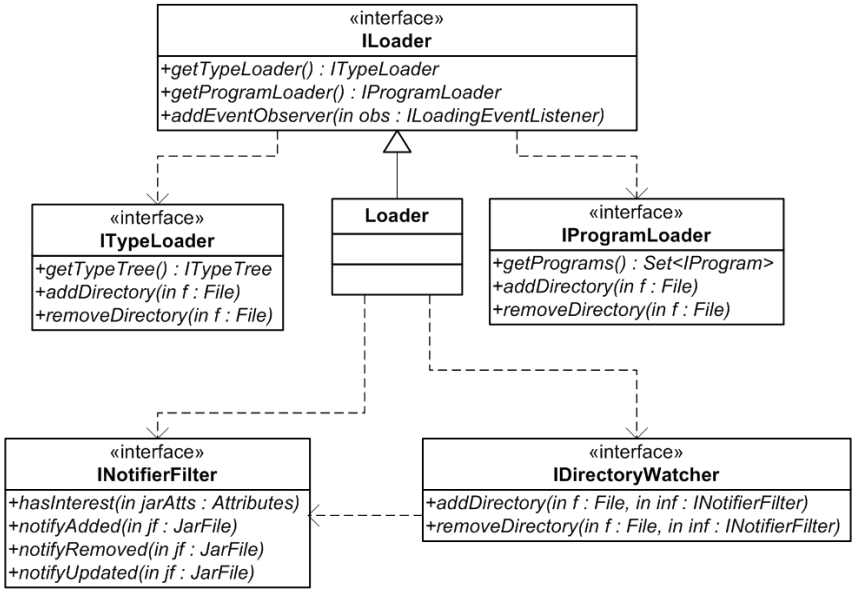
To do this in a useful way, it was decided to make the `IDirectoryWatcher` interface support a call-back event model. In this model, users of the `IDirectoryWatcher` request that a particular directory be watched and provide an `INotifierFilter` that is used to determine which jar files in the directory the notifier part of the `INotifierFilter` should be informed of.

4.2.1.2 Implementation Notes

The implementation of the `IDirectoryWatcher` was made as simple as possible. The core of the implementation revolves around its `run()` method, which re-evaluates the directories being watched for changes. Originally the `IDirectoryWatcher` had its own thread that called this method in a loop with 1 second sleeps in between. It was later decided to move the control of this thread into the `Loader` class which aggregates together all the loaders and the `IDirectoryWatcher`, so the thread could be shared by all.

While the program and type loaders where being written, it was noted that both decided whether a jar file was interesting to them depended only upon the manifest information found in the jar files. The code required to extract the manifest information (represented by a Java API class named `Attributes`) was refactored into the `IDirectoryWatcher`, and it is this information that ht `INotifierFilter` uses to decide if there is an interest in a given jar file.

Figure 4.2 The Exported Loader API



4.2.2 ITypeLoader

4.2.2.1 Design Notes

The `ITypeLoader` design was biased by its core functionality, which was to provide a way for the framework layers above it to get hold of the current tree of valid types loaded into the system. This was to be provided by the method `getTypeTree`.

A secondary function of the `ITypeLoader` was to provide a mapping from class-names of types to their `IInterfaceClass` representation for use by the argument model provider when reading in program descriptor files.

Finally the `ITypeLoader` had to provide a way to get all currently loaded jar files in a classpath representation, suitable for the forked children of Kevlar to use.

4.2.2.2 Implementation Notes

The implementation of the `ITypeLoader`, like that for the `IDirectoryWatcher`, is centered on its `run()` method. Here the method deals with any jar events (added, removed, updated) that have occurred concerning types. Originally this method was designed to be run in a thread that waits on its event queue, but it was felt that this was wasteful of a thread (and its associated resources), and was refactored to instead just deal with any events that may be in the queue, and then return.

As outlined in Section 3.4 types are exported in jar files which contain manifest information with a tag "Types" and values being the canonical class[1] name of the type. Upon jar added events, the type loader loads in the jar and extracts the types it contains from its manifest information. It then adds the URL of this jar to a customised `URLClassLoader`. [2] This `URLClassLoader` is reused in the execution system (see Section 4.6.3) to load in the class definitions of the implementations of types (that by the type specifications must also be in the type-jar files.

Once the type classes are loaded in, it is checked that they are Java interface classes, and that they do extend from `IType`. With their validity established, they are added to the integral set of available types.

[1] The canonical class name is the fully-qualified package and class name of a Java class, e.g. `java.lang.Math` as opposed to just `Math`.

[2] It has been customised to force the usage of this class loader and not its parent to ensure that classes that should be loaded from plugged-in jars do so. This is because the unit tests, and eclipse place the entire source tree on the class-path by default which includes the sources for all types, although the built version of Kevlar has the types (and program) in separate jars from the main Kevlar jar.

After all the events have been processed, if the state of the `ITypeLoader` has changed, it rebuilds its type tree. This is a representation of the type hierarchy of all loaded types augmented by the built-in types `IType` (the root type), and `IImageType`. The type tree is checked during construction for possible multiple inheritance (which, for simplicity we have deliberately disallowed).

During the testing of the `ITypeLoader`, it was discovered that under the Windows™ operating system, the use of an `URLClassLoader` "locks" any jars that it refers to, meaning only the Kevlar JVM can alter (update/delete) the file. Researching this problem revealed that separate class-loaders for each jar and careful control of all references to the classes they contain, coupled with a mechanism for the user to specify the types to unload from within Kevlar would provide a solution.[3] This was felt, however, to be too much work for a feature that is unlikely to be used and is not central to the project. The default alternative (requiring the user to restart Kevlar when wishing to remove type jars), although not perfect, is acceptable for our application given the time restrictions this project has been under.

4.2.3 IProgramLoader

4.2.3.1 Design Notes

The `IProgramLoader`s sole functionality is to provide the `ProgramRepository` with the set of `IProgram` objects, representing programs that can be used by the Kevlar shell. To this end, the `IProgramLoader`s principal exported method is `Set<IProgram>> getPrograms()`.

4.2.3.2 Implementation Notes

Again, the `IProgramLoader`s `run()` method deals with jar events. For each new jar, its manifest is parsed and program class names are discovered. As described in [TODO xref descriptors] the corresponding Program Descriptor files are loaded and parsed to establish argument, description and return-value models for the program. These are then used to build the `IProgram` representation of this program, and it is added to the internal set of available programs.

Additionally, upon the calling of `getPrograms()`, any built-in programs are added to the set before it is returned.

4.3 Contexts

4.3.1 Specification

The framework has a complete set of functionality for constructing, type checking and executing pipelines.

- **Pipeline construction.** The Context provides functions to the user to manipulate pipelines. This includes adding and removing of programs to a pipeline, as well as adding and removing pipe connections between programs.
- **Pipeline validation.** The Context will perform type checking each time a pipe is added or removed, and each time a program's IO nodes change due to arguments changing. The type checking algorithm is covered in detail in Section 4.5. This Context will use the functionality of the `TypeChecker` and consequentially the loaded Type Tree to validate the type safety of the pipeline.
- **Pipeline execution.** The Context interacts with the framework's execution engine to execute type safe pipelines the user has constructed.

A Context encapsulates a large amount of functionality needed by the HIAL. Its interface contains every action that might be done to a pipeline, and has methods for retrieving any useful sets of information on the pipeline. It also has a listener model for notifying the HIAL whenever a change occurs to the pipeline. For example, if a program is removed from the pipeline due to that program being unloaded by the user, the context will remove the program and send a `programFailed` notification event.

[3] We would like to that Robert Chatley <`http://www.doc.ic.ac.uk/~rbc`> for these suggestions.

4.3.2 Pipeline Construction

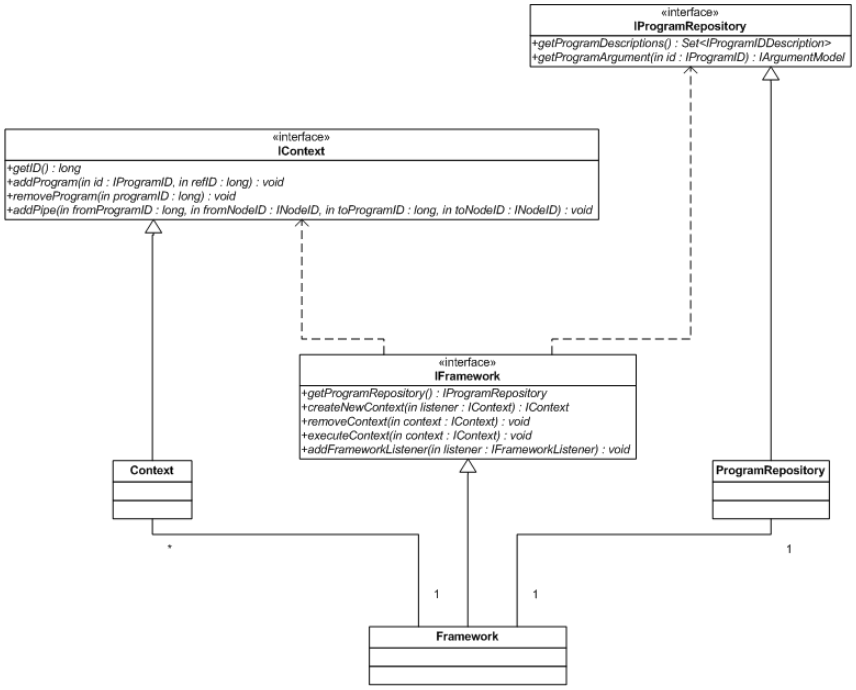
4.3.2.1 Implementation Notes

Getting a handle to a context. The first step to start constructing a pipeline is to obtain a handle on a new `IContext`. This is done by calling `IFramework`'s `createNewContext(...)` function which will add a new `Context` to the framework manager and return a reference to an `IContext`.

Adding a program to the pipeline. Before adding a new program to the pipeline, the list of loaded programs must first be accessed. As previousely explained, the Program Repository offers the HIAL access to the set of loaded programs and attaches unique identifiers to each program in the form of `IProgramID`s. To get access to the Program Repositry, the HIAL calls `IFramework`'s `getProgramRepository()` which will return a handle to the `IProgramRepository`. By calling `getProgramDescriptions()` we get a list to the loaded programs and their associated `IProgramID`s. Once the `IProgamID` of the program we want to add has been obtained, we can call `IContext`'s `addProgram(...)` method to add the program to the pipeline contained within the `Context` in the form of an `IPipelineProgram`.

Adding a pipe to the pipeline. Every `IPipelineProgram` that has the ability to connect input and/or output pipes exposes a collection of `INode`s used for the connection of input pipes and a collection of `INode`s used for the connection of output pipes. An `INode` can be seen as a connection point and also has an associated `INodeID`. Having obtained the two `INodeID`s of the programs we want to connect, a call to `IContext`'s `addPipe(...)` can be made to create a pipe between them. This action will be type checked internally by the context with the help of the `TypeChecker` and will inform the user if this connection is type safe and how it affects the other connected pipes in the pipeline. This will be detailed further in the next sections.

Figure 4.3 Framework public interface dependencies



4.3.3 Pipeline validation

4.3.3.1 Design Notes

After every user manipulation of the pipeline contained by the context, the context starts off a type checking routine to verify the validity of the last action. This happens whenever a user connects programs with a pipe, when a pipe is removed, when a program's arguments change causing that pro-

gram's IO nodes to have changed types, when a program is removed and consequentially the connected pipes to it are removed, or when the type tree has been modified.

4.3.3.2 Implementation Notes

The main entry point to the type checking of the pipeline is the private method `floodTypeCheck(...)` which is implemented by `Context`. The type checking algorithm, will try to consolidate with the user's change by perturbating the change through all the connected programs which are affected by the change. By consolidation we mean that types associated with `IParameteredTypeClass`es will be upcasted appropriately to make the new pipeline valid. Furthermore the algorithm will verify that two `IBasicTypeClass`es which are connected are valid when the input type is a subtype of the output type according to the loaded `TypeTree`. When the algorithm finds that the pipeline is not valid, it will tell the user which pipes are no longer valid after the change, else it will have updated the pipeline with the consolidated types and tell the user that the action was valid. This explanation might seem vague, but is explained in more detail in the Type Checker chapter which goes indepth about `IBasicTypeClass`es and `IParameteredTypeClass`es.

4.3.4 Pipeline Execution

4.3.4.1 Design Notes

The Context interacts with the framework's execution engine to execute type safe pipelines the user has constructed.

When a Pipeline has been validated as type safe by the type checking algorithm, the user may choose to execute it. The `IContext` contains the `execute()` method which will pass all the pipeline information to the execution engine, which in turn will run the pipeline.

4.3.4.2 Implementation Notes

Without going into too much details about the implementation, every `IPipelineProgram` contains the `run(...)` method. This method will start the execution of the program. Once all programs are running, the pipeline is in effect executing and I/O results are passed over the pipes between the programs. This will be further detailed in the Program Execution section.

4.4 Type Checking

This section assumes an understanding of lattice and type theory. The appendix contains a chapter on these topics, we therefore recommend the reader to refer to these sections if parts of the design and implementation of the Type Checker are unclear.

4.4.1 Motivation

4.4.1.1 Advantages of types

In a traditional command line console, the data that flows across pipes is raw binary data without any sense of type. The output is always binary data represented as ASCII text on the console. Although it is possible to make an image editing program that works on the binary data that it gets on its standard input, it is difficult as the binary data needs to be parsed, validated as being correct, and typically, only one image can be sent on the input this way. In our shell, images are represented as image objects. They do not need to be parsed by the program that consumes them, and the programs can take a list of images on its input. The image's invariants are guaranteed to hold.

Although, much data can be represented as text, a lot of information can be lost by converting the data to this loosely structured form. For example, an email and its headers can be represented as text. If you wanted to pipe the email into a spam filter, it would have to parse the email itself. If instead, we invented a standard email type with member fields, programs could be made which take the email object directly and use its member fields to get the email sender, the send time, and maybe attach information to the object such as a spam score. A special email client could be made that outputs AdvancedEmail objects, a sub-type of email. AdvancedEmail types can store a list of related emails (replies etc), and spell checking information, yet remain completly compatiable with programs that only take Email objects.

Finally, in traditional shells, output of an email pipeline would be shown as text. Our shell could have an associated visualizer for the Email and AdvancedEmail types. This visualizer could show the email text, but allow you to expand and contract the email headers, depending on if you wish to see them.

For these reasons, we realised that types would be a useful addition to our shell.

4.4.1.2 Consequences of types

However, when you add types, programs will expect those types, and will crash if they receive an object that is not compatiable with the type they are expecting. Therefore, type safety must be enforced over the pipeline.

The addition of type safety in the Kevlar system offers many benefits. First of all the user will be certain that its constructed pipeline is logically correct with regards to the types of information flowing over pipes, and can therefore assume that execution of the pipeline should not have unexpected behaviour relating to type mixups. This is comparable with a programming language that checks that the types of the arguments provided via a function call match the function signature. If they do not, it is certain the user has made a mistake.

In traditional shells, a common complaint is that it is hard to know which program to use to solve a problem. While our shell attempts to address this problem with program search, and program categorisation, another possiblitity can be to have a hint system which will list all the available programs in the repository which could be connected safely to the currently selected program. For example, the user places down a program that loads images, then selects its image output. Then the GUI could offer a set of programs that can take the image type as input.

4.4.2 Design Notes

After every manipluation of the pipeline by the user the `TypeChecker` will be invoked to perform type checking operations. The `TypeChecker` is hidden to the HIAL and will be invoked by the `Context`.

The hierarchical derivation structure of types is encapsulated by the `TypeTree`. The `TypeTree` is available to the `TypeChecker` so that it can use it to validate the pipeline.

In order to describe what the Input and Output types are that the program accepts the framework provides `IBasicTypeClass` which is used for basic tyoes, non parametered (ie. templated), and `IParameterdTypeClass` which is used for parametered (templated) types which contain a name as well as an upper bound. The types the `IBasicTypeClass`es and `IParameteredTypeClass`es refer to must be present in the loaded `TypeTree`.

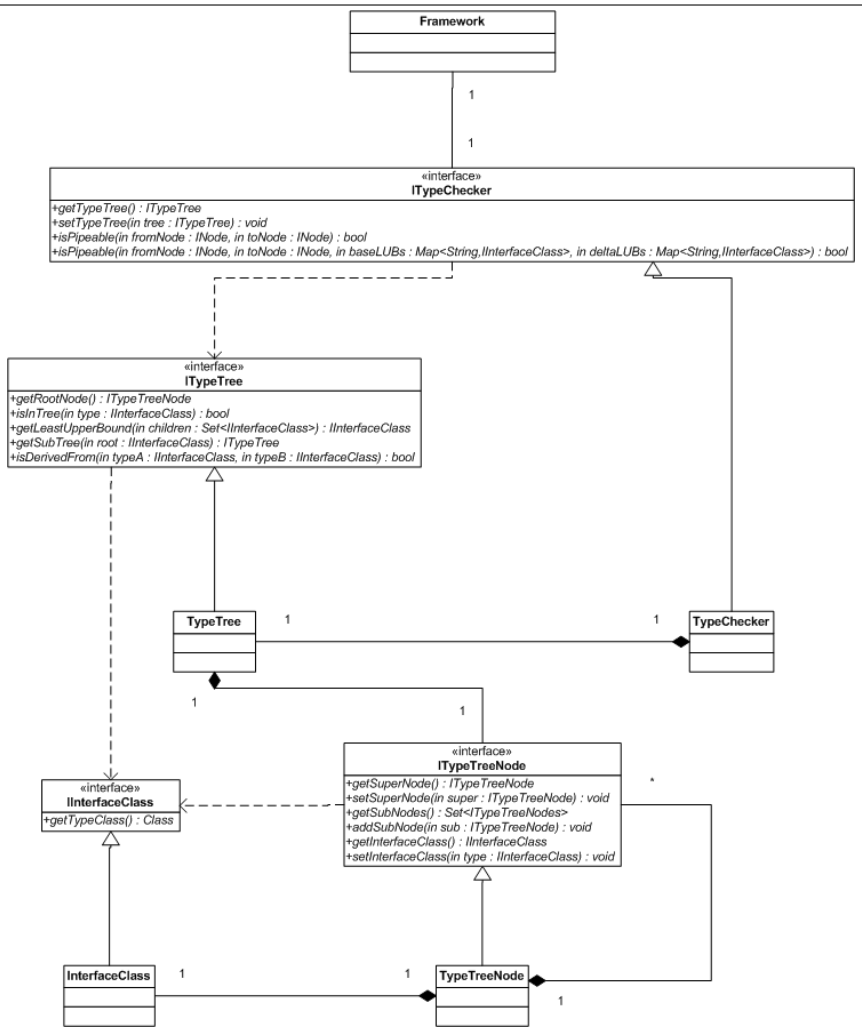
The algorithm that validates the pipeline is called the type flow algorithm. This algorithm is implemented by `Context`'s `floodTypeCheck(...)` as described in the Context chapter. To implement this method `Context` calls methods on `TypeChecker` which in turn uses `TypeTree`.

4.4.3 Implementation Notes

4.4.3.1 The Type Tree

One of the components used by the `TypeChecker` is the `ITypeTree`. At start up and whenever a user adds a new type to the system, a new `TypeTree` will be constructed using the dynamic type loader. The

Figure 4.4 Design of the TypeChecker component



`TypeTree` is a hierarchical tree that shows inheritances of the types. At the root of every `TypeTree` is the `IType` type. The tree will not support multiple inheritance, so every type will contain one and only one superclass.

The most interesting methods implemented by `TypeTree` are `IInterfaceClass getLeastUpperBound(Set children)` and `boolean isDerivedFrom(IInterfaceClass typeClassA, IInterfaceClass typeClassB)`. `getLeastUppeBound` calculates the Least Upper Bound of a set of types in the tree and `isDerivedFrom` returns true if and only if the first type is derived from the second type, ie. lower-bound or equal to the second type in the `TypeTree`. These methods form the basic calculations to support the type flow algorithm, which performs the validation of the pipeline. The `TypeChecker` uses `TypeTree` to implement its validation methods.

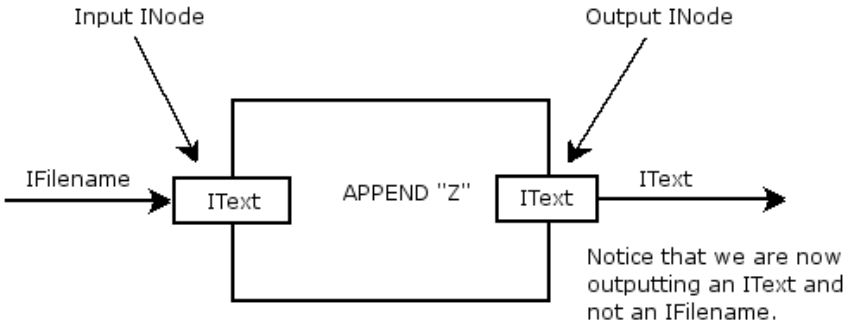
4.4.3.2 Input and output types

User defined programs in Kevlar will have zero or more `INode`s to which input pipes can be connected to and zero or more `INode`s to which output pipes can be connected from. Every Node is associated with an `IBasicTypeClass` or an `IParameteredTypeClass`. Both `IBasicTypeClass`es and `IParameteredTypeClass`es will contain references to a type in the `TypeTree`. However `IBasicTypeClass`es are used when the type of the input or output is known at the design time of the program whereas `IParameteredTypeClass`es are used when only the upper bound type of the input and output in the tree is known and when the type needs to be inferred by the `TypeChecker`. The following sections will go into more detail about `IBasicTypeClass` and `IParameteredTypeClass`.

IBasicTypeClass If the type of the input and output data is known by the designer of the program and does not need to be inferred dynamically by the Kevlar framework, the designer should associate `IBasicTypeClass`es to the classname. An `IBasicTypeClass` contains an `IInterfaceClass` which refers to a type in the `TypeTree`.

To illustrate this with an example, imagine the user wants to have a program that accepts an input pipe with `IText` as its type and will then append the letter 'Z' to it. It would then simply output the edited `IText` via the output pipe. So the designer of the program knows that the input will be of type `IText` and also knows that the output should be of type `IText`. If in the type system, `IFilename` is extended from `IText`, one could also pass IFilename to this program, although the output would still be upcasted to `IText` as the designer of the program expected. If this behavior of upcasting and associated information loss is not wanted, the user should associate `IParameteredTypes` to the `INodes`.

Figure 4.5 IBasicClassType upcasting. Diagram of a Command which upcasts `IFilename` to `IText`.

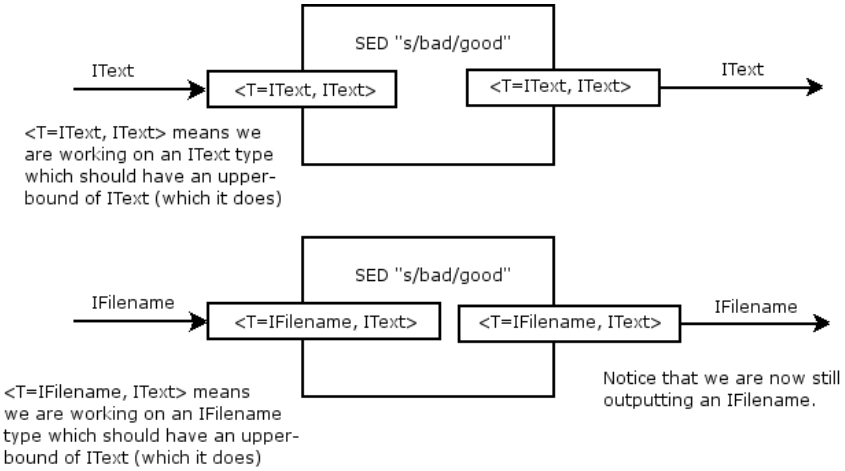


IParameteredTypeClass If the designer of a new program does not want to restrict its output to a predefined type as in the example above, an `IParameteredTypeClass` should be asscoiated with the `INode`.

An `IParameteredTypeClass` contains the name of the parameter and an `IInterfaceClass` which refers to a type in the `TypeTree` to give an upper bound to the types that can be handled or outputted by the command.

By way of example, if a group of third party developers wants to make a program that is generic for all `IText` types and subclassed types of `IText`, eg. `IFilename`, it should associate an `IParameteredTypeClass` with the upper bound `IText` to the `INode`s. An instance of this is to make a program like `sed` that takes in any subtype of `IText` and returns an edited version of it but with the same type.

Figure 4.6 IParameteredTypeClass example. Diagram of a program which uses `INode`s associated with `IParameteredTypeClass`es. Notice that if the user had inputted data with type IImage Kevlar would reject this since it has not an upper bound of IText as specified by the IParameteredType.



4.4.3.3 The TypeChecker

The `TypeChecker` contains the validation functions which support the validation of the pipeline. The most important methods it exposes to the `Context` are `isPipeable`, `getLeastUpperBound` and `inferTypeOfNode`. Method `isPipeable` returns if two programs in the pipeline are connectable with a pipe, `getLeastUpperBound` simply calls the same method on the typetree it contains, and `inferTypeOfNode` returns the type of the Node after templates types have been updated with their new upper bounds.

4.5 Type checking algorithm

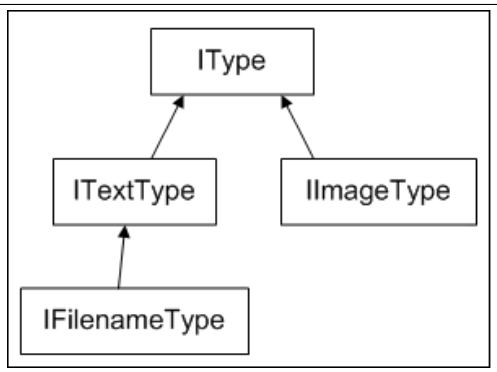
4.5.1 Symbols and terms

- **T < S.** T is a proper subtype of S.
- **T <= S.** T is a subtype of S. (T may be the same as S)
- **Least upper bound (LUB).** As explained in Section 10.1, the least upper bound of a set of types is their common ancestor. Since this term is long, it will be shortened to LUB.
- **A node is of type Text.** This notation will be used to say that a node has a type Text. (Which would actually be ITextType in the Kevlar system). Therefore the node has a `IBasicTypeClass` object to represent its type.
- **A node is of type A : Text.** This notation will be used to say that a node has a parameter name "A", and the upper bound of that node is the type Text. Therefore the node has a `IParameteredTypeClass` object to represent its type.

4.5.2 Type tree example

In order to allow explanations of the type checking algorithm, we will use an example type tree. This type tree is shown in Figure 4.7

Figure 4.7 An example type tree. IFilenameType < ITextType < IType. IImageType < IType. We will use the short names, Filename, Text, IType, and Image to refer to these types.



4.5.3 Bound parameters

At any time in the lifetime of a context, each program in the pipeline will have a set of parameter-type bindings. These bind each parameter name to a type.

For example, if a program has two input nodes of type A : TypeT and a single input node of type B : TypeT, then the program in the context would have to store the current calculated type binding of A and B, and these types would be a subclass of TypeT. Note that this is just a cached value, as it can be

recalculated at any time. However, one of the class invariants of the `Context` class is that the parameter-type bindings of a program are always up to date.

4.5.4 type-safe

First, some more explanations of terms.

- If an output node is of type T, we say it outputs type T.
- If an input node is of type T, we say it requires type T.
- If an output node is of type A : T, and A is bound to S for that program, we say it outputs type S.
- If an input node is of type A : T, and A is bound to S for that program, we say it requires type S.

4.5.4.1 Runtime condition test for type-safety

We can informally take type-safe to be the state of a pipeline, such that when it is run, a condition is guaranteed. This condition is that: If all programs that are defined to output a type T on a certain output node, only outputs objects of types S >= T on that node, then all programs that are defined to require an object of type U on their input node will only receives objects of type V >= U.

The above definition is an intuitive definition. It clearly guarantees stable execution of the system. Note that for the above condition, it must be assumed that all programs may output objects, or may not, and that if a program has input nodes and output nodes of type A : TypeT, and none of those input nodes are connected, it is impossible for that program to output any objects on those output nodes. This can be guaranteed since if the program does not receive any objects on its inputs of type A : TypeT, it will have no way of taking those objects and manipulating them to output them as discussed in Section 3.4.1.

4.5.4.2 Constraint test for type-safety

We can also express type-safe as being the state of a pipeline when it adheres to a set of constraints.

CONSTRAINTS

- **1.** When an output node that outputs TypeT from one program is connected to an input node that requires TypeS, it is required that TypeT <= TypeS.
- **2.** When a program has several inputs, all of type A : TypeT, the parameter-type binding of A for that program must be the LUB of all the output types from outputs that are connected to those inputs. A must be bound to a type TypeS, such that TypeS <= TypeT

There is an issue however. Given the second constraint, what if there are no connected inputs? What is the LUB of an empty set of types? For now, we will modify the constraint, so that if the connected inputs is an empty set, the parameter A (A : TypeT), is bound to TypeT.

Given the above modification, it is actually the case that the constraint method of testing for type safety has a set of type-safe pipelines that is a subset of the set for the runtime condition method. The reason for this is covered below, and the constraints will be modified later to cope with this issue.

We need to use the constraint method for testing for type-safety as this is the only one that can be tested for in the implementation of the type checking algorithm.

4.5.5 Motivating examples

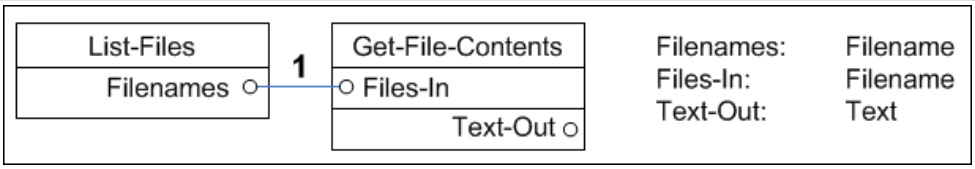
A set of examples is needed to demonstrate type-safe pipelines vs non-type-safe pipelines. In the following examples, pipelines are shown with numbers next to each pipe. These numbers represent the order in which pipes are added. We will discuss whether the pipeline is type-safe after each pipe addition until all pipes are added.

4.5.5.1 A simple two program pipeline

Figure 4.8 shows the simplest pipeline that can exist with more than one program.

- **pipe 1. added.**
  - **Constraint 1.** Holds trivially since Filename <= Filename.
  - **Constraint 2.** Is vacuously true. (Since there are no inputs of type A : TypeT.)

Figure 4.8 Example 1.. This pipeline would output the contents of all files in the current directory.

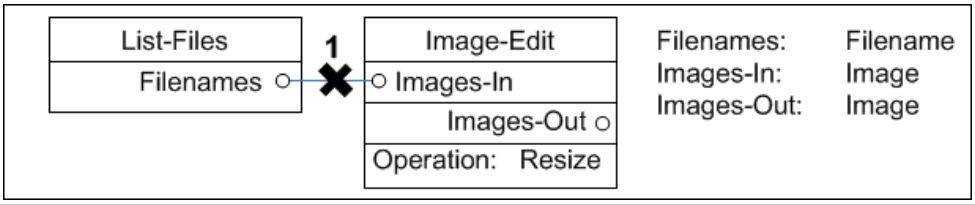


4.5.5.2 A pipeline that is not type-safe

Figure 4.9 shows a pipeline that fails constraint 1.

- **pipe 1. added.**
  - **Constraint 1.** Fails since it is not true that Filename <= Image.
  - **Constraint 2.** Is vacuously true. (Since there are no inputs of type A : TypeT.)

Figure 4.9 Example 2.. This pipeline is illogical, the user has clearly made a mistake. The user may have meant to use the Image-Loader program to first load images from filenames.



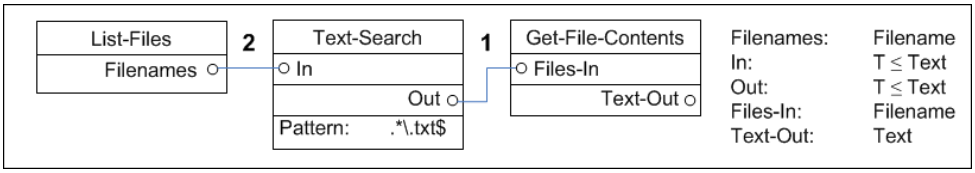
4.5.5.3 A pipeline that uses parameterized types

Figure 4.10 shows a pipeline that is not type-safe after the first pipe is added, but then becomes type-safe again after the second pipe is added.

- **pipe 1. added.**
  - **Parameter-bindings.** The `Context` would attempt to calculate the binding of T : Text so that it satisfies constraint 2. Since at this stage, only pipe 1. is connected, there are no connected inputs for Text-Search. Therefore, T is bound to Text, which is the upperbound.
  - **Constraint 1.** Fails since it is not true that Text <= Filename.
  - **Constraint 2.** Is satisfied because the context choose Text-Search's T to be bound to Text as the constraint requires.
- **pipe 2. added.**
  - **Parameter-bindings.** The `Context` would attempt to calculate the binding of T : Text so that it satisfies constraint 2. Since at this stage pipe 2. is also connected, there is one connected input for Text-Search. Therefore, T is bound to Filename, which is the least upper bound of { Filename }. (Filename is the output type of List-Files's 'Filenames' node.)
  - **Constraint 1.** Is satisfied since for pipe 1. Filename <= Filename, and for pipe 2. Filename <= Filename

– **Constraint 2.** Is satisfied because the context choose Text-Search's T to be bound to Filename as the constraint requires.

Figure 4.10 Example 3. This pipeline outputs the contents of all text files in the current directory.



Clearly, it is desirable that if a pipeline is type-safe, and any set of pipes are removed, the pipeline is still type-safe. If this is not the case, the user will have to worry about adding pipes in the correct order as is the case above. If in the example in Figure 4.10, pipe 2. is added before pipe 1. the pipeline is always type-safe. In fact, later it is shown that if a pipe is removed from a type-safe pipeline, the resulting pipeline is still type-safe by the runtime condition measure. Therefore, by transitivity, type-safe pipelines are still type-safe when any amount of pipes are removed.

4.5.5.4 Solving the problem in example 3

The root of the problem is that the constraints test for type-safety guarantees execution stability, but does not class all pipelines as being type-safe, which are type-safe under the runtime condition test.

The reason for this is that the runtime condition test assumes that a program with input nodes and output nodes of type A : TypeT, is guaranteed not to output any objects on those output nodes if the input nodes are all unconnected. Therefore it does not matter if the pipe satisfies constraint 1.

As we developed the Kevlar type system, we realised this was the case, and modified the constraints test. We alter the rules for parameter-type binding. A program's list of parameter name to type bindings may now additionally bind a parameter to "*" (Star). This essentially means that the parameter name is unbounded. The constraints have to be modified to cope.

NEW CONSTRAINTS

- **1.** When an output node that outputs TypeT from one program is connected to an input node that requires TypeS, it is required that TypeT <= TypeS or that TypeT is * (unbound).
- **2.** When a program has several inputs, all of type A : TypeT, the parameter-type binding of A for that program must be the LUB of all the non-* output types from outputs that are connected to those inputs. If this set of output types is an empty set, A must be bound to *. A must be bound to a type TypeS, such that TypeS <= TypeT, or TypeS is *.

4.5.5.5 Re-visiting example 3

Referring back to Figure 4.10, we will now look at how the new constraints hold as pipes are added.

- **pipe 1. added.**
  - **Parameter-bindings.** The `Context` would attempt to calculate the binding of T : Text so that it satisfies constraint 2. Since at this stage, only pipe 1. is connected, there are no connected inputs for Text-Search. Therefore, T is bound to *.
  - **Constraint 1.** Is satisfied since the output node of Text-Search is outputting *.
  - **Constraint 2.** Is satisfied because the context choose Text-Search's T to be bound to * as the constraint requires.
- **pipe 2. added.**
  - **Parameter-bindings.** As before, the `Context` would attempt to calculate the binding of T : Text so that it satisfies constraint 2. Since at this stage pipe 2. is also connected, there is one connected input for Text-Search. Therefore, T is bound to Filename, which is the least upper bound of { Filename }. (Filename is the output type of List-Files's 'Filenames' node.)

- **Constraint 1.** Is satisfied since for pipe 1. Filename <= Filename, and for pipe 2. Filename <= Filename
- **Constraint 2.** Is satisfied because the context choose Text-Search's T to be bound to Filename as the constraint requires.

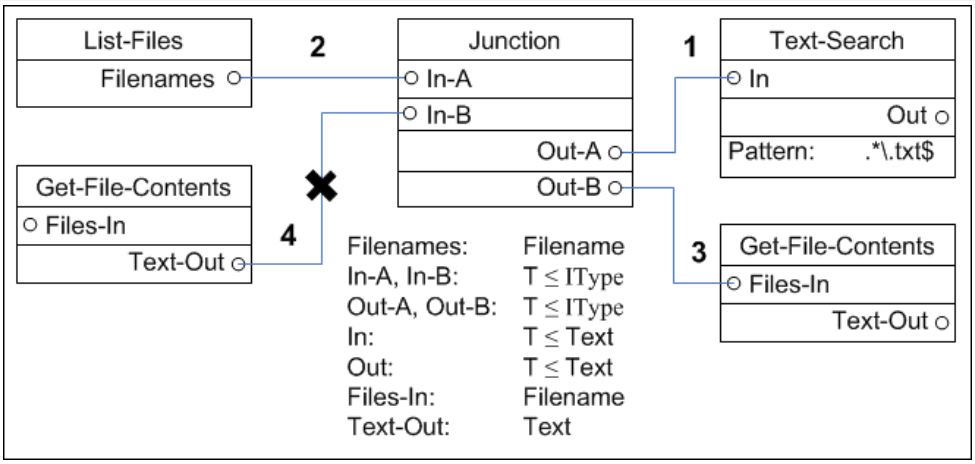
4.5.5.6 A non-linear pipeline.

Figure 4.11 shows a pipeline that fails when the final pipe is added.

- **pipe 1. added.**
  - **Parameter-bindings.** The `Context` would attempt to calculate the binding of Junction's T : IType and Text-Search's T : Text so that it satisfies constraint 2.
    - **Junction.** Since at this stage, only pipe 1. is connected, there are no connected inputs for Junction. Therefore, T is bound to *.
    - **Text-Search.** Note that since parameter names have program scope, Junction's T and Text-Search's T are different type-parameters. Since the output of Junction into Text-Search is currently *, there are no non-* types being inputted into Text-Search. Therefore Text-Search's T is also bound to *.
  - **Constraint 1.** Is satisfied for pipe 1. since the output nodes of Junction are outputting *.
  - **Constraint 2.** Is satisfied because neither Junction, nor Text-Search have any non-* types being inputted into them, so binding T to * for both is the required binding.
- **pipe 2. added.**
  - **Parameter-bindings.**
    - **Junction.** Now that List-Files is connected to Junction, Junction's T is bound to Filename. Therefore 'Out-A' and 'Out-B' of Junction output Filename.
    - **Text-Search.** Text-Search is receiving Filename on its 'In' input, so Text-Search's T is bound to Filename also.
  - **Constraint 1.** Is satisfied.
    - **Pipe 1.** Is correct, since the output of Junction's 'Out-A' is Filename, and Text-Search's 'In' requires input of type Filename (As Text-Search's T is bound to Filename). (Filename <= Filename)
    - **Pipe 2.** Is correct, since the output of List-Files is Filename and Junction's 'In-A' requires input of type Filename (Since Junction's T is bound to Filename). (Filename <= Filename)
  - **Constraint 2.** Is satisfied.
    - **Junction.** The LUB of { Filename } is Filename, so T is correctly bound to Filename
    - **Text-Search.** Again, the LUB of { Filename } is Filename, so T is correctly bound to Filename
- **pipe 3. added.**
  - **Parameter-bindings.** These are unchanged, Junction and Text-Search's T are both bound to Filename.
  - **Constraint 1.** Is satisfied for pipe 1. and pipe 2. as before.
    - **Pipe 3.** Is correct, since the output of Junction's 'Out-A' is Filename, and Get-File-Content's required input type on 'Files-In' is also Filename. (Filename <= Filename)
  - **Constraint 2.** Is satisfied as before.
- **pipe 4. added.**
  - **Parameter-bindings.**

    * **Junction.** List-Files and Get-File-Contents are connected to Junction, Junction's T is bound to Text which is the LUB of {Filename, Text}. Therefore 'Out-A' and 'Out-B' of Junction output Text.
    * **Text-Search.** Text-Search is receiving Text on its 'In' input, so Text-Search's T is bound to Text also.
  - **Constraint 1.** This time, this constraint Fails on pipe 3.
    * **Pipe 1.** Is correct, since the output of Junction's 'Out-A' is Text, and Text-Search's 'In' requires input of type Text. (Text <= Text)
    * **Pipe 2.** Is correct, since the output of List-Files is Filename and Junction's 'In-A' requires input of type Text. (Filename <= Text)
    * **Pipe 3.** Is incorrect. The output of Junction's 'Out-B' is Text, but the second Get-File-Contents requires type Filename on its 'Files-In' input node. (Note that it is not true that Text <= Filename)
    * **Pipe 4.** Is correct, since the output of the first Get-File-Contents is Text and Junction's 'In-A' requires input of type Text. (Text <= Text)
  - **Constraint 2.** Is satisfied.
    * **Junction.** The LUB of { Filename, Text } is Text, so T is correctly bound to Text
    * **Text-Search.** The LUB of { Text } is Text, so T is correctly bound to Text

Figure 4.11 Example 4. This pipeline is an example pipeline to show how parameterized types are used with Junction. If objects of type Filename and Text are fed into Junction, the output type from Junction's output nodes is Text. (The LUB.) This causes a failure of constraint 1 on pipe 3.



Of particular interest with this example, is that adding the pipe from List-Files to Junction, not only changes the type that Junction's T is bound to, but also changes the type that Text-Search's T is bound to. The change has propagated through the pipeline.

4.5.5.7 A pipeline that contains a loop

Figure 4.12 shows a pipeline that has a loop, and requires care when the `Context` sets the parameter-type bindings.

- **pipe 1. added.**
  - **Parameter-bindings.** The `Context` can actually choose any type to bind Junction's T to (including *). If it chooses to bind T to Text, Filename, Image, or *, then constraint 2. will hold for all of them. This is because the loop back causes 'positive feedback' with the type system. However, the best choice is *.

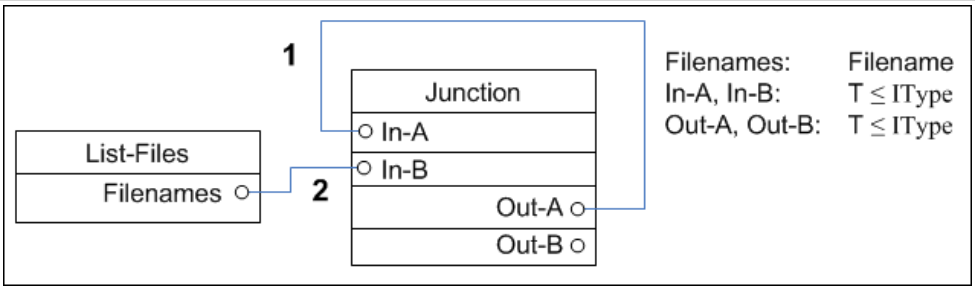
Consider the case where T is bound to Text. This satisfies constraint 2. Now, if the user wants to pipe Junction's 'Out-B' into Get-File-Contents' 'Files-In' node, it will not work as T must be bound to Filename. For this reason, T should always be bound to the most specialised type that satisfies constraint 2 to make the pipeline the most flexible for future pipe additions. The most specialised type that satisfies the constraint can be complex to determine if there is a pipeline that contains many loops. If it is possible to bind T to *, this should be done, as this makes the pipeline the most flexible possible.

In this case, if T is bound to *, any program can be connected to 'Out-B' of Junction. Therefore, T is bound to *.

- **Constraint 1.** Holds since 'Out-A' outputs *.
- **Constraint 2.** Is satisfied because Junction receives no non-* inputs, so * is a correct binding for T.

- **pipe 2. added.**
  - **Parameter-bindings.** Now, T is bound to Filename, which is the LUB of { Filename }.
  - **Constraint 1.** Is satisfied.
    - **Pipe 1.** Is correct, since 'Out-A' outputs type Filename, and 'In-A' requires type Filename. (Filename <= Filename)
    - **Pipe 2.** Is correct, since output of List-Files is Filename and Junction's 'In-A' requires input of type Filename. (Filename <= Filename)
  - **Constraint 2.** Is satisfied because T is bound to the LUB of { Filename }.

Figure 4.12 Example 5.. This pipeline sends the filenames from the current directory into an endless spin-cycle.



If pipe 2. is removed from the above example after it is added, it is notable that the binding for T should return to *, and may not stay at Filenames.

4.5.6 Context's algorithm for adding and removing pipes.

Given the above examples, the task of the `Context` can now be expressed as, finding the most-specific new type bindings for each type parameter-name that satisfy constraint 2. whenever the bindings might change.

There are 3 times when the bindings might need to change

- A pipe has been added.
- A pipe has been removed
- The arguments have changed on a program causing the type of its IO nodes to change.

To simplify implementation, whenever an argument changes, all connected pipes are saved and disconnected first, then the IO node type changes are refreshed, then the saved pipes are reconnected. This means that only the pipe added and pipe removed cases need to be checked.

4.5.6.1 Context's class invariant

The job of the context can now be thought of as preserving its class invariant. This invariant is to preserve the two constraints as discussed above. Adding a pipe should either be successful if possible, and the two constraints are preserved, or unsuccessful, in which case the pipe add fails and the offending new pipe is removed. Removing a pipe is always successful since the two constraints will always be preserved.

The `Context` contains a method `validateEntireContext()` which tests that the invarient holds. This method is run just before execution to help us catch bugs in the implementation.

4.5.7 Adding a pipe

When a pipe is added, if the pipe connects to an input node that has a parameterized type, there will be a need to update the binding stored for the parameter-name of that node. If this binding changes, this may in turn change an output type, and the change will be propagated throughout the pipeline, even possibly going around in several loops before terminating.

The main challenge therefore in the type algorithm is to keep the bindings up to date. One way of looking at this algorithm is as a data flow analysis problem. (Such as the data flow problems encountered in the control flow graphs of compilers).

The bindings for programs can be calculated using the algorithm shown in Figure 4.13

Figure 4.13 Data flow algorithm for keeping bindings up to date. This algorithm takes the same shape as most data flow algorithms.

```
for each Program p in programs
{
    // Set all bindings to the most flexible binding *
    // (no type constraint yet).
    p.bindings = {(T, *) | T ∈ p.parameterNames}
}

repeat
{
    p.bindings.T = getLUB( ⋃(q ∈ predNodes(T, p)){outputType( q )} )
}
Until bindings stop changing for all T and p
```

In the algorithm:

`predNodes(T, p)` will get all the output nodes that are connected to input nodes of program p that have parameter name T.

`outputType( q )` will return type T, if q is of type T, and it'll return the current binding for A, if q is of type A : T.

`getLUB( types )` will return the least upper bound of the types set ignoring * types. If that set is empty (or contains only the * type), then `getLUB( types )` returns *.

Note that this algorithm guarantees the most specific binding, as the type constraint that a parameter-name is bound to, is never lowered more than is needed to satisfy constraint 2. A binding is always set to the LUB of the connected output nodes as is required by this constraint. The bindings of programs will all gradually lower until constraint 2. is satisfied for all programs. Then the algorithm will terminate. At this point, constraint 1. needs to be checked for all pipes, and the detail that all bound types must be bound to a type more specialised than their upper bound of constraint 2. must be checked.

This algorithm has order of complexity that is based on the number of programs. However, since when a pipe is added, most of the time, only a small part of the pipeline will change, it is possible to write an algorithm for adding a pipe that propagates the changes recursively and therefore has an order of complexity directly proportional to the part of the pipeline that has changed. This would be useful if

Kevlar was ever expanded to have big pipelines (Such as scripts with many commands in one pipeline). We have chosen to have this propagation method for the type algorithm rather than the first method for this reason.

Another reason for using the algorithm we have chosen is that when the type checking algorithm finds a pipeline to be unsafe, it can return exactly which pipes will fail when you add a pipe. (Considering example 4, adding pipe 4 causes pipe 3 to fail.)

4.5.7.1 Outline of propagation based algorithm

- **1.** When a request comes in to add a pipe, that pipe is added. The recursive function `floodTypeCheck(newPipe)` is called. This method is explained from step 2 to 4. If the result is a failure, the pipe is removed completely and an error event is passed up.
- **2.** The pipe will connect an output node to an input node. It is checked that the types of these nodes satisfy constraint 1. If not a failed result is returned with this pipe specified as the problem pipe.
- **3.** If the input node of the program connected to the pipe is of type T, this is a base case, the algorithm terminates successfully.

  If the input node is of type A : T, we need to update the binding of A, to be the LUB of the union of the types of output nodes connected to input nodes of type A : T. This binding can only become more general since the set of types that are used to form the LUB will be the same as before, but will have one new type in the set caused by the new pipe. Therefore, the LUB can only be more generalised than before. We can take a shortcut when calculating the LUB. The LUB of the connected output nodes will be equal to the LUB of { CurrentBinding(A), newOutputNodeType } where the newOutputNodeType is the type of the new output node that the pipe is connected from. If the binding of A is less than T, we return an unsuccessful type check result.
- **4.** Now the new binding for parameter-name A has been calculated, we have to see if it has changed since the old binding. If it has not, this is a base case, the algorithm returns successfully.

  Otherwise, the algorithm gets all output nodes that have parameter-name A, and the algorithm recurses on any pipes connected to those nodes which takes us back to step 2. Each of these recursive calls will return either a successful type check result, or an unsuccessful type check result. If they are all successful, a successful type check result is returned. If any one result is unsuccessful, an unsuccessful result is returned with the union of problem pipes of all unsuccessful recursions used to generate the problem pipe set.

Since the algorithm may fail, we need a way to undo all the bindings caused by the propagation algorithm. To solve this problem, all binding changes are done to a temporary map, and these changes are only applied once the type check algorithm has verified that adding the pipe is safe.

4.5.7.2 Algorithm termination

Although the algorithm may propagate around in a loop on a pipeline for several loops, the algorithm is guaranteed to terminate. This is because, each time the binding of a program is updated, it is guaranteed the binding will become more generalised. Since the type tree will be of finite height, the maximum amount of times the algorithm can update a binding is the height of the type tree.

4.5.8 Removing a pipe

Removing a pipe can also be implemented by running the data flow algorithm to calculate the optimal type bindings for parameter names. However, for the same reasons as before, we chose to use a propagation based algorithm.

The algorithm is very similar to adding a pipe, except this time, there is no need to check for constraint 1. as the constraint is guaranteed to hold. This is because if you remove a pipe from a type-safe pipeline, the resulting pipeline is also type-safe. Note that for removing programs, to solve the problems that

loop backs will preserve their generalised types due to positive feedback, all bound parameters that are reachable from the removed pipes have to first be set back to *. The least upper bounds can then be flood updated as per the add pipe routine.

4.5.8.1 Removing pipes leads to type-safe pipelines

We know that before the pipe is removed, constraint 1. and constraint 2. hold. Once the pipe is removed, there is no longer a need to satisfy constraint 1. for that pipe. The input the pipe is connected to may have its type binding for its parameter-name changed, but this binding can only become more specialised or become the * type. This is because the set of types used to calculate the least upper bound has decreased by one element.

Therefore, output nodes of the same program will only have more specialised output types than before. We can guarantee that constraint 1. will not be effected on those pipes since the source type will be more specialised than before. In turn, the output types will propagate to the next program. If the next program's input is a parameterized type, the binding of that parameter can only become more specialised as the type of one of the inputs has become more specialised. This reasoning now applies inductively. The inductive variant is the fact that bindings always increase towards the top of the tree and cannot go past the stage where all parameter-name bindings are *.

4.5.9 Summary

The type checking part of Kevlar is essential to ensuring the stability of execution, yet most of the time, unless the user makes a mistake, they will never even know it is there. Yet every time they use Junction, or one of the other 7 of 18 programs in Kevlar that use the parameterized type system, they are relying on the type checking algorithm to correctly handle the type issues so that the programs behave as a user imagines them to behave like.

4.6 Program Execution

One of the core features of any shell is the ability to actually execute the pipelines (or single programs) that have been specified by the user.

4.6.1 Overview

As described in Section 3.3, it was decided that the execution of programs would happen in JVM's that are apart from the one in which Kevlar runs. Part of the execution of programs therefore required communication between the child processes and the Kevlar shell in order to pass the piped data between children.

There are three main sub-components that allow for Program Execution; the pipes that allow data flow inside a JVM, the `IPipeManager` that manages the routing of data between pipes and processes, and `Abstract Program` and the `IExecutionEngine` that actually execute the child processes.

4.6.2 Pipes

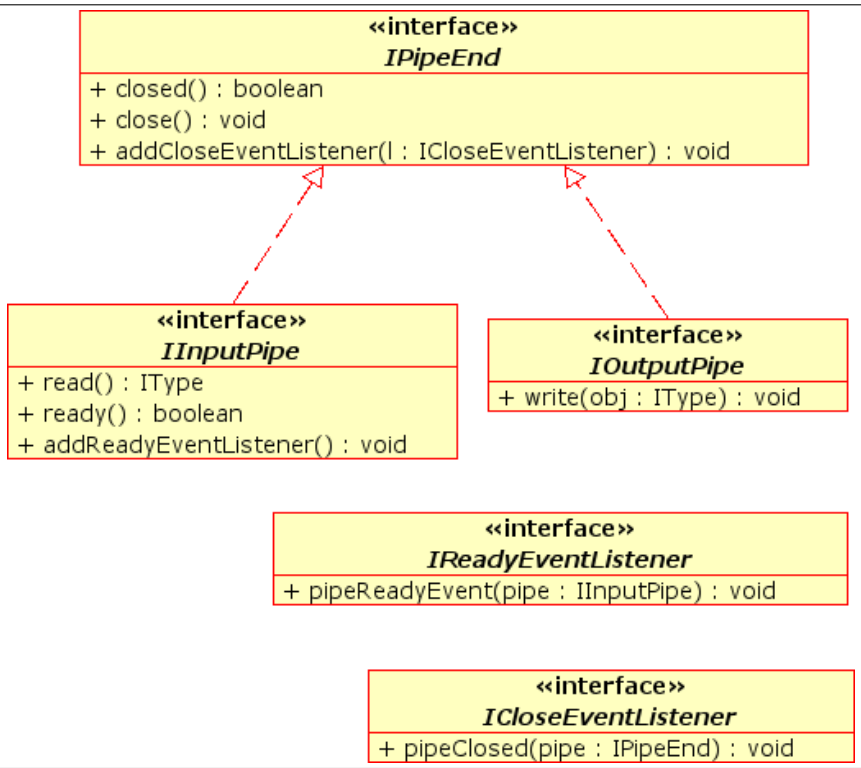
The Pipe API is what end developers would use to interact with their named input or output pipes as declared in their Program Descriptors. Although the programmer believes their data is being sent 'out' to another program when writing to a pipe, it was decided to make Pipe's a simple set of classes that work inside a JVM. The actual proxying of pipes and sending data to other processes is handled by the `IPipeManager` (see Section 4.6.3).

4.6.2.1 Design Notes

The programmatic interface provided by pipes in Kevlar has been designed to be simple and powerful. As part of this, it was decided to make the pipe model support listeners on the pipes, so that the listeners are invoked when the pipe moves into a state of having data available to read, or to being closed.

In line with most other Java pipe API's it was recognised that the `read` should be synchronous, blocking if necessary to wait for input, and (to allow recovery from a long-time-blocking read method), that the `read` method should be able to be interrupted.

Figure 4.14 The Pipe Interfaces



4.6.2.2 Implementation Notes

The pipes where written so that they always come in pairs, every `IInputPipe` would have an associated `IOutputPipe`, whereby writing to the output pipe would make data available to the input pipe to read.

The implementation of the Pipe API was therefore based on a producer-consumer idiom, and instead of making the `IInputPipe` and `IOutputPipe` pipes completely separate entities, they were instead different views of the internal data queue that was shared by both the input and output sides.

This meant that internally the Kevlar system never creates an `IInputPipe` or an `IOutputPipe`, but instead instantiates a `ClientSidePipe` class, which has access methods to get at the input or output views of its internal queue. These views are modelled by Java's inner class paradigm.

During the initial implementation of this `ClientSidePipe`, when events occurred that the listeners where to be notified about, a new `Thread` was spawned for each listener. This was to help avoid potential deadlock scenarios and other unpredictable concurrency issues if there was substantial code in the called-back listeners method. However, as described in Section 4.6.3, it was necessary to change these callbacks to occur in the same thread as the one that caused the event, for efficiency. The solution to the potential deadlock scenarios is now contractual - the listeners are not to make calls that add or remove items from pipes as annotated in their javadoc. A breaking of this rule, however, would not endanger the Kevlar system, as no external developer's code ever gets run within Kevlar itself, however

it could deadlock a child process, but that would be similar to writing a process that crashes, which the Kevlar system can handle.

4.6.3 IPipeManager

As described above, the Pipe API allows for in-JVM piping of data. Although the end developer for our system would work only with the Pipe API, some additional work has to be done to take data written to pipes and send it across streams to other JVM's to be read from the respective pipes there.

4.6.3.1 Design Notes

The `IPipeManager` was designed to provide principally a single piece of functionality, which is best illustrated by its initial method signature:

```
void managePipes(
    InputStream in,
    OutputStream out,
    Map<String,IInputPipe> ins,
    Map<String,IOutputPipe> outs )
```

The aim was to take any available data from any of the named `IInputPipe`s, and pass it down the provided `InputStream`, and also to read any available data from the `OutputStream`, and transfer that across to the appropriate named `IOutputPipe`. The underlying assumption is that at the other end of the streams is another instance of the `IPipeManager`, so that some guarantees about protocol can be made.

During the design of the `IPipeManager`, it became evident that the data written to the pipes was not the only thing that would have to be sent across the streams. Other control data, such as the receipt of the data, and pipe-closure events must also be sent.

It was deliberately decided to make the `IPipeManager` refer to Java's simplest `InputStream` and `OutputStream` classes, as this means that the class could, if wanted, be used to manage pipes not over the standard in and out streams provided by programs, but any streams desired (e.g. across a network).

4.6.3.2 PipeManager Implementation Notes

The implementation of the `IPipeManager` interface was as a server service, with a singleton access method. This design style was chosen because the `PipeManager` would potentially be creating a large number of threads, and it would be hard to reason about a server service if there could be several of them in existence.

The large number of threads that are created by the `PipeManager` was due mainly to Java's `InputStream`s being non-selectable[4], this forced a new `Thread` to be created for every `InputStream` in the system. In order to optimise the thread creation and startup-times, a Java `Executor` (a new 1.5 API addition that provides a thread-pool) instance was used.

For the handling of the `IInputPipe`, as single thread was used to deal with all of them, as it was believed that the writing out of the data read from them to their respective `OutputStream`s would be non-blocking.

Although the Pipe API had an event model, it was decided not to use it in the `PipeManager` implementation as the potentially huge number of threads that could be fired off would be extremely inefficient. Instead, the fact that the `ready` method for the `IInputPipe`s was non-blocking meant that we could check for pipes that had input waiting without ever having to block the thread for a non-deterministic

[4]For a description of select see [MANSEL], Java does feature selectable streams in its newer packages, but these are available primarily for network sockets, and we were unable to find a way to convert the legacy `InputStream`s that are returned from Java's `Process` class into selectable streams.

amount of time. If the thread where to become blocked waiting for a particular input, then all other inputs would not be dealt with creating starvation of the system.

To keep data organised, an `IPipeGroup` was created. This held an `InputStream` and `OutputStream` and the `IInputPipe`s and `IOutputPipe`s being multiplexed across them. It also contained utility data and methods to keep track of pipes known to be closed, and track of outstanding receipts for data that has been sent being but not known to be received across the streams.

When `managePipes` is called, its argument are turned into a `IPipeGroup`, and the group is added to the `PipeManager`s internal set of groups to manage. A thread is then started to listen to the `InputStream` and the method returns.

The `PipeManager`'s internal thread then deals with the `IPipeGroup`s that are currently registered according to the principals of pseudo code given in Figure 4.15. This pseudo code does hide some of the details of implementation for example the dealing of receipts, the propagating of information about pipes that have been closed, and also exactly how the data is read from or written to the streams.

The actual communication across the streams was implemented the streams in Java's `ObjectInputStream` / `ObjectOutputStream`, and then creating a custom serialized set of classes (all implementing an interface called `IPacket`) that could be sent across the streams, and would contain either control information (pipe closed, receipt of data), or piped data and the name of the pipe to output it to at the other end.

One issue with the `ObjectInputStream`s was the issue of class-loading the classes that contained the definition of the types. To achieve this, a `ClassLoader` from the `ITypeLoader` implementation had to be used during the `resolveClass` method of the `ObjectInputStream`.

Figure 4.15 PipeManager main thread pseudo code.

```
while ( true ) {
    while ( no groups to deal with ) {
        sleep_until_there_are_groups();
    }

    foreach registered group g {

        foreach IInputPipe p in g that is known not to be closed {

            if( p.closed() ){
                register p as closed in g
            } else if ( p.ready() ){
                read one item from p and write it to g.OutputStream
            }

            if( all pipes in g are closed ) {
                remove g from groups;
            }

            if( any errors occurred on g.OutputStream ) {
                close all unclosed pipes in g;
                remove g from groups;
            }
        }
    }
}
```

4.6.3.3 PipeManager2 Implementation Notes

`PipeManager` was based upon a polling `while( true )` loop. Even though it was running in a

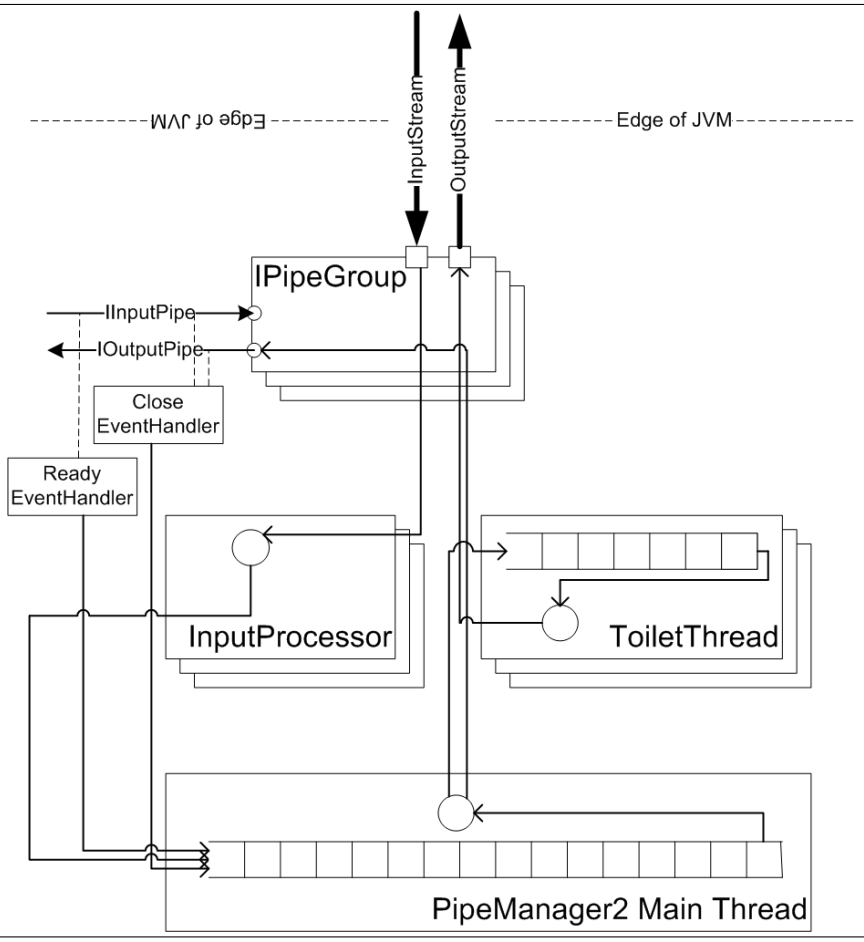
reduced priority thread, and `yield()` had been called to improve responsiveness, it was recognised that this code was leading to unacceptable performance when larger pipelines where being run, as this code is run in all child process JVM's as well as Kevlar itself.

To solve this problem, an event model for the `IInputPipe`s that did not create new threads was needed so as not to be inefficient for different reasons. The Pipe API was re-written so that event notifiers occurred in-thread (see Section 4.6.2.2 above). With this change, the `PipeManager` was re-written as `PipeManager2`. This time there is an internal queue of `PipeEvent`s where each pipe that becomes ready or closes registers an event on this queue. Also any "events" such as the writing of receipts are placed in this queue. The `PipeManager2` main thread then takes events off this queue (using Java's synchronization / `wait()` / `notify()` mechanisms) and processes them each in turn.

As in `PipeManager`, there is still a thread for each `InputStream`, but now these threads do no local processing, and for every data item read, produce an appropriate event and place it on the `PipeManager2` queue. Although this makes the queue a bottleneck, the performance is much improved over the polling version of the code, and it is much simpler and easier to reason about the concurrency this way.

It was realised, however, that writing to the `OutputStream`s could actually block. In `PipeManager2` this means that the event processing thread would halt running and all pipe communication would cease. To resolve this problem, a new thread had to be created to manage each `OutputStream`. Each thread has an internal queue of `IPacket`s waiting to be written, and uses synchronization primitives to take them in turn and write them down the stream, waiting if necessary for data to write.[5]

Figure 4.16 An overview of the process interactions in PipeManager2



[5]These threads are created and managed by a class affectionately named `ToiletManager`. This is because it is the `flush` method of `OutputStream` that was discovered to block.

4.6.4 Execution and AbstractProgram

4.6.4.1 Overview

As detailed in Section 4.3.4.2, execution is initiated by the calling of `run(...)` in `IPipelineProgram`. As part of the arguments to this run method, the named pipes are passed, as well as an instance of its argument model.

The `IPipelineProgram` delegates this call to its backing `IProgram` instance. In Section 4.2.1 we detailed that all programs in Kevlar, loaded and built-in, have an `IProgram` representation. Built-in programs define this method and spawn a new thread to execute in, directly reading / writing their pipes.

All loaded programs are given a default `run(...)` method that delegates to the `IExecutionEngine` to actually fork a new process and run the loaded program.

4.6.4.2 Design Notes

The design considerations to be solved when executing programs where:

- Executing the child in the correct environment, particularly the correct current working directory.
- Executing the child with a *CLASSPATH* that would allow it to find all necessary definitions for types that it relies upon, and the standard definitions for pipes, the argument model and `AbstractProgram`.
- Passing the child process the instantiation of its argument model, and its pipe mappings, so the `AbstractProgram` `run(...)` method can be invoked with the correct arguments.
- Providing a notification mechanism so that the layers above the framework can be informed when a single process terminates.
- Providing mechanisms to get the return values of the program (to be later looked up for its textual description in its description model).
- Providing a way to kill programs should the user so desire.

4.6.4.3 Implementation Notes

When writing the `IExecutionEngine`, the first problem was finding a way of managing the *current working directory* (hereafter abbreviated to CWD). Java does not feature an API call to programmatically change the CWD, so the `IExecutionEngine` instead maintains a `File` variable holding its value.

As the only way to change the CWD would be from within Kevlar itself (by making a call on the `IExecutionEngine`), this required a built-in program **ChangeDirectory** to be written to allow a user to change this CWD.[6] However, when the Visual Directory Tree was written, this variable is set to mirror the directory currently visible, and so the **Change-Directory** built-in became redundant.

In order to create the child processes, a Java `ProcessBuilder` is used. This has utility methods to set the desired CWD for the child process, and to set its environmental variables. To ensure the correct *CLASSPATH* exists for the child, the Kevlar *CLASSPATH* is appended with a path dynamically generated by the `ITypeLoader` (representing the path to all known valid type jars), and the path to the jar containing the program to be executed.

The end developer for our system implements the `AbstractProgram` class, and from their point of view, the `run(...)` method is their entry point. However the `IExecutionEngine` does not directly pass the end developers class to **java** to be invoked, instead `AbstractProgram` defines a main method and this is called with the target program to load as an argument to `AbstractProgram`.

`AbstractPrograms` `main(String[] args)` method is defined to set up communication with its Kevlar parent. It does this by first wrapping its standard in and out streams into Object streams, and then reading in an `ISetupInformation` instance from standard in, which the `IExecutionEngine`

[6] This is similar to existing shells, where environment altering programs such as **cd** or **chdir** are shell built-ins.

passes down to it. This setup information contains the arguments model instance, and details the names of the `IInputPipe`s and `IOutputPipe`s. `AbstractProgram` uses this information to manage them across standard out and standard in.

`AbstractProgram` finally uses Java's reflection API to create an instance of the desired class and invokes its `run(...)` method with the argument instance and pipe details that it expects.

Once a child process has been created by the `ProcessBuilder` object, the Java API gives back a `Process` object. This has methods to interrogate the process for its return value (if it has exited), to get hold of its standard input and output streams, and to kill the process. It also has a method to make the current thread wait until the process has exited.

Unfortunately it does not support a call-back or listener approach to process exiting events. The initial solution to this problem was to create a new thread and make that wait for the process to finish running, then when it has finished, call the listeners back. However instead of creating a new thread, an optimisation was chosen whereby the `InputProcessorThread` created in `PipeManager2` to handle the process's `InputStream` would wait for the process to exit after the `InputStream` was closed.

This exit notification mechanism, and other utility methods provided by the `Process` object (e.g. `kill` and `getReturnValue`) are wrapped into an `IProgramInstance` class, which is returned by the `IProgram.run(...)` method.

Chapter 5

Human Interface Abstraction Layer

5.1 Overview

5.1.1 Introduction

The Human Interface Abstraction Layer (HIAL) mediates between the GUI component and the Framework component and provides a number of services for the GUI to utilize. This chapter explains the motivation, design and implementation of the HIAL.

5.1.2 Motivations for building the HIAL

- **Keep the framework simple.** The HIAL provides the advanced features of pipeline construction and execution that the GUI component needs, using the primitive functionality that the framework component provides. This keeps the framework simple by factoring out functionality that can be derived. In this way, the HIAL is a mediator between the GUI and the framework.
- **Provide functionality not related to pipeline construction and execution.** The GUI makes use of many features not directly related to pipeline construction or execution. For example, searching for programs, saving and loading pipelines and more. The HIAL contains library functions to perform tasks that are unrelated to the framework. Having these features inside the HIAL ensures that the HIAL and framework completely contain the code that is reusable if another GUI were to be made.

5.2 Program discovery

5.2.1 Overview

In a traditional shell, there is often no simple way to find out what program to use to perform a particular task. To address this common complaint and to meet the specification, we decided that the system needs to provide a means to search for programs and look-up programs by category. The HIAL provides three features for obtaining programs.

5.2.2 Design

METHODS TO DISCOVER PROGRAMS

- **Keyword based search.** An advanced keyword based search is provided. The input is a set of keywords, and the output is a list of programs in order of match liklihood.

- **Category based lookup.** The HIAL provides a tree structure which contains the programs sorted into categories. The programs specify their own initial categorisation. The design was to allow users to then customize the categorization in the GUI so that the HIAL can save the changes to persistant storage. However due to time constraints, this feature wasn't implemented. An example of a category would be */Images/Format-Converting*.
- **Unstructured lookup.** An unstructured list of all programs for name based lookup

5.2.3 Keyword search implementation

Programs export a set of keywords that a user might search for as part of their meta-data. If a program exports the keyword "image", and a user searches for "Images", clearly it is useful if the system will still match the keyword. One way to solve this is to normalize words so that the last "s" is removed. But this will not solve the problem in the general case. For example, if the user types "directories" but a program exports the keyword "Directory". Similairly, we would like to catch spelling mistakes such as "Directrees".

To solve this problem in the general case, a sequence alignment algorithm is needed. This allows words to be compared for similarity, so that highly similar results can be treated as matches. For this task, the Needleman-Wunsch algorithm is suitable.

The Needleman-Wunsch algorithm works by taking a scoring scheme that scores an alignment of two strings. The algorithm can then find the optimal alignment and the corrosponding optimal score. This score can then be used as a measure of how similar the words are to each other. A match would be when a search keyword aligns with a program keyword and has a score above a threshold.

Figure 5.1 shows the search component in use.

Figure 5.1 An example keyword based search. The user accidently types in splitt instead of split. The search algorithm locates two matches. Junction is a program for splitting pipes so has split as a keyword. The user can now click each program to get a complete set of information about the program.

IMPLEMENTATION ISSUES

- **Setting the score weights.** Since the algorithm relies on a scoring matrix that could have any scores set to it, these need to be configured. To achieve this, we setup an experiment with a predefined set of search keywords and program keywords. The weights and thresholds were then adjusted in a loop in order to find the best set of numbers.
- **High complexity.** The Needleman-Wunsch algorithm is itself *O(n * m)* on the length of the keywords *n* and *m*. Since *n* and *m* are consistantly small (max ˜10), this execution time of one align-

ment is small. However, since a user will search for k keywords, and there can be p programs, with an average of l keywords each, the overall complexity is high. k is likely to be small, l might be typically be less than 15, but p can grow without bound.

To assess the impact of the high complexity, an experiment was run to find out how long the Needleman-Wunsch algorithm would take to execute for 1,000,000 iterations with long keywords on a modern machine. We found the time to be acceptable (˜3 seconds) so have not taken any measures to reduce the complexity. This would allow 10,000 programs with an average of 15 keywords each to be compared against 6 search keywords. One possible course of action would be to guarantee a maximum of 1,000,000 iterations of the algorithm by taking the first l' keywords from programs. The rest of the keywords could be compared using normal string comparision. This would require programs to list keywords in priority order.

5.3 Construction of pipelines

5.3.1 Overview

The HIAL implements many pipeline construction features as direct calls to the framework. Other features in the HIAL are built from the framework primitives.

Examples of derived features include presenting the different views of program arguments and IO nodes that the GUI needs to use and implementing a remove all programs feature in terms of removing each program individually.

Similairly, the HIAL adds intellegence to the calls to the framework, by performing state checks to verify the GUI is in the correct state for the method it called.

5.3.2 Execution

The GUI requires two things to happen implicitly for executing pipelines

- In a pipeline with only one program, which has a single output node, it should be as though a "show" program (output visualization program) is implicitly connected to that output. The intention is that this will speed up the creation of the many small one program pipelines that are needed regularly in a shell, such as to list the contents of the current directory.
- For every program that has an output node called "Error", it should be as though a "show" program is implicitly connected to that output. The intention is that this will ensure users get error messages from programs.

The HIAL provides these features by automatically creating the needed show programs and connecting their input pipes at the start of an execution. All messages about these programs are then filtered out so they are not received by the GUI. To the GUI, they are invisible.

5.4 Saving and loading

5.4.1 Design and implementation

It is not possible to completly factor saving and loading functionality out of the GUI as only the GUI knows details such as the canvas co-ordinates of programs that need to be saved. However, clearly, saving and loading a pipeline in general, is functionality of the pipeline model and the GUI should not be responsible for impleementing it. To solve this issue, the strategy pattern is used. The GUI provides a callback object that generates data that needs to be saved with each program and pipe. The HIAL then integrates this data into the save file.

Given this design, we decided that XML would make an excellent format for the saved information. This means that the GUI can specify the XML it wishes to save and it can be cleanly nested inside a set

of GUI specific data tags within the saved file. Also, because of the way XML validation works, it is possible to validate the HIAL generated part of the XML without knowing exactly what XML the GUI will provide. This is because XML XSD schemas comes with an <any> data type that allows any XML data to be included in specific tags.

XML DOM is used for the implementation of the XML parser. XML DOM is ideal because it allows an XSD schema to be specified to use to validate the XML, and also because it makes the job of requesting the service of the GUI to load its part of the XML file simple - an XMLDOMElement object representing the GUI information is simply passed to the GUI. There is a DTD and auto-generated XSD schema for the XML save format. An example save file snippet is presented below.

5.4.2 Example save file extract

```
<context>
<!-- This save file represents a pipeline to rename all files in ./web from *.htm to *
<gui-context>

<!-- The thumbnail image for the pipeline is cached here -->
<thumb>FFD8FFE...<!-- cut for berevity -->...735368408FFD9<thumb>

<macros>
   <!-- The GUI saves macros here -->
</macros>
</gui-context>

<!-- The program listing -->
<programs>

<!-- The program to list files like UNIX 'ls' -->
<program>

  <!-- The full package name of the program -->
  <name>uk.ac.ic.doc.kevlar.core.programs.fileUtils.ListFiles</name>

  <!-- Id used for pairing up with pipes -->
  <id>2</id>

  <!-- Specifies the current list of arguments for the program -->
  <arg-instance>
    <pair>
      <name>Paths</name>
      <value>./web</value>
    </pair>
    <pair>
      <name>Pattern</name>
      <value>*.htm</value>
    </pair>
  </arg-instance>

  <gui-program>
    <layout>
      <xposition>40</xposition>
      <yposition>40</yposition>
      <width>150</width>
      <height>71</height>
    </layout>
  </gui-program>
```

```
</program>

<!-- Rest of the programs omitted for brevity -->
</programs>

<!-- The pipe listing -->
<pipes>

<!-- A pipe from program 2 (ListFiles) to program 3 (Junction)
<pipe>

  <!-- From program to node -->
  <from>
    <program>2</program>
    <node>Filenames</node>
  </from>

  <!-- To program and to node
  <to>
    <program>3</program>
    <node>A-in</node>
  </to>

  <gui-pipe>
    <!-- GUI determined pipe specific information -->
  </gui-pipe>
</pipe>

<!-- Rest of pipes omitted for brevity -->

</pipes>
</context>
```

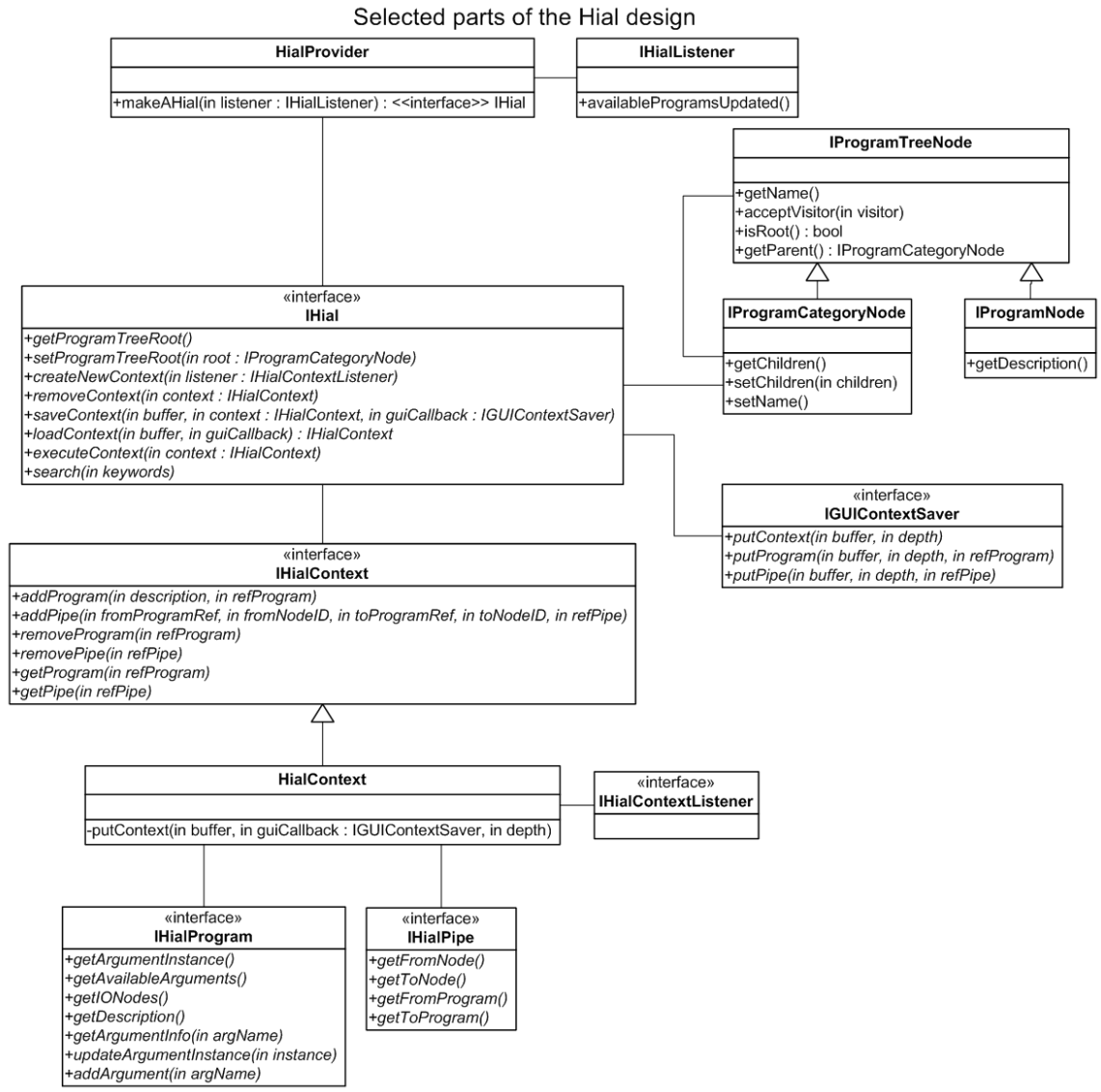
5.5 Class design

Figure 5.2 shows the class design of the HIAL.

FEATURES VIEWABLE IN THE CLASS DIAGRAM

- **Exploring program category structure.** Calling `getProgramTreeRoot()` returns the root of the program category tree. This can then be navigated with calls to `getChildren()` to find more categories, and programs within those categories. If the GUI modifies the category tree, it can update changes with a call to `setProgramTreeRoot(...)`.
- **Searching for programs based on keywords.** `List<IProgramSearchResult> search(Collection<String> keywords)` is used to perform a keyword based search. The returned list contains the search results in order of match quality.
- **Construction and execution of pipelines.** Pipelines can be constructed by first making a context with `createNewContext(...)` and then populating the context with calls to `addProgram(...)` and `addPipe(...)`. All constuction changes are reported back to an observer that implements `IHialContextListener`. Execution is then performed with a call to `executeContext(...)`.
- **Saving and loading of pipelines.** Saving and loading is done via calls to `saveContext(...)` and `loadContext(...)` respectivly. Both these methods take callback objects that allow the GUI to supply and receive co-ordinate data for each program and pipe. `saveContext(...)` takes a callback object that implements `IGUIContextSaver`. This is like the strategy design pattern.

Figure 5.2 Selected parts of the HIAL design



Chapter 6

Graphical User Interface

6.1 Drawing Engine

One of the first challenges we faced when designing Kevlar was to come up with a drawing engine that would allow us to display pipelines to the user.

6.1.1 Specification

The drawing system addresses the requirement that the system should provide a graphical representation of pipelines that is easy to understand. It is also a fundamental part of the user interface, guiding the user through the actions required to construct pipelines that perform the required tasks.

6.1.2 Overview

6.1.2.1 Design

We realised early on in the project that existing widget toolkits such as Swing and SWT would not be sufficient on their own, since they do not contain components designed for building pipelines. Instead we decided to take the existing SWT toolkit and extend it by writing our own components and systems for handling keyboard and mouse input (see Sections x and y).

The existing SWT drawing system has a set of predefined widget classes that map directly to operating-system-dependent widgets such as buttons and text boxes. Due to its use of operating system resources, SWT interfaces tend to provide very fast response times, however the existing widgets cannot be extended to provide additional functionality. Instead, the SWT `Canvas` class is intended to be subclassed and provides methods of accessing all the keyboard and mouse events required for creating a custom widget.

The SWT API documentation [SWTDOC] recommended subclassing the `Canvas` class for producing each non-standard widget, such as our representations of programs, arguments, input and output nodes and pipes. However, we noticed a number of problems with this in the design stage of our project. To begin with, we wanted the freedom to use transparency in our components, so that we could use irregularly-shaped widgets, such as pipes, without having a bounding-box around them. However, with each component in a separate `Canvas`, this would have to be simulated by first copying the background across before drawing the widget itself. Another problem is that instances of `Canvas` use system resources, which are not freed until that instance is explicitly disposed. This would require us to ensure that all GUI objects were explicitly disposed when no longer needed, which is different to Java's garbage-collected memory system and could result in memory leaks.

To avoid these problems, we decided to build a custom drawing system, using SWT's `Canvas` as a base. We wanted a hierarchical organisation of components, such that each object maintains references to a number of children and all except the root have a reference to their parent. The root component

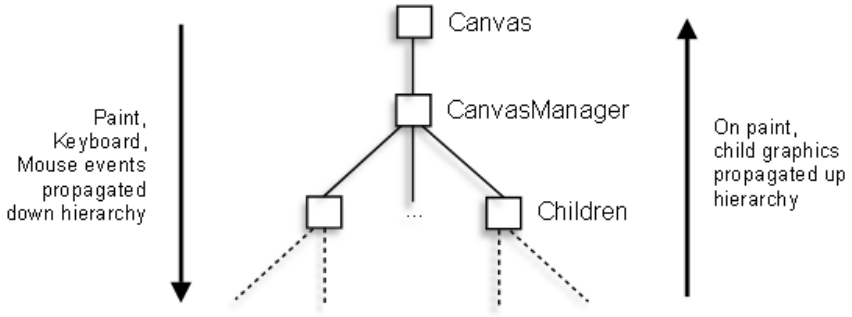
maintains a reference to the SWT `Canvas`, and receives paint events from it. It then propagates these events to its children, who draw onto a buffer, which is then copied across to the `Canvas` as the last stage of the paint process. By having this system of children drawing on top of their parent's buffer, transparency effects can easily be achieved, and using only one SWT `Canvas` instance makes resource management much more simple.

6.1.2.2 Implementation

The component hierarchy was implemented by creating an `ADrawableObject` class which maintains a `parent` reference to an ADrawableObject (which is null if it's the root), and a `children` list of ADrawableObjects for its sub-components. `ADrawableObject` also contains the final method `paint(GC gc, Rectangle redrawArea)`, which is called when a paint event occurs for this component. This method calls the abstract methods `drawBefore(..)` and `drawAfter(..)`, which are overridden by subclasses to actually draw the component, and then calls the `paint(..)` method in all its children.

The root was implemented by a `CanvasManager` class, which subclasses `ADrawableObject`, and creates the `Canvas` instance on startup and disposes of it on shutdown. It registers as a `PaintListener`, of this `Canvas`, and is such notified of any paint events. When it receives these events, it simply calls its own `paint(..)` method to propagate the events to its children.

Figure 6.1 . A brief overview of the component hierarchy in the drawing system.



6.1.3 Component Hierarchy

Our design included a number of important features of the component hierarchy that would allow us to implement a set of efficient and reusable widgets.

6.1.3.1 Design: Relative Coordinate System

To begin with, we wanted components to be able to draw objects and position children relative to their own coordinate space, not to their absolute positions on the screen. This allows components to be grouped, by making them children of a single super component, and then moved together, since moving the super component would also move its children.

6.1.3.2 Implementation: Relative Coordinate System

To achieve this, each `ADrawableObject` maintains an `area` rectangle which specifies its position relative to its parent as well as its width and height. When a component is drawn, it is drawn onto an image that is the size of this rectangle, and this is then copied onto the parent's image. In this way, a component can perform drawing operations local to its own coordinate space, and the results then get transformed to their parent's coordinate space when the image is copied across.

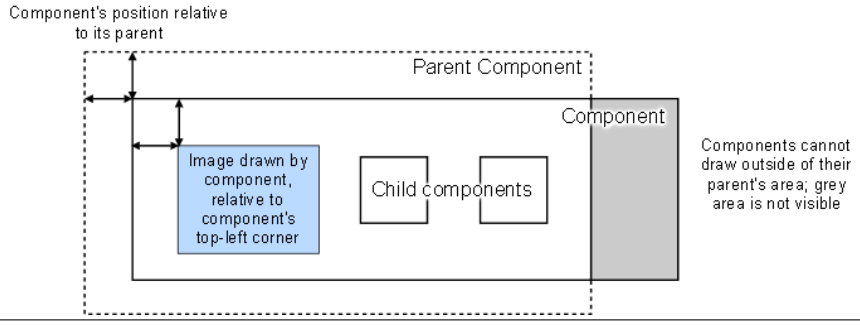
6.1.3.3 Design: Component Visibility

We also decided to restrict components so that they could only be visible within their parent's boundary, so that widgets do not have free roam to draw anywhere to the screen. This makes it easier to write windowed components such as scrolling boxes, since a component can be set to the size of the window and act as a clipping region for its children.

6.1.3.4 Implementation: Component Visibility

When a `ADrawableObject` instance's `paint(GC gc, Rectangle redrawArea)` method is called, it is passed a `GC` instance (Graphics Context) by its parent which represents the image on which it can draw itself. When this `GC` is constructed by the parent, its size is limited to be the size of the intersection of the parent's are and the child's area, so any of the child that lies outside of the parent's area will be drawn outside of the `GC` and therefore be ignored.

Figure 6.2 . Some of the features of our component hierarchy.



6.1.3.5 Design: Component Layout

Another important feature of a graphical interface is a component layout that caters for resizing. To cope with this, we opted to allow components to align and size themselves based upon their parent's dimensions. In addition to positioning components relative to the top-left corner of their parent, components can also be aligned to the right or bottom edges of their parent, and move as the parent is resized.

6.1.3.6 Implementation: Component Layout

A set of boolean flags in `ADrawableObject` represent each component's alignment and can be set by the method `setAlignment(boolean alignRight, boolean AlignBottom)`. A component's resizing behaviour is also represented by flags that can be set using `setSizeFillHorizontal(..)`, `setSizeFilLVertical(..)` and `setSizeFillBoth(..)`. Setting the component's position or size then sets the contents of an `alignedArea` variable, and an `updateDimensions()` is then called which translates this into coordinates relative to the parent's top-left corner, storing the results in the `area` variable, which is used for drawing. Since components can change their position and size when their parent's dimensions change, there is also a notify system, where a parent notifies its children of a change, so that the children can call their `updateDimensions()` method which recalculates their position and size relative to the parent's top-left corner.

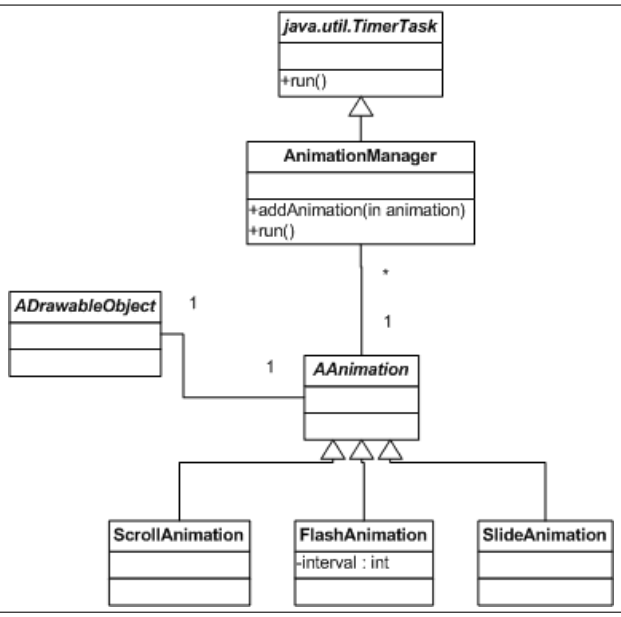
6.1.4 Animations

Although often viewed as a gimmick, we saw animations as an important tool to help users understand how the interface works. Often a smooth transition, such as when scrolling to a particular area of the pipeline, rather than a sudden jump prevents confusion and disorientation in the user's mind. We decided to use animations where possible in order to avoid this effects.

6.1.4.1 Design

We wanted to include an animation system that would run separate from the rest of the GUI and could be called to schedule animations of components. We also wanted a system that would allow different types of animations, such as a slide animation for moving a component between two points in a smooth movement and a flash animation, which periodically hides and shows a component. Therefore, the animation system was designed to maintain a list of animations, which themselves contain a reference to the component they are animating. At regular intervals, the animation system must call a method in each of the currently executing animations, which actually performs the animations and updates the animated component.

Figure 6.3 . UML diagram for animation system.



6.1.4.2 Implementation

An `AAnimation` abstract class was created, which takes an `ADrawableObject` in the constructor representing the component to be animated. The class also contains an abstract method `boolean animate()` which is implemented by subclasses to specify particular animation behaviour. The return value of this method signifies whether a call to this animation should be scheduled again, so must be true if the animation is to continue and false if it has been completed.

Several different animation classes were written to be used by components in the GUI. `SlideAnimation` takes a `Point position` in its constructor and smoothly moves the animated component from it's current position to this new position. `ScrollAnimation` extends this class to provide a similar movement for scrolling windows, so that the contents of the window can be scrolled smoothly to a particular position. `FlashAnimation` takes a `int interval` in its constructor and makes the animated component flash for this amount of time in milliseconds. This was used to provide the animation of the flashing caret for text boxes.

Controlling these animations is a singleton `AnimationManager` class which maintains a set of scheduled animations. It also maintains a single `Timer` instance, which fires an event every ten milliseconds, causing the `AnimationManager` to iterate through the executing animations and call their `animate()` method. These animations run asynchronously to the rest of the GUI, however they all run in sequence in a single animation thread which syncronises with the main GUI thread on redraws and component disposal.

With this implementation, it was very easy to add animations to components previously written without them. For instance, to get a component to move smoothly rather than jump when it's position is set, the `component.setPosition(Point position)` call must simply be replaced with a call

to `AnimationManager.getSingleton().addAnimation(new SlideAnimation(component, position))`.

6.1.5 Optimisation

One of the main hurdles we had to overcome with the GUI was to try to ensure that it would be fast and responsive even for large window sizes. To do this, we investigated a number of commonly used strategies for improving GUI efficiency.

6.1.5.1 Change Redraw

A simple optimisation of the drawing engine is to maintain as much of the drawn screen from one redraw to the next and only redraw the parts of the screen that have changed. To implement this, we first had to work out which parts of the screen change and need to be redrawn. This was done at the component level by providing a `redraw()` in the `ADrawableObject` class which schedules a redraw based on that component's area only. So, when a component changes its representation and needs to be redrawn, a simple call to `redraw()` will schedule a redraw for only the part of the screen that it currently occupies.

The next stage of implementation was ensuring that only the components that lie within the redraw area are drawn. This is performed when the `paint(GC gc, Rectangle redrawArea)` method of a component is called, with the `Rectangle redrawArea` parameter indicating the region to be redrawn relative to the parent's coordinates. The first thing a component does is to calculate the intersection between it's `area` and the `redrawArea`. If this intersection is empty then the component's `drawBefore(..)` and `drawAfter(..)` methods are not called, and the paint event is not propagated to its children (since children cannot draw outside of their parent's area, so their visible area must lie outside the redraw region also). If the intersection is not empty, then a new `GC` instance is constructed to be the size of the intersection and is passed to the `drawBefore(..)` and `drawAfter(..)` methods so that any drawing they perform outside of the `redrawArea` is ignored. Finally, the `redrawArea` is translated to be relative to the component's coordinates and passed, with the new `GC` instance, to it's children.

6.1.5.2 Redraw Aggregation

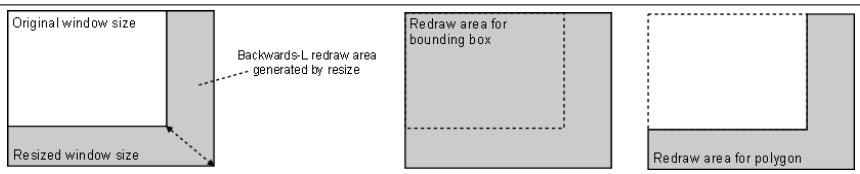
One of the most effective and commonly used GUI optimisations is the aggregation of consecutive redraw requests. In a graphical interface such as ours which can contain hundreds of components at a time, each with the ability to request a redraw, there is often a large degree of overdraw. This effect occurs when two overlapping components request a redraw one after the other, so that the overlapping area is drawn twice in quick succession. Since redraws always occur due to some event in the system, whether it be a mouse press or a timer tick, a form of aggregation can be used where all the redraws generated by a single event are only applied at the end of the event.

Redraw aggregation is implemented in our drawing system in the `CanvasManager` class. When a component calls its own `redraw()` method, a redraw doesn't actually occur, but instead it is scheduled in the `CanvasManager` and the area to be redrawn is logged. If another `redraw()` is called, the logged redraw area is simply extended to include the new area to be redrawn, such that a bounding box containing all the regions that need to be redrawn is built. Finally, when all handlers for an event have been called, the `CanvasManager` calls a `applyRedraw()` method which initiates the first `paint` method, passing it the bounding box as the area to be redrawn.

More complex methods exist to calculate the redraw aggregate, such as using a polygon rather than a rectangle to represent the redraw area, so that complex shapes can be redrawn without the overhead of redrawing the unchanged areas of their bounding box. An example of this is the commonly-known backwards-L effect that occurs when a window is resized. Instead of the redraw area being a small rectangle, it consists of two rectangles, one covering the bottom edge and one covering the right edge of the window. The resulting bounding box would cause the whole window to be redrawn, whereas a polygon could accurately represent the backwards-L shape saving a lot of unnecessary overhead. We

decided not to implement this technique in Kevlar's drawing engine, since we realised that the majority of redraws are the result of a single event and tend to occur with close proximity of one another (a mouse click, for instance, often only generates redraws around the clicked position). When the window is resized, the majority of our components would also need to be resized, since they are often aligned or sized based on the window's dimensions, so for Kevlar this technique would not eliminate the need for a full redraw.

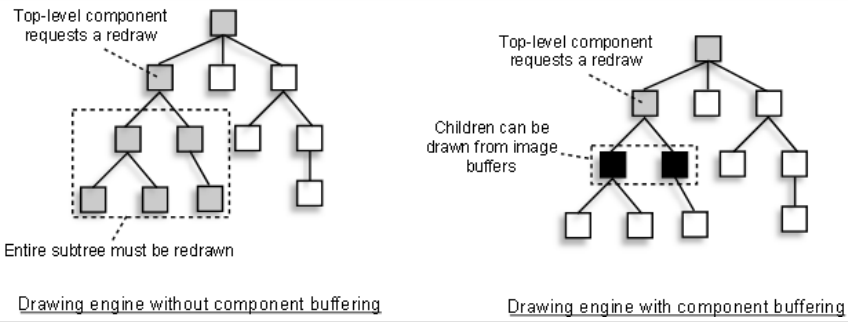
Figure 6.4 . Illustrating the backwards-L redraw area generated by resizing the window.



6.1.5.3 Component Image Buffering

One of the problems of a hierarchical component system is that when components near the top of the hierarchy need to be redrawn they must also redraw the whole component sub-tree that is beneath them. To avoid this, components can paint to a buffer which they hold on to between redraws. When the next redraw occurs, if the component and its children have not changed, it can simply copy their image from the buffer and do not have to redraw themselves or propagate the redraw event to their children. With this system, if a component near the top of the hierarchy requests a redraw, only its immediate children need to be redrawn, since their images will be buffered.

Figure 6.5 . Depth of redraw for non-component-buffered and component-buffered drawing engines.



We implemented this technique during Kevlar's development but chose to remove it from the release version, having identified a problem with the approach. One of our reasons for using only a single SWT `Canvas` for the whole drawing engine was to avoid the need to dispose of GUI components explicitly, however this problem is also encountered with SWT's `Image` class. Since each component would have to maintain it's own image buffer, these buffers would have to be disposed of when the image is no longer needed, otherwise the system's memory usage would continually increase. Also, having implemented the technique, we found that it did not offer a significant performance increase, which we attribute to the shallow component trees in the Kevlar GUI. Overall, we decided that the hassle and potential problems caused by using this technique outweighed the gains, so we removed it.

6.2 Widgets

Having designed our drawing engine, we needed to construct a set of widgets which we could use throughout our graphical interface to provide a consistent visual style.

6.2.1 Specification

This section doesn't cover any specific requirement set out in our specifications document, however it is important for ensuring the overall usability requirement of the system.

6.2.2 Roll-Over and Selectable Buttons

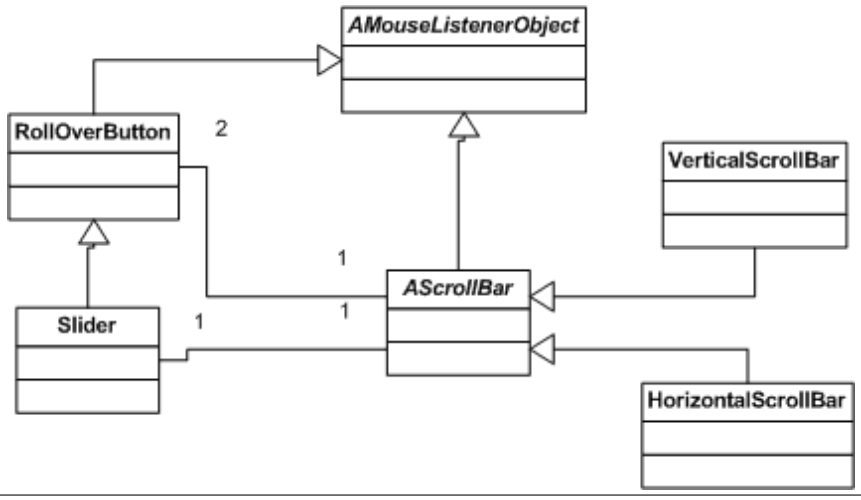
Since our interface contains a large number of non-standard widgets (programs, arguments, nodes and pipes, for instance), users may not immediately recognise how to interact with these widgets as they would with a standard GUI. Therefore, we decided to create widgets that would provide visual prompts for which widgets can be interacted with using the mouse, and how they behave.

The `RollOverButton` class can be used for representing more than just traditional GUI buttons. Instead it is a general class for representing widgets that should appear highlighted when the mouse moves over them, and performs some action when they are clicked. It simply takes three image parameters in its constructor; `normal`, `rollover` and `down`, and changes the button's visual appearance to match one of these. By overriding `AMouseListenerObject`'s `mouseEnter()` and `mouseExit()` methods, it gets notified when the mouse moves over the button and can change it's appearance to the `rollover` image. The `mousePress(..)` and `mouseRelease(..)` methods are also overridden so that while the mouse is pressed on the widget it's appearance changes to the `down` image to signify interaction.

We also wrote a `SelectableButton` class which extends `RollOverButton`. This is used to represent buttons that maintain some sort of state, such as a button that stays depressed after it's been clicked. This simply takes four parameters in its constructor; two `normal` and `rollover` images for each state, which it switches between when pressed. The current state of the button can be queried using the `boolean isSelected()` method.

6.2.3 Scroll Bars

Figure 6.6 . UML diagram of scroll bar implementation.

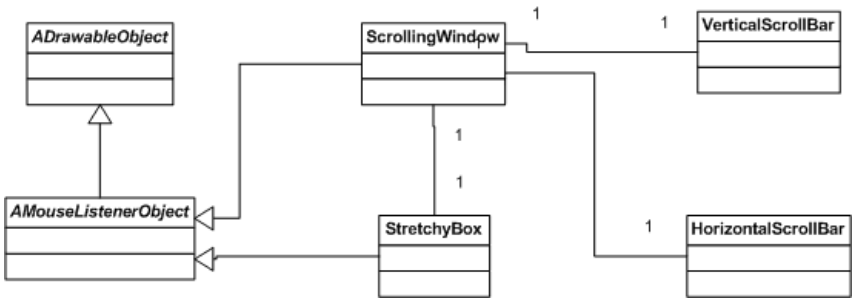


We originally intended to use the existing SWT widget set in our interface where possible, however we found that SWT does not allow its existing `ScrollBar` widget to be used separate from its scrollable widgets. Instead, the SWT documentation [SWTDOC] recommended using the `Slider` widget, but this looked very basic and did not provide the functionality that we desired. To correct this, we wrote our own abstract class `AScrollBar` which contains three `RollOverButton` instances; two for buttons that allow for fine movement and one for the scroll bar's slider, which can be dragged. This class is then extended by `HorizontalScrollBar` and `VerticalScrollBar` which display the scrollbar at the desired orientation.

6.2.4 Scrolling Windows

One of the most important widget in our interface is the scrolling window, since it is not only used for the main region on which users construct pipelines, but is also used for task panes and even list boxes. Given our design for the drawing engine, constructing a scrolling window widget was very simple. The `ScrollingWindow` class itself is simply a subclass of `ADrawableObject` which is sized to the desired dimensions of the window and acts as a clipping region. It constructs an instance of a `StretchyBox` widget, which is simply an `ADrawableObject` that resizes to accomodate its children. The `addChild(..)` method of the `ScrollingWindow` is overridden to add children to the `StretchyBox` instead, so that only the `StretchyBox` need be moved to scroll the contents of the window. Finally, instances of the `HorizontalScrollBar` and `VerticalScrollBar` classes are constructed if required, and the `ScrollingWindow` listens to events from these scroll bars and moves the `StretchyBox` as appropriate.

Figure 6.7 . UML diagram of scrolling window implementation.



One important feature of the `ScrollingWindow` class is its ability to automatically scroll to the currently selected object. This is implemented by a `scrollTo(ADrawableObject child)` method, which scrolls the window to make the specified child object visible if it is currenly outside the visible region. The `ScrollingWindow` first uses the child's `getAreaRelativeTo(ADrawableObject parent)` method to translate the child's local area to be relative to its coordinate system. It can then compare the child's area to it's own area to work out whether the child lies outside the visible region. If this is not the case, the `ScrollingWindow` calculates the difference in the X and Y values needed to shift the child into the visible region, and then subtracts this from the `StretchyBox`'s current position to calculate its new position such that the child is visible. It then schedules a `ScrollAnimation` to scroll the window smoothly to this new position. This effect can be seen when using the keyboard to tab between programs in the current pipeline.

6.2.5 SWT Widgets

Despite our original intention to use existing SWT widgets where possible, and only add custom widgets for representing non-standard controls such as programs, arguments and pipes, we encountered a number of problems integrating SWT widgets into our drawing engine.

6.2.5.1 Relative Coordinate System

The first problem is that SWT widgets must be associated as children of other SWT widgets in order to be positioned relatively. Since an optimisation of our drawing engine was to eliminate the need for a separate SWT `Canvas` for each widget, only one actual SWT widget exists in our system; the root `Canvas`. This meant that SWT widgets had to be added as children of this canvas and positioned absolutely. To work around this problem, an `SWTWidget` class was created to wrap SWT widgets. This class extends `ADrawableObject`, so can be included in the component hierarchy, and simply calculates the absolute position of the SWT control based on its current relative position (by using `ADrawableObject`'s `getAbsoluteArea()` method). Unfortunately, this method would only update the SWT widget's position when the component itself was moved, but one of the advantages of the component hierarchy is that moving a parent component causes all of its children to move with it. To produce this behaviour, we had to make the `SWTWidget` wrapper register itself as a listener to the movements of all of its parents.

This way, whenever one of its parents it moved, it can update the absolute position of the SWT widget accordingly.

6.2.5.2 Keyboard Focus

Since SWT widgets operate using a keyboard model separate to our own, we needed a way of redirecting all keyboard events to our `KeyboardManager` class. Unfortunately, SWT widgets tend to aquire keyboard focus for themselves at various points in the program (such as at creation time, and whenever the user clicks on them), diverting all keyboard input to themselves. We were unable to find a way of preventing this from happening while still maintaining the functionality of the wigets, so instead we had to make sure that they `KeyboardManager` was registered as a `KeyListener` for all SWT objects in the system to avoid it missing key events.

6.2.5.3 Overlapping Components

Although SWT widgets are represented by the `SWTWidget` instance which is part of the drawing engine's component hierarchy; this is merely a proxy object which contains a reference to the true SWT widget, which is a child of the root `Canvas`. Because of this, SWT widgets are always drawn on top of the `Canvas`, after all other components have been drawn. Therefore, there's no way of enforcing the rule that they must not be able to draw outside of their parent's area.

A solution to this problem was to include a method in the `SWTWidget` proxy which would detect if its area is visible or not and, if not, hide the SWT widget. Unfortunately, this did not cater for the case where objects in the component hierarchy partially overlap SWT widgets, in which case the SWT widgets would either have to be made completely invisible or appear on top of the overlapping object.

Although we were able to find ways of working around the majority of problems associated with using SWT widgets in our drawing engine, there were some which we were unable to avoid. We first developed Kevlar using these widgets, but at the end of development realised that there was enough time to fix these problems by developing our own set of widgets using similar interfaces. Once we'd written our own `TextBox`, `MultilineTextBox` and `SelectionBox` widgets to be completely compatible with the drawing engine, we were able to easily replace the SWT widgets we had used.

6.2.6 Swing Widgets

Above we outlined some of the problems that where found with using SWT Widgets in the Kevlar toolkit. Before the writing of our own text-based widgets, we first explored the possibility of wrapping existing swing controls and using them.

Most widgets in Java's swing toolkit (predominantly those that extend from `JComponent` are "lightweight". This means their rendering and state management code is all performed using Java classes and not native method calls. We believed we could make a subclass of `ADrawableObject` in the Kevlar toolkit, and proxy the events and rendering events to an aggregated swing widget. For example the swing widget would render on graphics for an image, and this would be then be drawn in `ADrawableObject`s `paint` method.

There were some challenges to overcome with this, firstly converting Kevlar's mouse and keyboard events to ones suitable for the `AWT` `MouseEvent` and `KeyEvent` classes. The second was the conversion of images to and from swing to swt. The former was overcome by careful studying of the API's, the latter by re-use of an existing helper library (see [IBMEX]).

For simple swing components (e.g. `JButton`), this conversion process worked fine, however it was discovered that the swing text-box-based components appeared to use a low-level way of interacting with text that we could not understand. As text-based controls were our motivation for wrapping swing components, it was decided to abandon this route of solving the problem when it became uncertain whether we could solve the problem, or how long it would take if we could.

6.3 Layout

The layout component focusses on the layout of programs on the canvas and on the routing of pipes between programs. The following sections give an overview of the design and implementation of the layout algorithms.

6.3.1 Specification

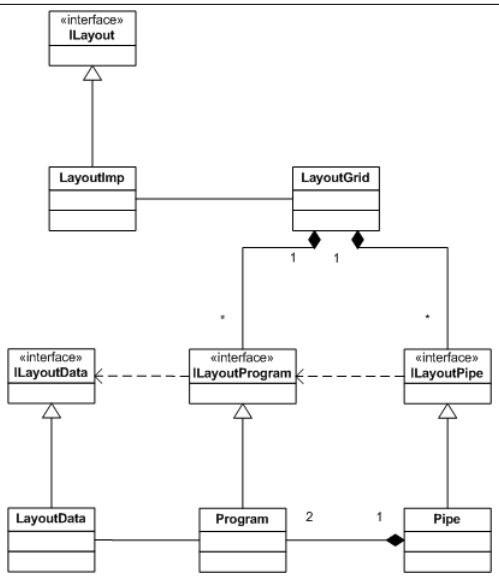
The construction of the pipeline on the canvas will be the user's main point of visual representation. It is therefore imporant that the programs and pipes are visually appealing and logical to understand. We found that there are three main points that the layout of the pipeline needs to fullfil in order to satify the user.

- No two programs in the pipeline should overlap The layout system should make sure that one of the overlapping program gets repositioned logically so that the overlap is avoided.
- No pipe should overlap or intersect with a program The layout system should efficiently reroute the pipes so that they do not cross programs.
- The layout system should try to minimize overlapping pipes This is enforced so that ambiguity of pipe routings are kept to a minimum, although steps must be taken to ensure that the performance of the overall system is not affected too much.

6.3.2 Implementation

The layout component is implemented by `LayoutImp` which is an implementation of `ILayout`. The most important methods of `ILayout` are `addProgram(...)` and `alterPipe(...)`. Whenever a program is added or moved on the canvas, the `PipeLineManager` obtains a reference to the current layout implementing the `ILayout` interface from the `LayoutManager` and invokes the `addProgram(..)` or the `suggestProgramPosition(..)` method. Additionally, if any pipes are connected to the program, the `alterPipe(..)` is called for every pipe that needs to be updated, and a reference to the pipe concerned is passed.

Figure 6.8 Layout Components. Overview of the Layout Components in UML.



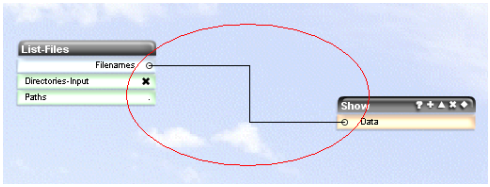
The layout's implementation of `addProgram(...)` makes sure that the program passed as the parameter does not overlap with any of the other programs already positioned on the canvas. In order to do this, another component called `LayoutGrid` has been designed. `LayoutGrid` is a datastructure that holds

all the layout information of programs and pipes on the canvas. The `LayoutImp` will infact simply delegate most of its methods to `LayoutGrid`. When a program is added for the first time, `LayoutGrid` will store its layout information, and whenever a program's layout is changed the `LayoutGrid` will also be informed. When a program is moved, `LayoutGrid` makes sure that it will not be placed in a location that overlaps other programs it is holding. When the user places the program so that it overlaps with other programs `LayoutGrid` will start to move the program to the nearest space large enough so that it does not overlap with other programs. The algorithm that takes care of this is initiated with the `checkMove()` method which is recursively called. If the program is overlapping when `checkMove()` is called it is repositioned to avoid the program it is overlapping, and `checkMove()` is then called recursively in case this movement has caused another overlap, until no overlaps occur.

The `alterPipe(...)` method will make sure that the pipe added between two programs is not overlapping or crossing any programs held by `LayoutGrid`. The pipe on the canvas will be represented by `ILayoutPipe` which has the method `addPoint(...)` which simply adds a new connection point to the pipe. In order to route the pipe without overlapping, `LayoutGrid` delegates the call to a private method of `LayoutGrid` called `alterPipeAux(...)`. `alterPipeAux(...)` is the entry point of the pipe routing algorithm which is also recursive. The following description contains a brief overview of the routing algorithm:

- **Base case.** When the start of a pipe is to the left of the end point, ie. from left to right, and when the pipe does not overlap a program on it path, then the algorith will apply the "Z pattern". By this we mean that the pipe will go right till the horizontal middle of the two programs it connects, then it will go down or up till it is on the same y-coordinate as its end point and finally continue right till it reaches the end point. A picture of this situation, Figure 6.9, is shown below for clarification.

Figure 6.9 Base Case. This shows the base case of the algorithm forming a "Z pattern". The base case is highlighted.

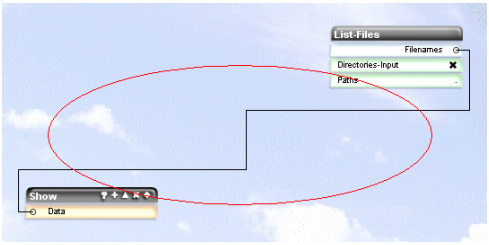


- **Recursive Case 1.** When the pipe tries to form the "Z pattern" but on its path it overlaps a program on the canvas, the layout algorithm will end the path just before it overlaps that program. The algorithm will then move the path so that it has an open way to the end without having to hit the program it just overlapped again. The algorithm then recursively calls the `alterPipeAux(...)` again so that it forms a path between the current point and final point. A picture of this situation, Figure 6.10, is shown below for clarification.

Figure 6.10 Recursive Case 1. This shows the first recursive case of the algorithm.

- **Recursive Case 2.** When a pipe's starting point is to the right of its end point, the algorithm will first move the starting and end points away so that they can be connected with a "normal" pipe. When we say normal, we mean that the path can again go from left to right instead of right to left. To form the path, now from end to start point, the algorithm simply calls the `alterPipeAux(...)` method again. A picture of this situation, Figure 6.11, is shown below for clarification.
- **Minimizing Pipe overlaps.** Finally we wanted to minimize pipe overlaps without reducing the performance too much. We decided that it would not be efficient to check for pipe overlaps. Instead we found out that pipe overlaps usually occured because the same algorithm was applied and thus same routes were taken by pipes. We then decided that the best way to reduce overlaps with minimum overhead was to add a random spacing to the pipes, so that different pipes would

Figure 6.11 Recursive Case 2. This shows the second recursive case of the algorithm. The base case is highlighted.



take slightly different routes. Note that this solution does not totally reduce overlap, instead it minimizes it.

6.4 Keyboard Events

Although the main aim of Kevlar is to provide a shell that is easy to understand and use, we realised that it would also have to be fast to use in order to attract users of existing shells. To achieve this, we needed a comprehensive set of keyboard shortcuts that would allow an experienced user to access all of Kevlar's features as quickly as possible.

6.4.1 Specification

Keyboard events address the requirement that users should be able to construct pipelines using the keyboard. Our main goal for this section was to provide all the functionality that is accessible using the mouse through keyboard shortcuts. Additionally, we aimed to keep the number of key presses required to construct pipelines to a minimum.

6.4.2 Key Binding Model

6.4.2.1 Design

We wanted to create a flexible key binding model that would allow us to easily alter which keys perform which actions so that we could test various configurations and evaluate them to find which work the best. To achieve this, rather than make each component of the system responsible for handling its own key presses, we opted for a single, centralised class that would contain a map of all the keys and actions supported by the system. To change the key bindings, we would simply have to change map, and this system would allow us to easily make key bindings user-customisable in the future.

We realised early on that Kevlar required far too many key bindings to simply map every key to a different action. Instead, it was more intuitive to provide a state-based system, such that keys do different actions in different states. Since states add an extra level of complexity to the keyboard model, we had to make sure that users have visual aids to remind them which state they are in. For example, if a program is highlighted then pressing certain keys will perform actions upon that one program, whereas if a multi-select box is visible then the actions are performed upon all programs in the selection.

Finally, the key binding model was designed to include default actions, which would be executed when no specific key binding could be found. This allows us to specify certain key bindings that will execute in any state, as long as a more specific binding doesn't already exist for that state. An example of this is the CTRL+ENTER binding for executing a pipeline, which can be performed during any state of the system.

6.4.2.2 Implementation

A `KeyBindings` class was written, which maintains a four-level map representing the events, states, modifiers and keys that map to the actions in Kevlar. It has static methods `void addBinding(Item events, Item states, Item modifiers, Item keys, IAction action)` and `IAction getBinding(Item event, Item stateID, Item modifier, Item key)` for setting and getting the actions respectively. The `Item` class contains a single keycode or state integer and has a subclass, `Group`, which can be used to specify a set of keycodes or states. This allows us to associate an action with a range of different state, modifier and key combinations in a single line.

The keyboard events themselves are received by the `KeyboardManager` singleton class, which extracts the event type (key pressed or released), the current modifiers and the key code. This information is passed along with the identifier of the current state to the `getBinding(..)` method, which traverses the three-level map and retrieves the action associated with the supplied combination (if any). It does this by first seeing if a direct mapping exists between the supplied key combination and an action and, if not, then it starts looking for default handlers. This process begins at the key level, where it tries to match the current keycode to any default alphabetic, numeric or symbolic handlers that may exist. If that fails, it then tries to match any default modifier handlers (handlers that do not specify which modifier keys need to be held down). Finally, it attempt to match any default state handlers, after which it fails and returns null (in which case no action is performed).

6.4.2.3 State-Based Key Handling

States are used in order to specify key bindings that are only active at certain times, such as when a text box has focus or the user is selecting options from a drop-down list. Each state has a class which extends the abstract `State` class, and must implement the abstract method `Item getStateItem()` which returns an `Item` instance containing a unique identifier for this state. States themselves are not directly called when any specific keyboard event occurs, however they can override `onEnter()` and `onExit()` methods to update various sections of the GUI when the system enters and exits that state. The main role of state classes is to contain any state-specific information that may be required for actions to be performed. For example, the `TextBoxState` contains a reference to the active text box and also a reference to the last active state, so that the last state can be restored when the user leaves the text box.

The singleton `KeyboardManager` class maintains a reference to the active state which can be accessed using its `State getState()` method. Any class in Kevlar can also set the current state by using it's `void setState(State s)` method.

6.4.2.4 Key Actions

Every action to be performed on a key press is implemented in its own class which must implement the `IAction` interface and provide implementation for the `perform(Item event, State s, KeyEvent e)` method. When actions are executed, they are provided with a reference to the current state, from which they can extract object references and invoke methods. It is also usual for actions to change the active state by calling the `KeyboardManager`'s `setState(State s)` method.

6.5 Mouse Events And Dragging

Mouse events and dragging play an important role in the usability of our project, since all of the features of the project are accesible through using the mouse.

6.5.1 Specification

Allowing the user to construct pipelines using the mouse was a minimum specification requirement of our project. The mouse event and dragging engine allows the user to construct pipelines by dragging programs onto the canvas and then clicking on input and output nodes to connect them together.

6.5.2 Overview

6.5.2.1 Design

The design of the mouse events and dragging was partly dictated by the design of the drawing engine. All our components are custom and are organised in a tree hierarchy with children and parents. Since the SWT widget we used to draw the components on is the `Canvas`, it is the only SWT widget we could rely on to listen to various mouse events. Therefore, we had to organise the components that are capable of receiving and dealing with mouse events into a hierarchy similar to that of a drawing engine.

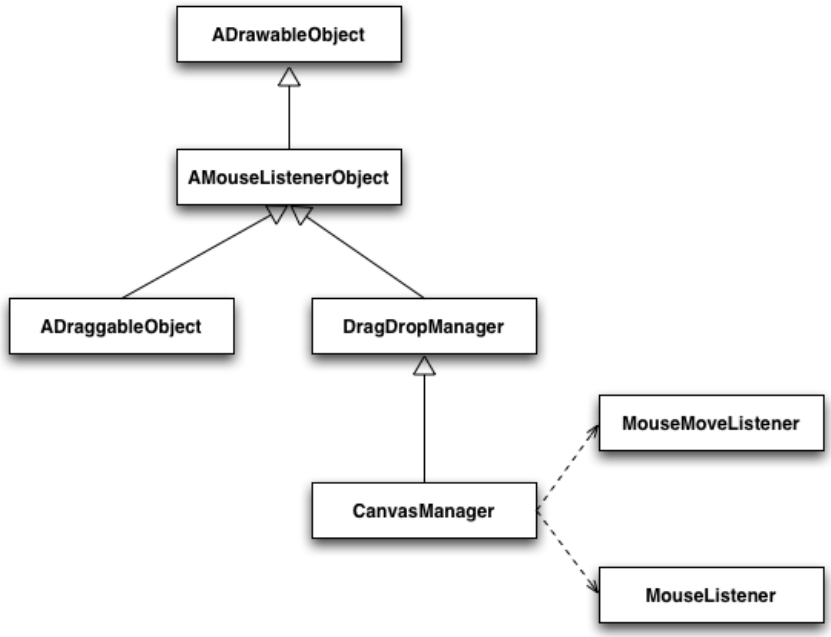
The general idea is that the canvas receives the mouse events then calls methods responsible for distribution of mouse events to determine which component should deal with the event that has occurred.

We have also customized mouse focus events in our GUI. Instead of an object acquiring focus for normal response to mouse events it only needs to get it if it wants access to mouse events that occur outside of its own area. So, widgets can call `getMouseFocus()` and they'll then get mouse events from anywhere on the GUI. They can then call `releaseMouseFocus()` when they want to revert to only receiving mouse events inside their area. This is used in things like the scrollbar slider so that when the user is dragging the slider, the mouse doesn't have to actually be over the slider widget for it to move.

6.5.2.2 Implementation

The overall structure of the mouse events listeners is as follows:

Figure 6.12 . An overview of the mouse event listening engine.



Each component that receives and deals with mouse events subclasses either AMouseListenerObject or ADraggableObject, depending on whether the component is draggable or not.

We have used `CanvasManager` class as a base in propagating the mouse event to the correct component, since `CanvasManager` contains the `Canvas` SWT widget. `CanvasManager` implements the `MouseListener` interface, in order to listen to events generated by mouse buttons being pressed or released, and the `MouseMoveListener` interface to listen to events generated by the movement of the mouse. In addition, the `CanvasManager` class extends the `DragDropManager` class, which is a subclass of `AMouseListenerObject`.

When a mouse event occurs, the `CanvasManager` calls the corresponding concrete methods implemented in AMouseListenerObject to allow the event to be passed onto a component that will respond to it. In order to be able to distribute mouse events to the right component `AMouseListenerObject`

uses the list maintained by ADrawableObject which contains all objects that are drawn on the canvas, i.e. its children. `AMouseListenerObject` then goes through that list checking whether the position of the mouse is within the boundaries of any of its children. If it is, `AMouseListenerObject` passes the event to the child by calling methods corresponding to the occurred mouse event in the child.

6.5.2.3 AMouseListenerObject class

AMouseListener class contains methods used to distribute the mouse events to the right component, such as `concreteMouseUp(..)`, `concreteMouseMove(..)`. Methods that actually deal with the occurred events are declared as abstract, so are implemented in classes corresponding to the component that the event has been distributed to.

One other feature of the design and implementation of mouse events in our GUI is the existence of `mouseEnter(..)` and `mouseExit(..)` methods. These methods are called to notify the component that the mouse pointer has crossed the boundaries of the component. They are used for rollover buttons and components that change their appearance when a mouse pointer is being over them.

6.5.2.4 Drag and Drop

Drag and Drop couldn't be implemented using SWT library available for drag and drop. This is because most of our components are customary and could not implement SWT standard interfaces.

We implemented dragging by registering a 'drag-and-drop' component with the `DragDropManager` class. `DragDropManager` follows a singleton design pattern, since there can only be one object being dragged at any time. This class coordinates all dragging activity and makes dragging visible by repositioning the object and calling the `CanvasManager` to redraw the dragged component every time a mouse is moved.

Dropping is implemented by allowing the 'drag-and-drop' targets to be dropped onto the components that implement IDropTarget interface. To drop a target a dropDragObject(...) method is called on the component that the target is to be dropped onto, which is implemented differently depending on the component that is receiving the drop target.

Although we have a centralised object coordinating dragging, the responsibility of dealing with dropped targets is delegated to the individual components that implement `IDropTarget`. This structure gives us more flexibility over dragging and dropping and allows us to make components that were previously unable to receive drop object drop targets just by implementing `IDropTarget` methods for those components.

6.6 Programs

This section talks about a GUI representation of the pipeline programs that exist in the framework. GUI programs contain GUI representations of arguments and IO nodes. They allow the user to visualise the pipeline and its components and make it possible for the user to manipulate those components.

6.6.1 Specification

This section refers to the part of our original specification which states that one of the minimum GUI requirements of the project is to provide a graphical representation of pipelines that is easy to understand. It must be easy for users to identify programs, arguments and IO nodes used to connect pipes in the pipeline.

6.6.2 Overview

6.6.2.1 Design

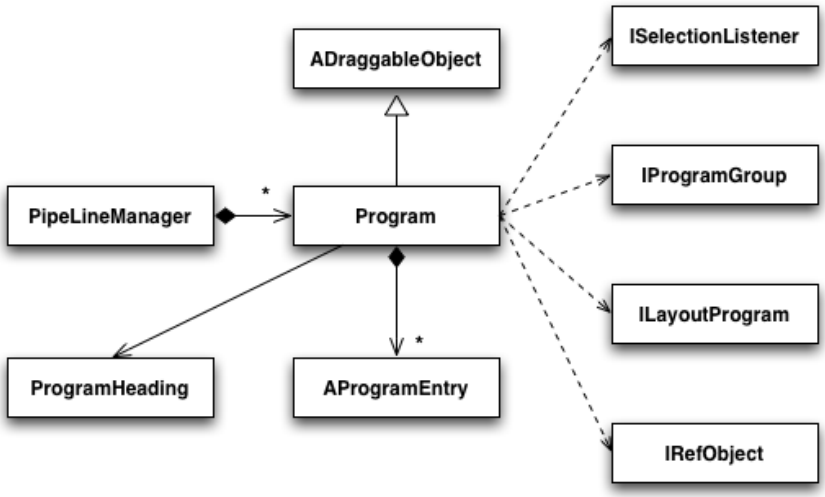
The functionality of the program in the GUI is very broad, therefore the design of this part of the system had to take this factor into account. In particular, the design of GUI programs had to deal with the following:

- **Programs must be easily manipulated by the user.** The user must have the ability to drag and drop program onto the canvas and move it within the canvas after it has been dropped. The user also must have the ability to access all the programs and their entries via the keyboard.
- **Programs must be resizable.** Programs contain and display arguments and IO nodes, therefore they must be able to change their size if extra arguments or nodes are added.
- **Programs must be easily identifiable.** The user must easily identify programs, therefore programs must have their user-friendly names displayed.
- **Programs must have the ability to notify the framework of any changes made to them.** The programs must be able to notify the framework when they are added or removed by the user to/from the pipeline. Also, the notification of change is required if the user added extra arguments to the program or connected a pipe to the program's IO nodes.
- **Programs must be able to communicate with the Help system.** They must call relevant functions on the help system to display help when it is requested by the user.
- **Programs must have the ability to expand and contract.** Expandeds program show all their argument and IO node entries wheras the contracted programs only shows their names.
- **Programs must have layout information associated with them.** Programs can be saved to disk and reloaded at the later stage. In order to draw the programs at the positions they were on the canvas when the pipeline was saved, each program must make its layout information available to the loading and saving engine. They must also allow the layout information to be assigned to them.
- **Programs must be able to be selected as part of the group.** This is required in order to allow the user to create simple macros.

6.6.2.2 Implementation

In order to fulfil the design requirements we came up with the following implementation architecture:

Figure 6.13 . An overview of the architecture used to implement the functionality of GUI Programs.



In order to allow users to drag and drop programs onto the canvas, the `Program` has to subclass `ADraggableObject`. Such architecture also deals with the distribution of the mouse events to the `Program` and allows the user to manipulate programs using the mouse.

To capture the functionality of expanding and contracting the program, resizing the program and calling help, we have introduced the `ProgramHeader` class which has the responsibility of dealing with the above features of GUI programs. When a program is selected, the program header shows buttons which, when pressed, will perform contracting, expanding, deletion and resizing. `ProgramHeader` also addresses the issue of easily identifying programs by displaying the program name as obtained from the HIAL, or a user-defined alias for it.

`AProgramEntry` is a superclass for classes used to display program entries, such as arguments, input and output nodes, which will be covered in detail in the appropriate sections.

`PipeLineManager` is the class used to put the pipeline together. It holds all relevant information about the pipeline, i.e. the programs used in the pipeline and the pipes connecting these programs. When a program is added to the canvas it is automatically added to the `PipeLineManager`, which then contacts the HIAL to notify it that the program should be added to the framework. If the correctness of the program has been verified, the HIAL calls the `PipeLineManager` back at which point the program and its arguments and IO nodes get displayed on the canvas.

6.6.2.3 Interfaces

We have introduced interfaces that `Program` implements in order to achieve the functionality described in the design requirements, such as notifying the framework when the program has been added to the canvas, assigning the layout information to the program and using it to auto-layout programs and to load programs at the same positions they were at before the pipeline was saved.Defining interfaces also allowed different group members to work on inter-dependant parts of the project with minimal conflicts.

- **ILayoutProgram interface.** This interface addresses issues of assigning and retrieving layout information of programs. This information can be used by the layout engine to place programs on the canvas and by the saving and loading engine to load the programs to the positions specified at the time of saving.
- **IRefObject interface.** This interface is used for the communication between the HIAL and the GUI. To encourage modularity and transparency, the HIAL should know as little as possible about GUI abstractions and representations of concepts used in other parts of the project. `IRefObject` is implemented by `Program` and `Pipe` and used by the HIAL to distinguish between GUI objects and to refer to them when needed. Thus, when the program is added to the canvas and the GUI informs the HIAL that such event has occurred it passes an `IRefObject` to the HIAL as one of the arguments to the `addProgram(..)` method. When HIAL needs to communicate back to the GUI that the program is verified and can now be drawn on the canvas, it passes back the `IRefObject` received from the GUI so that the correct program can be displayed on the canvas.

6.7 Macros

One of the most powerful features of existing shells is the ability to package up many commands into one single command. Kevlar provides this functionality through macros.

6.7.1 Specification

This section covers the requirement that the user should be able to construct basic macros. The user should be able to select a section of an existing pipeline and group it to form a single macro program. The user should then be able to save this program and load it for use in other pipelines. Any unconnected nodes of programs inside the macro should be connectable from the macro program.

6.7.2 Design

Macros do not provide any additional functionality to Kevlar, however they do make it easier for the user to manipulate groups of programs. Because of this, macros are handled entirely by the GUI, and any operations upon them are translated to their component programs before being sent to the HIAL. Therefore, the framework has no knowledge of macros.

To create a macro, the user must have a way of selecting multiple programs using both the mouse and the keyboard. For the mouse, we decided that a form of drag-multi-select was appropriate, where the user drags a box around the programs to be selected. For the keyboard, we chose to have a shortcut that would toggle whether the currently selected program is included in the selection. Once selected, a single button press should remove all the selected programs and pipes from the graphical representation and replace them with a single macro program.

6.7.3 Multi-Select Implementation

To allow for multiple programs to be selected, a new 'highlighted' state was added to each program to represent when it's part of a selection. The `PipeLineManager` maintains a list of programs currently selected, and constructs a `SelectionBox` widget which displays a bounding box around them, along with a number of buttons for performing actions on all the selected programs.

To implement drag-multi-select, the `PipeLineManager` records the position of the mouse received in its `mousePress(..)` method, and uses this to calculate the dimensions of the drag box whenever its `mouseMoved(..)` method is called. When the drag box is resized, it calls an internal `updateHighlightedProgram` method which calculates which programs lie inside the box, calling their `highlight()` method and adding them to the `multiSelect` list if they do, and calling their `unhighlight()` method and removing them from the `multiSelect` list if they do not. When the `mouseRelease(..)` method is called, the `PipeLineManager` invokes an `adjustSelectedArea()` method which constructs a `SelectionBox` instance and dimensions it to be the smallest bounding box surrounding all the selected programs. This box contains buttons for performing delete, expand/contract and create macro operations upon the selection. When the 'create macro' button is pressed, the `PipeLineManager`'s `createMacro()` method is called.

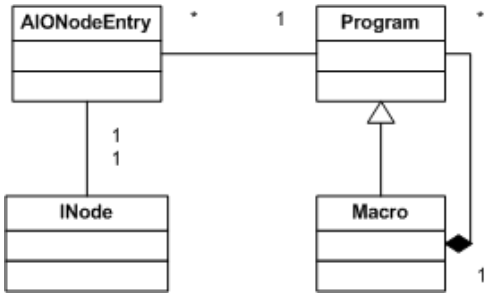
To implement multi-selection using the keyboard, a new `Action` class, `ToggleMultiSelectAction`, was created and associated with the 'spacebar' key press in the 'program selected' state. When invoked, this action calls the `PipeLineManager`'s `addToMultiSelect(Program p)` method passing it the currently selected (singly selected) program, and switching the current state to a new `ProgramMultiSelectState`. In this state, certain keyboard operations such as expand/contract, program movement and deletion are applied to all the programs in the multi-select. An additional `CreateMacroAction` action was created and associated with the 'ctrl+M' key press in the `ProgramMultiSelectState` state, which invokes the `PipeLineManager`'s `createMacro()` method when executed.

6.7.4 Macro Representation Implementation

Since the visual representation of a macro is very similar to that of a program, we decided to reuse the program implementation by making the `Macro` class extend the `Program` class. The `Macro` class additionally maintains a set of references to the `Program` instances that it contains, which it uses for translating operations upon the macro to operations upon its component programs that can be understood by the HIAL.

In order to propagate unconnected and externally connected nodes to the macro level, the `Macro` class contains an `Update` method which obtains a list of nodes from the programs contained within the macro and iterates over them. For each node, it creates a new `AIONodeEntry` for the macro iff the node is either not connected to a pipe, or is connected to a pipe which leads to a program that is not contained within the macro. If a node is added that already has a pipe, the `Macro` must also disconnect that pipe from its current node and reconnect it to the new node entry created by the macro, so that the pipe appears to be connected to the macro in the graphical representation.

Figure 6.14 . UML diagram showing the relationships between classes related to macros.



In programs, `AIONodeEntry` instances contained a reference to the HIAL's representation of the node on which any operations would be performed. Unfortunately, such a representation does not exist for macros, since they do not exist inside the HIAL. However, whenever an operation is performed upon a macro node, we want the operation in the HIAL to be performed upon the corresponding node of the program contained within the macro. To achieve this, when iterating through the nodes of the programs contained within the macro in order to decide which nodes are propagated to the macro level, any entries created are given the same HIAL reference as the node they are propagating.

Once a macro has been created, it can be saved as a pipeline and then loaded multiple times using the normal saving and loading procedures to create many instances that can be used in different situations.

6.8 Pipes

GUI pipes are graphical representations of pipes in the framework, which show to the user how data will flow between programs when the pipeline is executed.

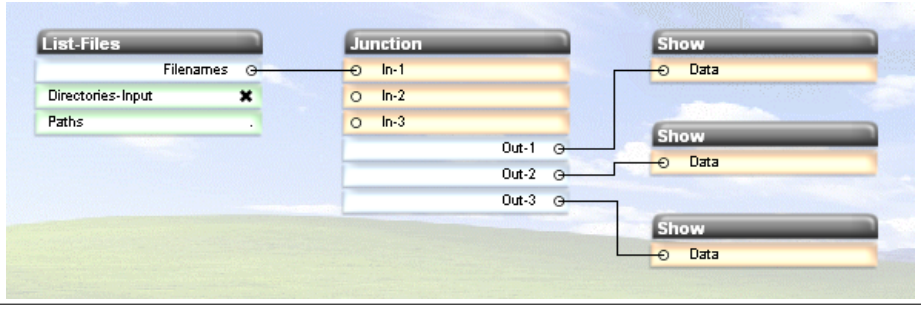
6.8.1 Specification

Graphical pipes, along with other pipeline components, fulfil the requirement that Kevlar must provide a graphical representation of pipelines that is easy to understand. They also alert the user if connected to incompatible nodes, fulfilling the requirement that the user must be given feedback on construction errors.

6.8.2 Overview

The pipe is a fundamental component of a Kevlar pipeline, and its graphical representation is therefore used very often. For simplicity and performance reasons we decided to represent pipes as simple lines, although to avoid the interface looking too cluttered we opted against diagonal lines, so we restricted each segment of the line to be either horizonal or vertical.

Figure 6.15 . Screenshot showing pipes in the release version of Kevlar.



Pipes are connected between input and output nodes of programs, and must therefore be updated whenever either their source or destination programs are moved or resized. Since nodes in Kevlar have type information associated with them, pipes can be in either a valid or invalid state, depending on whether their source and destination nodes are of compatible types.

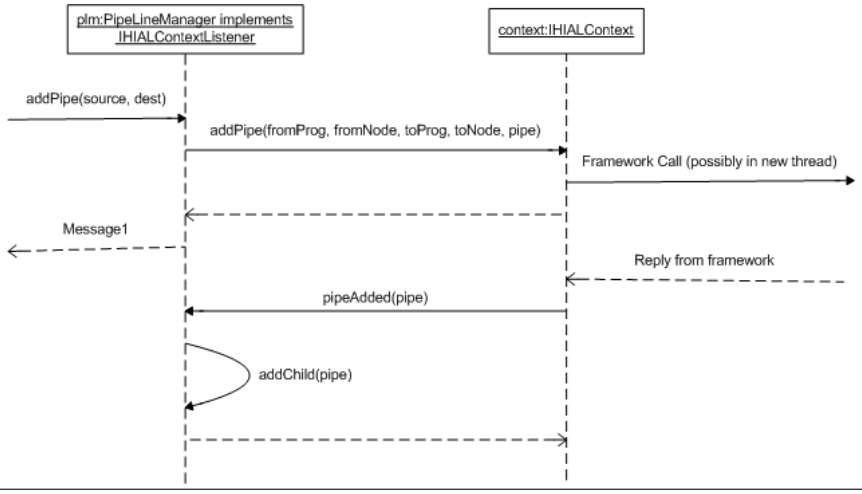
6.8.3 User Interaction

The user adds a pipe to the pipeline by selecting any input node and output node. The `PipeLineManager` is notified by program nodes when they are clicked, and ensures that only one node may be in the selected at any one time. If a node is already selected when another is clicked, the `PipeLineManager` first checks to see if they are both inputs or both outputs, in which case the previously selected node is simply deselected and the new node selected. If they differ, then an input-output pair has been chosen, so a pipe is created and the selection is cleared so that no node is selected. If a pipe is already connected to either of the nodes chosen, that pipe is first removed before the new pipe is added. To remove a pipe, the user must right-click on either of the pipe's nodes, which notifies the `PipeLineManager` to remove it.

6.8.4 HIAL Interaction

In order to successfully add pipes to a pipeline in Kevlar, the graphical interface must alert the framework to add the pipe to its internal representation, so that the framework can transfer data between the connected programs when the pipeline is executed. The GUI does this by contacting the framework indirectly through the HIAL mediator. The GUI's `PipeLineManager` first constructs an instance of the `Pipe` class which will be used to represent the pipe on the screen. However, to avoid the graphical representation and the framework's representation from getting out of sync, this pipe is not directly added to the GUI pipeline. Instead, it is passed in a call to the HIAL context's `addPipe(..)` method, along with the identifiers of the nodes that the pipe is to connect. The HIAL then contacts the framework and requests the pipe to be added, causing the framework to check whether the nodes have compatible types. The HIAL then calls-back the GUI's `IHIALContextListener` instance (which is the `PipeLineManager`), using either the `pipeAdded(..)` or `pipeAddFailed(..)` method. The original `Pipe` object constructed by the `PipeLineManager` is passed as a paramter to these methods so that it can be added to the graphical pipeline if necessary.

Figure 6.16 . Sequence diagram illustrating the invocations when adding a pipe in the GUI.

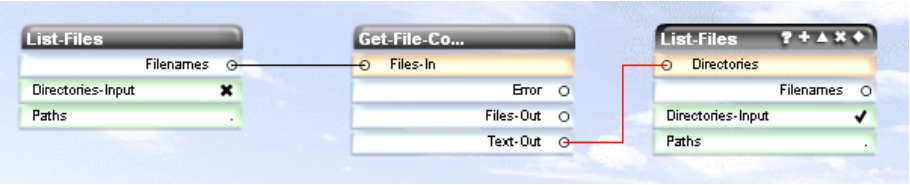


6.8.5 Valid and Invalid Pipes

Pipes that are valid appear as black pipes in the graphical interface and also exist in the framework. The GUI is responsible for ensuring that the framework's representation and the graphical representation

displayed to the user is semantically equivalent, so it must ensure that when the user wishes to remove a valid pipe, it is also removed from the framework's representation.

Figure 6.17 . Screenshot showing an invalid pipe, since text cannot be passed as a list of filenames.

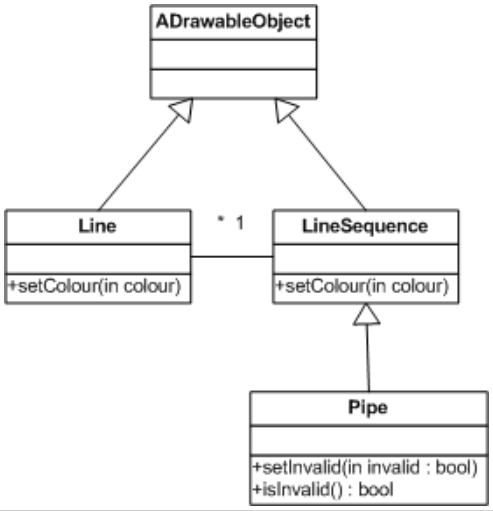


Invalid pipes, however, are merely a graphical aid to tell the user that the requested pipeline is not type-safe. Although they appear in the GUI as red lines, they do not exist in the framework's representation of the pipeline. Because of this, the GUI must not request for the framework to remove invalid pipes, since doing so would cause an exception.

6.8.6 Implementation

Pipes in the GUI are represented by the `Pipe` class, which simply extends the `LineSequence` widget. The `LineSequence` class maintains a set of `Line` instances representing each horizontal or vertical segment of the line. The actual path of the line is decided by the current layout (provided by the `LayoutManager` singleton), which is notified of the start and end points and calls the `addPoint(Point p)` method in `LineSequence` which creates a `Line` instance between each point in sequence.

Figure 6.18 . UML diagram showing the relationships between classes related to drawing pipes.



6.9 IONodes

Input and output nodes provide a reminder to the user of the data taken and produced by each program and represent connection points to which pipes can be connected.

6.9.1 Specification

Along with pipes and programs, IO nodes fulfil the requirement that the graphical interface must provide a graphical representation of pipelines that is easy to understand.

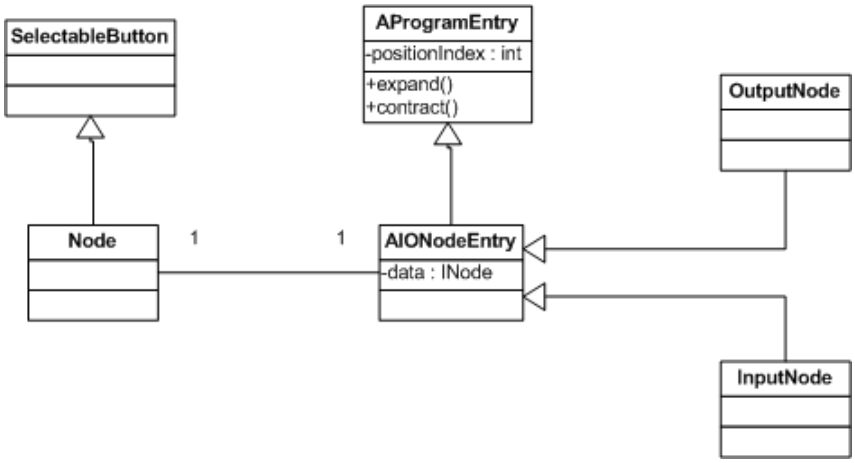
6.9.2 Design

In our design, we highlighted a number of features that IO nodes must contain:

- **They must be easily identifiable by the user.** The user should be able to distinguish between input and output nodes easily, and they should be named to indicate the data received or produced.
- **They should encourage a left-to-right flow direction.** In order to make the direction of data flow easier for the user to understand, we decided to encourage a left-to-right flow direction in pipelines. IO nodes should make it easier to connect up pipes in this way.
- **The user should be able to select IO nodes to connect up pipes.** IO nodes should be selectable using both the mouse and using keyboard shortcuts. Selecting an input and an output IO node should cause a pipe to be connected between them.
- **The user should be able to access help for each node, if available.** If a program has specified help for a node, the user should be able to access it by right-clicking on the node's entry.
- **The user should be able to hide them.** Since programs may contain a large number of IO nodes, there should be functionality that allows them to be hidden by the user when not needed, so that they do not take up unnecessary screen space.

We realised from these requirements that IO nodes and arguments share similar properties, in that both are contained within programs, must be displayed by name and can be hidden if the user chooses. To reflect this, we decided to create an `AProgramEntry` abstract class which would perform these shared actions, which is then extended by the abstract `AIONodeEntry` class, which provides implementation specific to IO nodes. Finally, this is subclassed by `InputNodeEntry` and `OutputNodeEntry` to provide specific implementation for inputs and outputs.

Figure 6.19 . UML diagram showing the relationships between classes related to IO nodes.

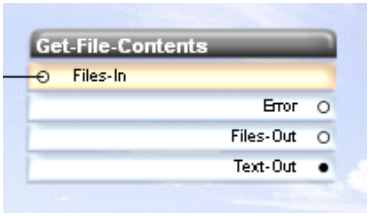


6.9.3 Implementation

The `AProgramEntry` class provides functionality for representing a single entry in an argument and IO node list. It's constructor takes an `int positionIndex` which represents its logical position in the list, which is then used to calculate its actual position as a child of the `Program` instance it belongs to. It also provides `contract()` and `expand()` methods which trigger an animation for hiding and showing the entry respectively.

IO nodes are represented in the HIAL by classes that implement the `INode` interface, which are retreived by the GUI by calling methods of the program's HIAL representation. A new `AIONodeEntry` is created for each HIAL `INode` of a program, and the `String getName()` method is used to obtain a user-friendly identifier for each node. The `AIONodeEntry` class also constructs an instance of the `Node` class, which represents the actual connection point on which the user can click to select the node.

Figure 6.20 . Screenshot showing input and output nodes. The input is connected to a pipe, whilst one of the outputs is selected.



The drawing methods for IO nodes are stored in the `InputNodeEntry` and `OutputNodeEntry` classes. Each draws a different, colour-coded background to help the user to distinguish between input and output nodes. These classes are also responsible for positioning the `Node` widget on the left for inputs and on the right for outputs, in order to encourage the left-to-right flow direction.

The `Node` class for representing the clickable connection point is a subclass for `SelectableButton`, which provides the implementation for changing its appearance when it is selected and when the mouse hovers over it. When the user clicks the node, its `mousePress(..)` method checks whether the node is currently selected or not and calls the `selectNode(..)` and `deselectNode(..)` methods in the `PipeLineManager` respectively. The `PipeLineManager` is responsible for detecting which nodes are selected and connecting pipes between input-output node pairs.

6.9.4 Node Help

In order to display node help, the `AIONodeEntry` class overrides the `mousePress(..)` method to capture when the user right-clicks on the argument entry. When this event occurs, it calls the `getArgumentHelp()` method of its HIAL object to obtain any available help on this node. This information is then passed to the `HelpManager` class, which constructs an HTML page of help and displays it on an SWT `Browser` widget (see Section 6.13).

6.10 Arguments

6.10.1 Specification

This section refers to such specification requirements as allowing users to easily identity components of the pipeline and use them to construct pipelines efficiently. GUI arguments also address the issue of restricting users ability to enter incorrect values for arguments.

6.10.2 Overview

6.10.2.1 Design

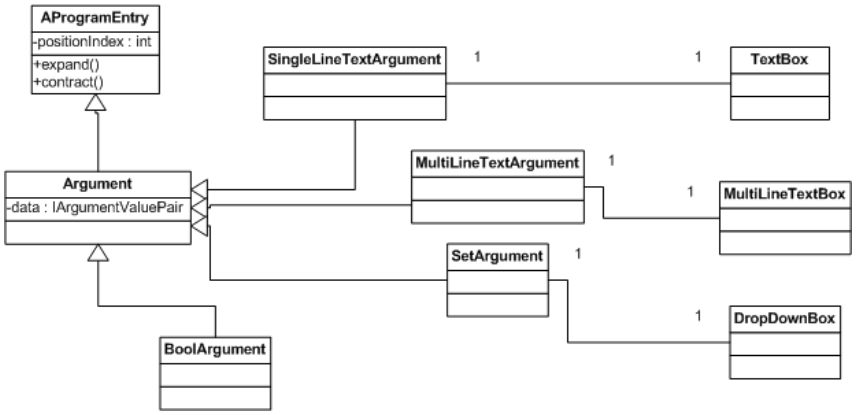
We highlighted a number of features of argument design that we thought were important for the final product:

- **They must be easily identifiable by the user.** Arguments should be displayed by name so that the user can easily identify how each argument is likely to affect a program.
- **They must be edited using a widget suitable for their type.** In order to restrict the user's ability to enter incorrect argument values, appropriate widgets should be used when editing them. For instance, a drop-down box should be used for set values, since it restricts the user to choosing one option from a set of valid options.
- **Text arguments must be validated.** If the program specifies a regular expression for validating a text argument then it must be applied every time the argument is edited. If the new value is invalid, the user should be notified.

- **Boolean or set arguments must be able to change the program's IO nodes.** The available IO nodes of a program can depend on the current values of any boolean or set arguments, so editing these arguments may cause new IO nodes to appear and existing ones to disappear. Therefore, whenever arguments of these types are edited, the program must update its IO node representations to match the nodes in the framework's representation.
- **The user should be able to access help for each argument, if available.** If a program has specified help for an argument, the user should be able to access it by right-clicking on the argument's entry.
- **The user should be able to hide them.** Since programs may contain a large number of arguments, there should be functionality that allows them to be hidden by the user when not needed, so that they do not take up unnecessary screen space.

Having already highlighted in our design of IO nodes (see Section 6.9) many feature that they share in common with arguments, we designed the `Argument` class to subclass `AProgramEntry` which would provide the `contract()` and `expand()` methods for hiding them. The `Argument` class itself would provide the implementation for accessing argument help when the user right-clicks on the entry, by overridding the `mousePress(..)` method. From the HIAL's design, we discovered that there are four different argument types; single-line text, multi-line text, set and boolean. Since each of these would need to be displayed and edited separately, we decided to produce a different subclass of `Argument` for each. These subclasses would implement the `drawBefore(..)` method for displaying the argument values, and would also provide implementation for a `showEditor()` method that would construct the widgets needed for editing the argument's value.

Figure 6.21 . UML diagram showing the relationships between classes relating to arguments.



6.10.2.2 Displaying and Editing

For single-line text arguments, the `SingleLineTextArgument` class displays as much of the text value that will fit in the argument's entry. When the user wishes to edit the argument, the `showEditor()` method constructs a `TextBox` instance and sets its contents to be the current value. It then passes the text box to the `PopUpManager` which displays it as a pop-up over the current argument. This allows the text box to be larger than the argument entry, so the user can see more of the argument value while editing it. The `SingleLineTextArgument` class registers itself as a listener on this pop-up, and is notified when it is hidden (which is triggered either by it losing keyboard focus or the user clicking elsewhere in the window). When it receives this notification, it sets the value of the argument to the current contents of the text box and notifies the HIAL of this change.

For multi-line text arguments, the `MultiLineTextArgument` class acts very similarly to the `SingleLineTextArgu` class. Since the argument's value can contain newlines, only the first line of the value is displayed in the argument entry. To edit the value, a `MultiLineTextBox` instance is constructed to show and allow the user to edit all the lines. As before, this is passed to the `PopUpManager` so that it can appear larger than the argument's entry.

For set arguments, the `SetArgument` class displays the currently selected option from the valid set. To edit this, a `DropDownBox` instance is created and populated with the possible options. As before, this is displayed as a pop-up, and any changes are applied when the pop-up is hidden.

For boolean arguments, the `BoolArgument` class displays either a tick or a cross in the argument's entry to represent whether the value is true or false. To edit the value, no editor is required, instead simply selecting the argument inverts its value.

6.10.2.3 Applying Argument Changes

When any arguments of a program are edited, the `applyArgumentChanges()` method of the `Program` class is called, which constructs a list of all the arguments and their new values. This list is then passed to the HIAL through the `updateArgumentValues(..)` method of the program's HIAL object, which validates the text values against any supplied validation regular expressions and updates the framework's internal representation of the program. The `Program` instance then finds out which arguments failed validation by calling the `getInvalidArguments()` method of it's HIAL object, which returns a list of arguments that are invalid. It can then iterate through this list, locate the graphical entry for each invalid argument, and call its `setInvalid(boolean invalid)` method which turns its visual representation red to alert the user. Finally, if any of the changes made to the arguments cause the IO nodes of the program to change, the HIAL calls the `programStateChanged(IRefObject program)` method in the GUI's `IHIALContextListener` which updates the program's IO node entries to match those supplied by the HIAL.

6.10.2.4 Argument Help

In order to display argument help, the `Argument` class overrides the `mousePress(..)` method to capture when the user right-clicks on the argument entry. When this event occurs, it calls the `getArgumentHelp()` method of its HIAL object to obtain any available help on this argument. This information is then passed to the `HelpManager` class, which constructs an HTML page of help and displays it on an SWT `Browser` widget (see Section 6.13).

6.11 Task Panes And The Toolbar

Task panes contribute a large part of the functionality of our project. They serve two purposes; to achieve required functionality described by the specification, and are used to simplify operations users have to perform to achieve a task.

6.11.1 Specification

This section refers to a few sections of our original specification. It covers requirements such as giving the users ability to save and load pipelines and to allow the user to search for programs that can be used to achieve a task by entering keywords associated with the task. This section also introduces design decisions that we believe encourage high usability, such as having a history of constructed and executed pipelines that can be reused in the current session without saving them to disk.

6.11.2 Overview

6.11.2.1 Design

Task panes are shown on the right side of the canvas. However, they are not visible all the time. The user can expand a task pane to its full size, alternatively the user can hide the task pane to have more space on the canvas to construct pipelines. This feature makes task panes non-intrusive, i.e. the task panes are only visible when user needs them and they don't interfere with user's task of pipeline construction.

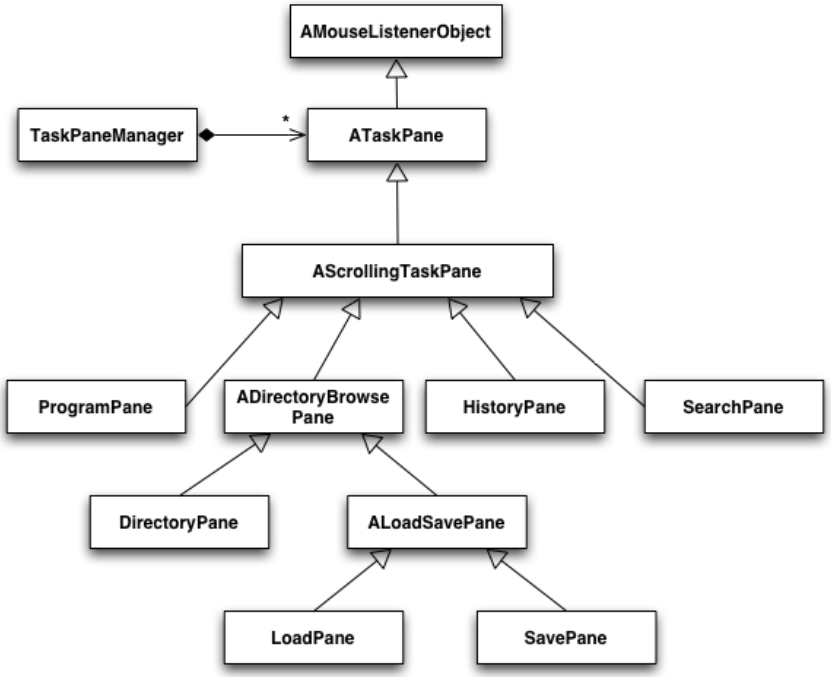
We have decided assign a task pane functionality aspect that it is supposed to serve. Therefore, we have a few task panes each serving a specific purpose. Only one task pane is visible at a time and the user can switch between task panes.

Since each task pane only deals with one functionality aspect, this allowed us to develop individual task panes separately of the rest. It also reduced coupling between different components and if the functionality of the system expands requiring additional features, some of them can be implemented by adding another task pane which would deal with a particular functionality aspect.

However, all task panes share some common functionality, e.g. expanding and contracting. Also, all task panes use a customary vertical scroll bar widget, which is used in case a task pane contains entries that have to be scrolled to by the user. Mouse events have to be distributed to all task panes in a same manner, so that functionality is also common for all task panes. Thus, a common superclass was needed to provide common functionality to all task panes and also to provide methods that deal with events response to which is the same across all task panes.

Therefore, we have decided on the following architecture:

Figure 6.22 . Architecture used to implement Task panes.



6.11.2.2 Implementaion

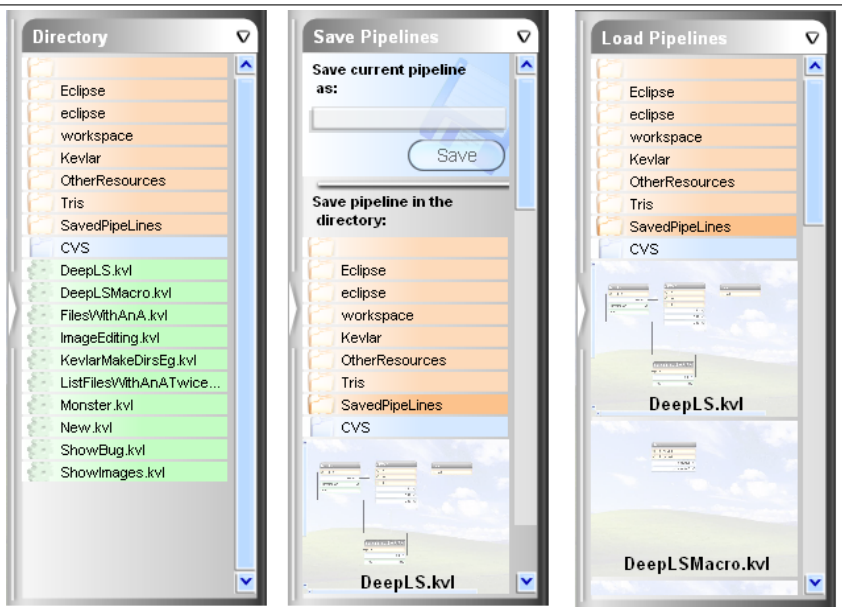
All task panes subclass `ATaskPane`, which is the class that outlines common functionality and by extending `AMouseListenerObject` guarantees distribution of the mouse events to the task panes. `AScrollingTaskPane` is responsible for the behaviour of our customary vertical scroll bar and therefore by extending this class all task panes get this customary widget.

`TaskPaneManager` contains references to all task panes and it controls switching between panes. It is also responsible for sliding of panes in and out on user demands. In addition, this class deals with providing reference to a given task pane to other parts of the system when such reference is required.

`ADirectoryBrowsePane` was introduced, since `DirectoryPane`, `LoadPane` and `SavePane` all share similar functionality which only differs slightly. All these task panes display a tree of directories available on user's computer. `DirectoryPane` also shows the file entries whereas `LoadPane` and `SavePane` only show thumbnails of saved pipelines if those are contained in the directory rather than showing all users' files.

Thus `ADirectoryBrowsePane` deals with displaying the parent and child folders for a given directory, `DirectoryPane` is responsible for displaying file entries for the directory pane and `ALoadSavePane` is responsible for displaying the thumbnails of saved pipelines.

Figure 6.23 . Screenshots of director, save and load panes to illustrate similarity in displaying the directory tree.



6.11.3 Components overview

6.11.3.1 Program pane.

Program pane is used to display a tree of programs available in our shell. The programs are organised in the directories according to the functions that they perform. The user can navigate through a program pane by clicking on the directory of interest. The entries are colour coded to make it easier for the user to distinguish between them.

Program pane allows the user to quickly access programs and view help for those programs. If the user wants to use the programs displayed on the program pane, he/she can simply drag the program entry off the pane onto the canvas.

6.11.3.2 Search pane.

Search pane is used to fulfil our extended specification requirement which states that the system should allow the user to search for programs to achieve a task. The search pane contains a textbox where the user can enter the keywords he/she wants to search for. The keywords are then passed to the search pane, which filters any punctuation, space or new line characters that the user might have used to separate the keywords and submits a list of keywords to the HIAL, which actually performs the search.

The HIAL returns a list of program entries that match the keywords it has been given by the search pane. If nothing matches the user's criteria a relevant message is displayed on the search pane, otherwise the program entries returned by the HIAL are displayed. If the user wished to use any of the program entries found by the search, he/she can just drag that program entry onto the canvas, similar to the program pane.

6.11.3.3 Directory pane.

Motivation behind the directory pane is that commands like "change directory" and "list the contents of the directory" are used very often in a shell. To save the user time and effort of constructing a pipeline for such an easy task, a directory pane was created.

The directory pane displays the directory tree, which exists on the user's machine. To encourage the same look and feel across all the panes, so that it's easier for the user to adapt to our shell, we have used

the same navigation principles for the directory pane as the ones used in program pane, i.e. entries are colour coded and the user can navigate by clicking on the directory he/she wants to go into. When a particular directory is selected, the directory pane displays the contents of the directory including parent directories, child directories and file entries.

Every time a user changes a directory, a directory pane informs the framework of this change and passes a current working directory to it. Therefore, often used combination "change directory, list files" is supported. If the user wants to construct pipelines which involve knowledge or use of the current working directory, the directory that the user has visited last in the directory pane is used for that purpose.

6.11.3.4 History pane.

History pane is used to keep a history of pipelines that have been constructed and executed in the current session. Once a user executes a pipeline it is added to the history pane in a form of a thumbnail showing the pipeline that the user constructed. The user can access the pipeline shown in the history by clicking on the thumbnail.

The context of the pipeline is saved, so that it can be loaded from the history pane later if the user so wishes. The mechanism of saving of loading to the hisotry pane the pipeline is the same as the one used in the saving and loading engine (discussed in the next section), except that the pipeline is not stored to disk.

To obtain a thumbnail of the pipeline, a method in the `PipeLineManager` was created called `createSnapshot()` which creates a new graphic context (GC) and makes a call to the `ADrawableObject` passing a newly created GC to draw the contents of the canvas on it. The image data that is created as a result of this call is then rescaled to the size of the thumbnail.

6.11.4 Saving and loading of pipelines.

Save pane and Load pane address a minimum specification requirement that once a pipeline has been constructed it should be possible to save it to disk and reload at the later date. Saved pipelines are represented as thumbnails and are displayed on both save and load panes.

Most of saving and loading is performed by the HIAL and the GUI only saves information which is relevant to drawing programs, arguments, IO nodes and pipes on the canvas. It is also responsible for saving and loading thumbnails data and for saving and loading of macros, since macros are a GUI abstraction only and other parts of the system don't know about them.

When the user decides to save a pipeline, save pane contacts the HIAL supplying the absolute path name of the file the pipeline should be saved in and the reference to a GUI object which the HIAL should call to save GUI related pipeline information.

Similarly, when the pipeline is loaded, the load pane calls HIAL and provides the name of the file to be loaded and the GUI object that should be called to load GUI specific information. The HIAL loads all the information relevant to it from the file and then calls GUI supplying the information needed to load the graphical content of the pipeline. The loading mechanism works as if the user has added the program to the canvas manually.

6.11.4.1 Overview of some utility classes used in loading and saving

GUISaverCallback. This class was used to save GUI sensitive data about the pipeline. For example, it was responsible for saving the layout information of the program, i.e. its size and position on the canvas. Since the format of the save file is xml, all the information written by GUISaverCallback had to be enclosed in consistent xml tags.

GUILoaderCallback. This class is responsible for loading of the GUI representations of programs, arguments and pipes. It takes the DOM object supplied to it by the HIAL and retrieves the GUI information about objects to be loaded. This information is used to place the objects correctly on the canvas, i.e. to the positions they were at when the pipeline was saved. Alternatively, if the canvas already contains components at those positions, the layout algorithm uses the retrieved information to draw the loaded pipeline around the components on the canvas.

Converter. Image data of the thumbnails showing the constructed pipeline that we use for representing saved files is stored as a hex string in the xml file used to save pipeline information. This class is used to convert between image data and a string of hex digits formats of the thumbnail representation.

To perform a conversion it makes use of the `ImageLoader` provided in the SWT library which saves the image data to the input stream. The `Converter` takes a byte array that represents the stream and converts that byte array into a string of hex digits.

6.11.5 Toolbar

Toolbar is used most of all as means of accessing task panes or a show pane quickly. It contains buttons, which when pressed slide out or switch to the relevant pane. It also contains buttons related to the execution of a pipeline; execute button and stop execution button. These serve as means of executing pipelines when the user uses mouse manipulation of pipeline components.

One other helpful function of the tool bar is that it contains a ”clear canvas” button, which when pressed removes all programs from the canvas instead of making the user to manually remove all programs before a new pipeline can be started

6.12 Show Pane

6.12.1 Purpose

In traditional command line consoles, output is always text (with some limited formatting abilities). In our shell, because we know the type of the data to be outputted, we can select a display method that is most helpful for the data type.

Images can be shown as an image slideshow, files can be shown in a list view control, which displays each detail of the file under a column that can be sorted.

This section explains the design of the output visualization component of the GUI. Figure 6.24 shows the show program that allows the user to specify what parts of a pipeline they want to have visualized.

Figure 6.24 . The user will use the show program to tell the shell which parts of the pipeline's output, they want to have visualized. The show program design was covered in Section 3.5

6.12.2 Structure

Figure 6.25 is a diagram which shows how the Show programs and the GUI's Show execution manager communicate in order to visualize output. When a pipeline is run, every show program executes the following routine:

- Show get's the current `IShowPaneExecutionManager` from the `ShowPaneExecutionManagerProvider`. This is the rendevous point with the GUI. When the GUI first initalizes, it sets this manager to its own version of the manager. Having this intermediate step ensures the GUI and Show implementation are independant.
- Show retrieves the 'Name', 'Show-Using' and 'Open-Lazily' arguments. It then examines the pipe connected to its input to discover the type of the data coming in. It will need this to instruct the GUI on what kind of output visualizer it requires.
- Show now requests an `IShower` from the Show execution manager. An `IShower` is an output visualizer provided by the GUI for showing objects. It displays the data it receives from calls to `.addObject( o )`. An example of an `IShower` would be, `ImageShower`, which is a GUI window for displaying images. `ImageShower` will take the object 'o' from `.addObject( o )` and add it to its image slideshow.

  The request for an `IShower` contains, a name which the GUI can use for the title of the visualization window, a request to use a specific Shower if the user specified one with the 'Show-using' argument, and the type of the data coming into Show.

  If 'Open-Lazily' is set, this step only happens when Show receives the first data object. This way, if no data objects are received, an output visualizing window will never appear.
- The GUI's `ShowPaneExecutionManager` now has to answer the Show program's request for a shower of a certain spec. First it has to find an `IShowerProvider` to make the Shower. The GUI keeps a pool of `IShowerProvider`s, such as `IImageShowerProvider`, and `ITextShowerProvider`.

  If a 'Show-Using' preference is not specified, the GUI will ask each shower provider what type of data its showers take. If it finds a match, it uses that ShowerProvider, otherwise it uses the closest type match.

  But sometimes, a user will not be happy with using the ShowerProvider that the GUI picks. In this case, the user can set the 'Show-Using' argument of the Show program to specify which shower to use. For example, they could specify to use a Text shower for the output of 'List-Files' even if a File shower is available.

  Now that a shower provider is picked, the GUI has two options. It can make a new shower. However, in the traditional console, output from programs can be mixed. It is the same in our shell. If there is already a shower open which has the same ShowerProvider as the one currently chosen, and the 'Name' passed from the Show program is the same, it is assumed that the user wants to mix the output of the two different Show programs, so the existing `IShower` is reused.
- The `IShower` is passed from the GUI to the Show program, and the show program starts adding data objects to the shower. To improve effeciency, the Show program will take objects in batches while there are still objects waiting to be shown, then it will add them all with a call to `.addAllObjects(obje`

6.12.3 Available showers

Two showers are implemented in the GUI. These are `ImageShower` and `AnyShower`. AnyShower is capable of showing any IType object by making a call to the `.toString()` method that all objects have. This also means that `AnyShower` makes a suitable substitute for a text shower. Figure 6.26 shows the `AnyShower` in use and Figure 6.27 shows the `ImageShower` in use.

The showers are nested inside a tab control. The title of the shower is placed in the tab title, and the user can freely switch between showers.

6.13 Help

One of the goals of our project was to create a shell that is user friendly and easy to use even for inexperienced user. From this perspective, having context-sensitive help available to users is an important feature of our project and is tightly coupled with the usability of our system.

6.13.1 Specification

Provision of context-sensitive help to the user when the user needs it was one of the extended specifications for our project. Help should become available for the program, argument or IO node when it becomes selected by the user. This help should include information on the selected item and on how to use it.

6.13.2 Overview

6.13.2.1 Design

When we started designing help system for our project we decided on a few features of the help systems that should be taken into account when implementing it:

- **It must be easily and conveniently accessed.** Help should be only a 'click' or a 'keyboard shortcut' away to allow the most efficient use of help facility. It also encourages more frequent use of Help and therefore the user doesn?t need to remember all the commands available in our shell.
- **It must not interfere with the user's current task.** Help should disappear as soon as the user clicks off the selected item.
- **It must be clear in description and easy to read.** Help should be displayed in the format most familiar to the user, so that it can be used efficiently.
- **It must be navigatable.** The user must be able to scroll it if necessary.
- **Only relevant information should be displayed.** Only information on the selected item should be displayed at any one time. This prevents overwhelming the user with information which might not be needed to him/her at the moment and therefore makes it easier for the user to receive most relevant help.

6.13.2.2 Implementation

We have implemented help system in our project by making use of the HelpManager class. Since we have decided that only relevant information should be displayed at any one time, only one help window should be visible at any one time. Thus, it made sense for the HelpManager to follow a singleton design pattern.

Help is available for all relevant components of the pipeline by clicking on the component of interest or by selecting it and pressing Ctrl+H keyboard shortcut. When such action is performed on the component it will call HelpManger.displayHelp(..) method to configure the help pop-up window to display the desired context-sensitive information.

In order to make Help easy to read and have an interface familiar to users we have used SWT Browser widget to display help. The Browser widget renders HTML, which allows HTML formatting to be used, including style sheets and images on the context of help. HelpManager receives help information from the pipeline component for which help is to be displayed, wraps it in HTML code and uses Browser to display it.

This approach makes the layout and appearance of help more user-friendly. The Browser widget also inherits from SWT Scrollable class, which makes it navigatable and allows users to scroll the help window if scrolling is needed.

Figure 6.25 . The communication structure between show programs and the GUI.

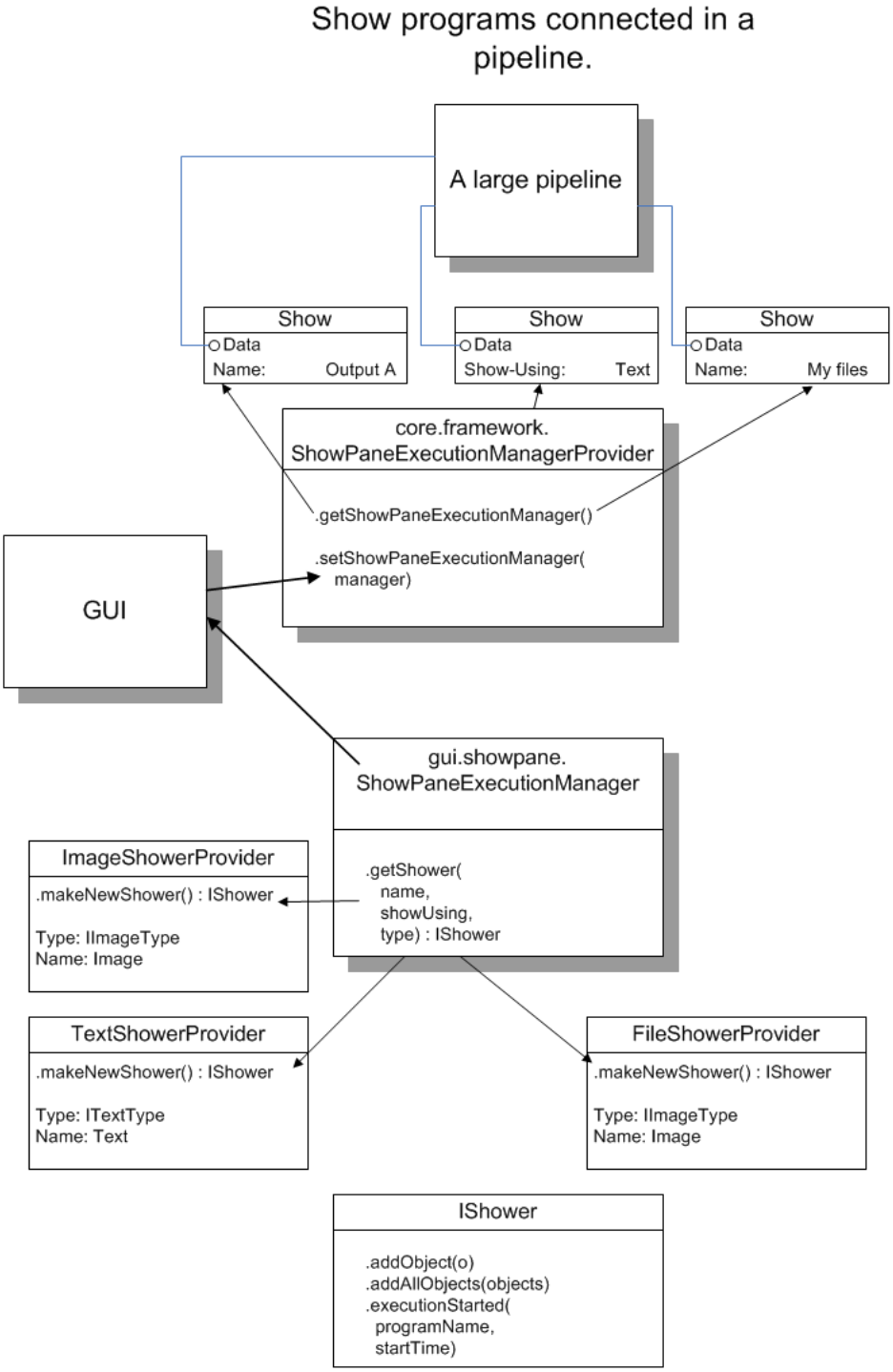


Figure 6.26 . The `AnyShower` is capable of showing text output. When the output gets to large, above a threshold of 32K characters, the top lines are flushed to ensure the shell does not use a never ending amount of memory.

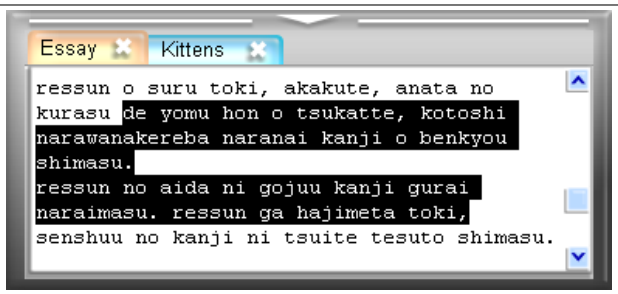
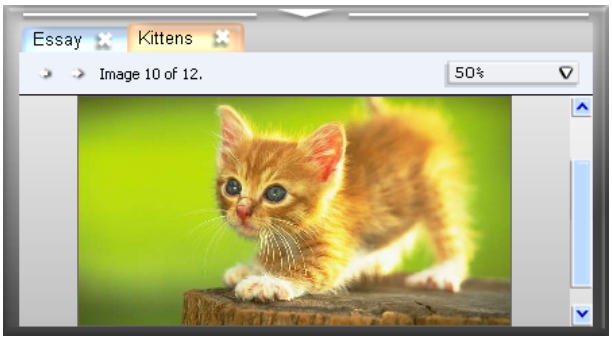


Figure 6.27 . The image shower allows the user to view the images as a slide show. They can go back and view the recent images. They can also set the zoom at which the image is viewed. If the most recent image is currently being viewed and a new image comes in, the newset image will be shown. This ensures the user can easily view older images without interruption.



Chapter 7

Evaluation

7.1 Specifications

In this part of the evaluation, we are going to take the original specifications, as outlined in Report1, and comment upon the level of achievement for each point.

7.1.1 Program and Types Specifications

7.1.1.1 Minimum Specifications

- **Programs can define input and output nodes that are named and typed.** A program can connect to other programs (via the Kevlar main shell) in order to exchange data if necessary. This is 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 can be given descriptive names to help the user, and also are 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 has 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 is able to check whether the given arguments are valid before the pipeline is executed as it exports a description of these arguments, and the Kevlar system validates the inputted arguments against these.
- **A core set of programs should be provided to demonstrate the system.** There is a small, but feature-complete set of programs that demonstrate all the different ways programs may interact with the Kevlar shell.
- **Third parties should be able to write programs for the framework.** Programs can easily be written and added to the framework. There is also a simple program that will wrap existing programs and pipe their input/output into the system.
- **Programs should have human-useable names.** Programs that have been written have been given intuitive names where applicable. They also export help and keywords to aid usage.
- **Types should be in a type hierarchy.** Programs are able to accept a range of different types by declaring acceptance of a supertype, and therefore all sub-types will then be accepted.
- **Users should be able to define their own types.** Types are easily added to the system using a plug-in based architecture.
- **A basic set of types should be provided for demonstration.** A basic set of types that allow the core features of Kevlar to be demonstrated is provided.

7.1.1.2 Extended Specifications

- **Programs should export help information.** A program can provide its own help on its function, arguments and the input and output nodes it contains.
- **Programs can accept and produce templated types.** Programs are 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.

7.1.1.3 Optional Specifications

- **Programs should notify the system when they are waiting for input.** Due to time constraints, this was not implemented.

7.1.2 Framework Specification

7.1.2.1 Minimum Specifications

- **Can connect programs together using pipes between their input and output nodes.** The Framework is able to make programs be connected together using pipes, and they can pass data between them.
- **Pipes should have a flow direction.** Pipes can only be connected from output nodes to input nodes, signifying that data flows out of the output and into the input. Any other combination is not be allowed.
- **Pipes should be type checked.** The Framework checks 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 will only execute valid pipelines after they have been checked for type-safety.
- **Should handle program crashes.** The whole system does not crash if an individual program crashes during its execution.
- **Should pick up newly registered types and programs from a central repository.** New programs and types are easily added to the Framework. This is done by simply allowing users to drop JAR files into certain directories which are periodically checked and the files dynamically loaded.

7.1.2.2 Extended Specifications

- **Pipes should have the functionality of being Templated.** The Framework accepts pipes that are templated. This means that the type of the pipes is inferred dynamically by the Framework.

7.1.2.3 Optional Specifications

- **Should support temporal and conditional temporal options (cf. && and ; in Unix based operating systems).** Due to time constraints, this was not implemented.
- **Should support for and while loops.** Due to time constraints, this was not implemented.
- **Security managers should control what programs do at runtime.** Due to time constraints, this was not implemented.

7.1.3 The GUI Specification

7.1.3.1 Minimum Specifications

- **Should provide a graphical representation of pipelines that is easy to understand.** New users of the system can 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 is 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 is also supported so that complete pipelines can be constructed without using the mouse.
- **Should gives the user feedback on construction errors.** Users are 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 is suitable for 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 are 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 provides tools for visualising simple program output such as text, and images, 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, it can be saved to disk and reloaded at a later date.
- **Should follow a design process that encourages high usability.** The group internally where constantly trying and providing feedback on the user-experience within Kevlar, which then influenced future GUI design. Towards the end of the project we also undertook a usability study.

7.1.3.2 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 provides 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 can be shown context-sensitive help on how to use the selected item. This appears in a pop-up that is easily hidden and does not interfere with the user's current task.
- **Should allow for construction of basic macros.** The user is 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 inherits any unconnected input and output nodes from the pipeline it contains and renames those with conflicting names. The macro does not inherit any arguments; those set inside the macro are fixed.

7.1.3.3 Optional Specifications

- **Should allow the user to choose which arguments and nodes should be visible in macros.** Due to time constraints, this was not implemented.
- **Should give help and suggest corrections for invalid pipelines.** Due to time constraints this was not implemented.

- **Should give help about invalid argument values.** Due to time constraints, this was not implemented.
- **Should allow macros to be expanded and edited.** Due to time constraints, this was not implemented.
- **Should allow macro changes to affect multiple instances.** Due to time constraints, this was not implemented.
- **Should allow the user to construct temporal scripts.** Due to time constraints, this was not implemented.
- **Should provide a plug-in-able architecture for visualising different types of output.** Due to time constraints, this was not implemented.

7.2 Usability study

7.2.1 Motivation

The aim of the Kevlar project is to design and implement a new kind of shell which would have advantages over the traditional command line shell. In particular, we aimed to solve usability problems in existing shells. Therefore, to evaluate our product, we need to analyze the usability of Kevlar. This section discusses how we decided on a usability study that would be useful to the project, and how the results were used to improve and evaluate Kevlar.

7.2.2 Choosing a usability study type

7.2.2.1 Study type

The information on usability study types is taken from Jim Cunningham's second year notes on usability studies [1]. There are two types of usability evaluation studies that are potentially useful.

Summative report on conformance

- does not explain reason for performance
- pass/fail report against agreed measure

This is appropriate for simple decision making. A summative report just includes comparison between actual and desired performance, in terms of times, errors, etc. to justify the conclusion.

Formative or diagnostic report

- explains causes for users performance
- a diagnosis for feedback into design

This is necessary to support design decisions during the development process; performance inadequacies should be related to user behaviour, and data provided for re-design.

Since the Kevlar project is designed to address the usability issues in the current command line, it would be useful to perform a comparative, summative study that tests to see if Kevlar offers benefits in task completion over the traditional command line console. This way, we could analyze whether Kevlar is an improvement over the command line with regards to usability. However, in practice, this form of study would be difficult to do.

CHALLENGES WITH DOING A COMPARATIVE, SUMMATIVE STUDY

- For this type of study, statistical analysis is needed to decide the results.

[1] For copyright reasons, these notes are not on general release, so could not be cited.

- Since the Kevlar project does not have the resources to fund a study involving many people, it would be very difficult to get enough people to allow for statistically meaningful results.
- Since most of the people who are likely to volunteer for such a usability study would be the technologically literate peers of the Kevlar team who also have experience with using a traditional console, there would be a strong bias in the results.

However, a formative report that analyses user performance at would also be of use as this information could be used to shape the design of Kevlar, and identify usability issues. For this kind of study, it is not as crucial to have a large sample population. We decided to initiate a formative report on the usability of Kevlar to determine improvements for the Kevlar system and to help evaluate our product.

7.2.2.2 Evaluation process

Given that we have chosen to perform a formative study of usability, we need to decide what evaluation process we would use for this analysis. There are four main methods to evaluate usability [**?**].

1. **User Reporting.** This method involves interviewing users after using a system, or undertaking a survey of those system users.
2. **Specialist reporting.** A human computer interfaces specialist walks through the system as though they were a user and analyses usability.
3. **Observational methods.** An analyst observes users as they use the system. The observations are then classified, analyzed and evaluated.
4. **Analytic methods.** This method does not involve any use of the actual system. Instead, established principles are used to reason about the system.

We ruled out specialist reporting due to the impracticalities of finding a HCI expert given the project budget, and decided that it would be difficult to find enough users willing to participate in an observational study.

Analytic methods are possible, and we recognized the usefulness of being able to evaluate the system without needing actual users. We found it useful to do keystroke modelling of using the Kevlar system compared with using the Console. (See Section 7.3.2). However, in general, for the majority of the usability issues we are interested in, we could not find a suitable analytical methods.

User reporting is easy to implement, and does not require any budget. This makes it a feasible evaluation method for the study. We decided to use user reporting as our evaluation method. This means that we will receive design feedback from surveys and interviews with users that have used Kevlar.

7.2.3 Designing the usability study

7.2.3.1 Overall design

Since we had decided on performing a formative usability study with a user-reporting based evaluation process, we need to decide on the design for collecting user reports.

AIMS OF THE STUDY

- **Direct design decisions.** As the main use of this kind of study is to help direct design decisions, this will be the focus of the survey design.
- **Find design issues that can be improved.** We wish to locate individual usability issues with Kevlar that can be improved.
- **Evaluation.** However, since we also wish to evaluate the Kevlar system, this will also be a consideration. For this reason, there will be a small amount of questions that are possibly more appropriate in a summative report. However, we will have to be careful not to draw any conclusions based solely on numerical analysis of results due to the problems covered in Section 7.2.2.1

To simplify the task of collecting results, we decided on using survey-based report collection over conducting interviews. This avoids problems associated with interviews such as interviewer-introduced bias, and requiring increased labour over the survey method.

Since there are no existing users of the Kevlar system, we felt we had to start by developing tasks for users to undertake, so that they were able to answer questions on usage of the system. These tasks were designed to cover the full range of tasks that can be done in the Kevlar shell.

7.2.3.2 Task list

We required users to perform 6 tasks before filling out the Kevlar survey. Each task has an overall aim, and a set of steps on how to perform the task. Below, the aims are presented, but the full tasks including the steps for those tasks can be found in Section 10.2

TASK AIMS

- **1.** Get a list of files in your current working directory. Learn how to navigate in Kevlar and learn the different ways of getting at programs.
- **2.** To save the pipeline made in task 1.
- **3.** To load the pipeline saved in task 2.
- **4.** To search for programs and use program arguments. We want to list all files that have the letter 'a' in their name.
- **5.** To use arguments to change nodes of programs. To build a pipeline to list the contents of just the first sub-directories of the current directory.
- **6.** To build an advanced pipeline that takes images in a directory, resizes and rotates them, and shows the before and after images.

7.2.3.3 Survey design

Good question design for usability surveys has many potential pitfalls and issues. Here is a list of good design practices and pitfalls, [QUE]

- A questionnaire tells you only the user's reaction as the user perceives the situation. Thus some kinds of questions, for instance, to do with time measurement or frequency of event occurrence, are not usually reliably answered in questionnaires. Therefore, although useful, it may not be a good idea to ask a user to report how long they spent on each task.
- Open ended questionnaires are good if you are in an exploratory phase of your research or you are looking for some very specific comments or answers that can't be summarised in a numeric code.
- Only use clear, well formulated questions

7.2.3.4 Questions

With this in mind, and considering the design aims, we decided on the following set of questions.

We wanted to find out what the users experience with the command line console was. We asked these questions at the start of the survey.

- I know what a UNIX command line console is. (Yes or No)
- I have used the command line console to copy files and delete files. (Yes or No)
- I have used the command line console to build a 'pipeline'. (Yes or No)
- I have used the UNIX program 'tee' in the command line console. (Yes or No)

For each task we asked:

- I successfully got the correct result from this task. (Yes or No)

- If the answer is no, explain which step gave difficulty. (Open ended)
- What problems or annoyances were there while completing this task? (Open ended)

These questions aim to give maximum insight into design issues that can be improved and information for design decisions. They are also a useful evaluation tool since we can see how many of the tasks the user was able to complete and therefore whether our system is usable for getting tasks done.

We also asked a set of questions once the user had completed all the tasks.

- Which task did you find the hardest? (1, 2, 3, 4, 5, 6 or Unsure)
- Why did you find that task the hardest? (Open ended)
- Which aspects, not mentioned in your answer to the above question, were the most difficult to use, or the most damaging to the user experience? (Open ended)
- If you were to redesign Kevlar, what would you like to add or change? (Open ended)

These questions focus on discovering design issues that can be improved.

Finally, we asked users to give their opinion on a set of statements by choosing one of, 'Strongly agree', 'Agree', 'Neutral', 'Disagree', 'Strongly disagree'.

- "It is easy for a new user to get used to the system."
- "The performance of the system was acceptable. (ie. It ran fast enough.)"
- "The user interface was easy to use. The controls (eg. text box, list box) were easy to use."
- "I feel the Kevlar's system is an improvement over the classic command line. It is more intuitive."
- "I would consider using Kevlar instead of the normal command line."

For the last two statements, we also gave the user an option of choosing 'I have not used the command line before or do not know what it is' instead of stating their agreement.

For the semantic differential type questions (with options between Strongly agree and Strongly disagree), there is a known problem relating to when a user is biased towards clicking the options that line up in a column. To combat this problem, the options were placed vertically instead of horizontally on the HTML form. There is now no problem with options lining up.

7.2.3.5 Result collection

A website was prepared that contained instructions on how to install Kevlar, the list of tasks to perform, and the set of survey questions. The results of the survey question are then saved to an XML file which we can analyse as results come in.

Group Usability Study Website. <`http://www.doc.ic.ac.uk/project/2004/362/g04362341M/study/`>

7.2.4 Results

We received feedback from 7 participants who had all used the console and pipelines before according to their answers to the first 4 questions. Since this number is small and strongly biased towards a particular user-group, the non-open ended questions could not be analysed. However, we have still found the information from the open ended questions invaluable in finding usability issues that can be improved, in helping shape the design of Kevlar, and in assessment of how effective Kevlar is as a tool. Additionally, we unexpectedly found the study to be useful for getting ideas for future extension of Kevlar and finding bugs.

The full set of results can be found in Section 10.3

We were able to address nearly all the usability issues introduced by these comments, and when a response was unclear, we asked the user to clarify.

In total, 54 design issues, bugs or suggestions were raised by the respondents.

7.2.4.1 Summary of task completion

In total, 7 participants attempted 6 tasks. 32 of 42 tasks were completed. Only 1 of the 7 respondents successfully completed task 6. This was because there was a bug in the installer version of a program needed for the task which we only discovered towards the end of the usability study. However, all respondents who failed task 6. were able to build a pipeline for the task, but were unable to get the expected results due to the buggy program. If we ignore this task, 32 out of 35 tasks were completed successfully.

7.2.5 Our Response

We were able to use the constructive criticism given by users in response to the usability study to improve issues with the Kevlar interface. An outline of these is below:

- **"No feedback that the pipeline was saved".** In the beta release provided for the usability study, the save pane was not refreshed when the user saved a pipeline, so the directory that it was saved to was updated to show the newly saved file.

  This was solved initially by refreshing the pane, but because the save file was sometimes 'scrolled' off the bottom of the pane, it was instead decided to additionally close the pane in response to a successful save.

- **"If you RIGHT-click on the program, help should appear".** In the beta release, program help was initiated by clicking on a help button in the header of each program. This was inconsistent with the right-click method of accessing help for arguments, IO nodes and programs in the task pane.

  This was solved by removing the button, and migrating the help-popup behaviour to whenever the user right-clicked on the program header.

- **"do NOT make us click on the circle to select it, clicking anywhere on the name should suffice".** In the beta release, pipes were added between programs by clicking on the circular node widget which was part of each programs node entry list. This was a very small target to click on and some users found this frustrating.

  This was solved by extending the clickable region to include the complete node entry.

7.3 System Evaluation

In this section we are going to evaluate Kevlar with respect to the design and implementation.

7.3.1 Testing

As part of the code development for this project, we wanted to ensure that components developed individually by each group member fulfilled their contracts, and would therefore work together when integrated into the whole system. We wanted a way of quantifying our evaluation of the system, and testing provided a way of achieving this. We divided our testing strategy into two sections as described below.

7.3.1.1 Unit Testing

Unit tests provide a definite pass or fail result for each unit of functionality tested. Early on in the project, we wanted some way to evaluate definite progress, and the pass-fail ratio provided a quick way of obtaining this after every change. It was the responsibility of each member to write tests for their code during the first stages of implementation so that we had some level of quality control.

Our design consisted of three layers which were developed at different rates, which meant testing the system as a whole was unachievable early on. However, rather than leave testing until each layer had

progressed enough to be integrated with its neighbours, we used unit testing to ensure that each component met its requirements, and so would provide the necessary functionality.

Unit testing also allowed us to make changes to the source tree and check that contracts of other components had not been broken. Further to this, we wrote unit tests to quickly reproduce bugs that had been found and used these tests to ensure that the bugs were fixed and remained so throughout development.

7.3.1.2 Continuous Integration

As already stated in the build system (see Section 2.1) we used CVS and Ant build scripts which allowed the whole product to be continually integrated. Since the results of most GUI methods were changes to the visual output of the program, these methods were difficult to unit test. Instead we relied upon developer-oriented integration testing to ensure that the GUI functioned correctly. As the GUI was able to use more and more features from the lower layers, we were able to test that all the components required for particular tasks worked together to fulfil their joint contract.

If during the testing of a task a bug was found, we were able to use Eclipse's debugging capabilities and Java's stack traces to identify the point of failure and discuss the best course of action with the group member responsible. When the bug was resolved, it could be instantly shared with all group members so that they could verify that the problem had been resolved. This peer checking for resolution of bugs provided another level of quality control, and since a checklist of bugs was available to every group member, we had another way of evaluating project progress.

7.3.2 Keyboard Model Evaluation

Although our specifications document only stated that we had to provide a keyboard model, we recognised the need for rapid access to commonly used features in order to attract experienced users of existing shells. In order to evaluate how successfully this was achieved, we decided that keystroke-level analysis was a fair way of comparing Kevlar to existing shells. We took several use cases and generated semantically identical pipelines to fulfil them using both Kevlar and the existing Linux console. We then compared the number of keystrokes required for each, assuming that an experienced user was using both systems, and would therefore know the names of the programs required (elminating the need for searching or using help).

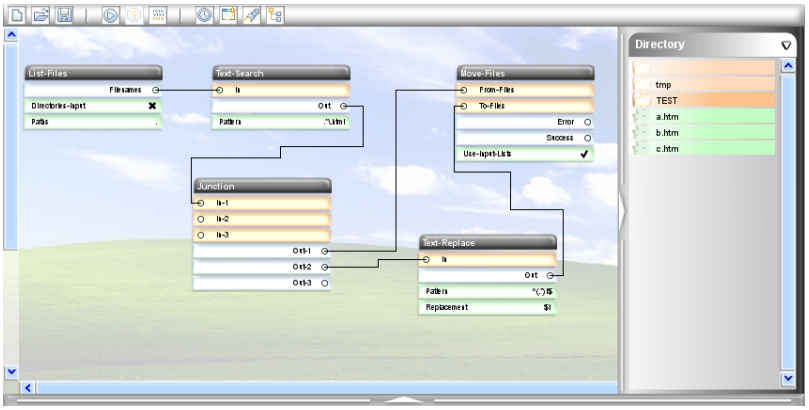
Since in both the Linux console and in Kevlar, the number of keys required to locate a particular program depends on the number of programs in the system (due to tab-completion and auto-complete), we decided to fix the number of programs such that on average three keypresses was required to identify any particular program. We also counted keystrokes for the Linux command to include the typically-added space between program names. By way of example, two such comparisons between the two systems are supplied below.

- **Rename any .html file in the current directory to .htm.**

  Linux pipeline: ls *.html | sed 's/\(.*\.htm\)l/\1/' | xargs -ri mv '{}'l '{}'

  Number of keystrokes: 61

  Kevlar pipeline:



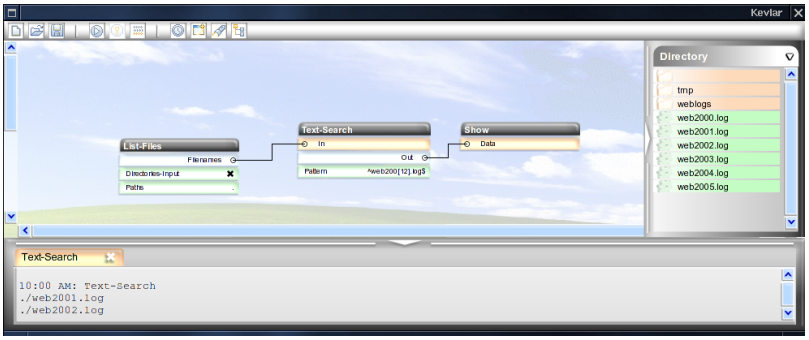
Number of keystrokes: 65

- **Find all log files written in 2001 or 2002.**

  Linux pipeline: ls | grep -E "^web-200[12].log$"

  Number of keystrokes: 32

  Kevlar pipeline:



Number of keystrokes: 40

Overall, we found Kevlar pipelines to take within 50% more keystrokes compared to the equivalent Linux commands. Although Kevlar generally took more keypresses to complete each task, the cause of this was often the longer, user-friendly names we had used for programs which would always require four keystokes (three letters and a tab) to locate, compared to short names such as 'ls' and 'rm' which we took as only requiring two. Kevlar's feature of allowing programs to have multiple input and output nodes also caused further keypresses, since the user had to locate the node on each program by name in order to pipe data to other programs, where this could be done using the single pipe shell directive in Linux.

7.3.3 Areas of Improvement

Generally we are happy with the design and implementation of Kevlar, however during development we realised that there were potentially better ways of implementing some features which we did not have time to explore. A few of these are summarised below.

- **Optimised Drawing Engine.** In order to improve performance, we noticed a number of optimisations that could be made to the existing drawing engine. We performed a spike on a separate branch of the CVS in order to investigate these optimisations, and managed to achieve a significant performance improvement, but at the cost of introducing a number of unexplained visual glitches. Unfortunately, being near the project deadline, we decided not to attempt to resolve these issues, since there was no indication of what was causing them and therefore how long they would take to fix.
- **Execution.** A major inefficiency with execution is the time required for the Java Virtual Machine to start up for every program in the pipeline. We discovered that the latest Java Virtual Machines

support an option to create an image of the startup memory footprint which can be saved and then reused by other JVMs to decrease the start up time. However, because the execution system was not started until mid-way through the project and suffered from unforseen complications, this was an optional feature that became low priority.

- **Argument Visibility Issues.** During the original design of the graphical interface, we wanted to hide any unused arguments in order to conserve screen space, however in the released system this means that arguments must be explicitly added before they can be used. A better system would have been to hide unused arguments only until a program was selected, when it would expand to show all its entries. This would allow us to optimise the keyboard system further by removing the entry selection auto-complete shortening the the number of keypresses required to select entries for certain operations.

Chapter 8

Conclusion

8.1 Our Achievements

At the start of this project, we identified a common frustation within popular command shells which we attributed to the fact that they are mainly based upon historical products which do not use modern capabilities such as a graphical user interface. We then set about quantifying the precise problems that we had with these shells and produced a list of seven usability issues that we wanted to be remedied (see Section 1.1).

Taking these requirements, we looked for existing projects that attempt to find solutions to these problems. We discovered that, although there had been attempts to solve some of these issues by different projects, none had successfully achieved them all (see Section 1.3). We then set about designing and implementing "A User-Friendly, Type-Safe, Graphical Shell", which we called "Kevlar".

Three months later, after the research, requirements gathering, design and implementation phases were complete, we set about evaluating whether our project had fulfilled the principle aims set out at the start. Our internal testing showed we had a product that was functional, stable and met internal quality requirements (see Section 7.3.1).

With a stable beta release now possible, we constructed a usability test and advertised this to external users in order to gain feedback about our project from its use in the real world. The responses elicited from this study demonstrated that our system is usable, but also raised novel issues that we had not considered. We were then able to respond to some of these issues to further improve Kevlar, with any unresolved problems mentioned in this report (see Section 7.2).

Our evaluation showed we have a functional and usable system that fulfilled the seven aims set out at the start of the project. Due to our rapid design process, the Kevlar memory footprint and processor requirements for a consistently responsive system were too great for it to gain widespread acceptance. However, positive results from our usability study suggested that the existing system did have a niche user base.

Kevlar has shown that the user-orientated, graphical, workflow based shell concept can work, and provides the starting point for the production of a universally accepted product.

8.2 Group Conclusions

Working on a project for three months as a group has allowed us to gain an insight into working with other people on a large and complicated software product for a sustained length of time. In this section we will reflect upon what we have taken away from this experience as lessons learnt.

The single most important factor to the success of this project has been good group communication. Communication was no only through our regular formal weekly group meetings, but also through daily contact, discussing what had been done, what was to be done, what was found challenging and other pertinent matters. When people were not around (for example over the holidays) the use of a group

forum, internet-based instant messaging and the telephone were still used to maintain the cohesion of the group. On its own communication did not solve or create any problems, but the fact that it was constant made it much easier to design interfaces that people would be working to, pass on bug-reports, and keep other people appraised of when which features were likely to be needed. In a more formal way, our group milestones helped communicate to everyone the overall view for were the project should be at particular dates.

Communication of progress was also made through the use of definite unit tests that passed or failed, to highlight areas of code that needed work, and areas of code that we tested to show the meeting of their specifications. When integration testing occured, bug-lists were kept on the group forum, and the fixing of these communicated to the group further progress.

A third way progress was communicated was through the use of CVS, and regular committal of work by the group. This allowed us to see and test the integration of Kevlar, and its development towards the final product. Any small bug fixes or new features were instantly available to all other group members for review and appraisal.

8.3 Future Work

8.3.1 Short Term

We recognise that with more time there are many additional features we would like to have implemented that were not critical to the success of the project but are the natural extensions to what we have done.

- **Plug-in-able Visualisers.** Currently Kevlar has two built-in visualisers for displaying text and images from the pipeline. However, because types are dynamically loaded, an obvious extension would be the dynamic loading of visualisers for these types into the Kevlar system. This would enhance the user experience by increasing Kevlar's usability to adapt to fulfil tasks not intended in the original design, e.g. the ability to play sound and video files.
- **Scripts and Control Structures.** One powerful feature of existing shells is the ability to script multiple pipelines and conditionally link them based upon their success or failure. Although we identified this as an optional requirement at the start of the project, we recognised that it was outside of the scope of what we had time to do. With the core of single-pipeline construction and execution done, the next logical step is to add the ability to link pipelines and therefore create scripts.
- **Complete the Macro Model.** Although early on we planned to implement a more complete macro system for Kevlar, we noticed a number of problems in their design that would have taken too long to resolve for our purposes. Currently, the macro implementation does not propagate any arguments from the internal programs up to the macro level. This is because arguments are allowed to change the IO function of a program, and so altering a macro argument could break its internal pipeline. We therefore needed some method of either restricting the argument changes so that the internal pipeline could not be broken or notifying the user of these problems and providing functionality to allow them to be fixed by re-expanding the macro.

  Furthermore, since macros often represent commonly used sub-pipelines that are duplicated within larger pipeline structures, if a macro is expanded and interally edited there is the additional question of whether these changes should be propagated to any other instances of this macro. However, despite this functionality being quite useful, we would have to distinguish an expanded macro from the rest of the pipeline in order to determine whether that macro was being edited or a completely new macro was being constructed.
- **More Help.** Currently, Kevlar provides consistent help for programs, arguments and IO nodes, which is exported by the programs themselves. However, it does not contain any help on how to use the Kevlar system itself more efficiently. This could be provided through tool-tips to suggest keyboard shortcuts for equivalent mouse-based actions and also tutorials that could explain the fundamentals of the system to a novice user and also how to use the keyboard model effectively to a more advanced mouse user.

Kevlar also provides a sophisticated search facility for program discovery based upon keywords that are exported in program meta-information. However, a further form of program discovery could be provided by allowing the user to search for programs that have a particular IO function. For instance, if the user has a program that currently outputs filenames, and wants to convert these to images, this could be used to search for a list of programs that contain this IO function in order to locate the require Image-Loader program.

8.3.2 Long Term

As outlined in our research section, Kevlar is one of several existing work flow tools. What sets Kevlar apart from most of these products is its design centered around usability and its flexibility to be completely extensible to contain any kind of type and program that the user wants. If execution was optimised, Kevlar or a successor could become the generalisation of all workflow tools. If a secure way of allowing programs to customise their visual representation within the Kevlar shell could be found, we begin paving the way to integrating complete graphical applications into our system, rendering any other form of desktop or command-line obsolete. Eventually this could become a usability-centered workflow-based operating system.

Another exciting possibility for the Kevlar system would be to allow discovery of program descriptors from remote machines and remote execution of programs. Couple this with the operating system endpoint of the project mentioned above and you have a fully distributed and intuitive operating system.

Chapter 9

User Guide

This manual provides an overview of how to use the Kevlar system.

9.1 Your First PipeLine

This section will take you step by step through construction of a simple pipeline, which lists the files in your current directory.

9.1.1 Getting started

First you need to get programs that will perform the desired task for you. In Kevlar, there are three ways in which you can access programs (see sections corresponding to each method for more details):

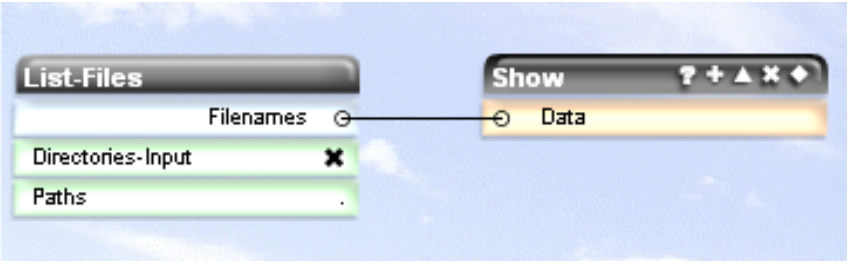
- **Using Programs pane.** You can use Programs pane to locate programs and to drag them from the pane onto the canvas.
- Using keyboard You can use keyboard to type the name of the program you want to use and it will appear on the canvas.
- Using Search pane You can use Search pane to search for programs and then use the results returned.

Let's use Programs pane to get your first program. Open Programs pane by either clicking on the arrow at the right side of the canvas or by pressing Ctrl+P. Next step is to click on "FileUtils" and to locate "List-Files" program. Now click and drag "List-Files" onto the canvas - you have your first program successfully added to the pipeline!

What you need to do now is to connect "List-Files" program to the program that will show you the outputed result. Kevlar has a build-in program called "Show" that does exactly what we want - displays the results of executing the pipeline.

If you click in the middle of the canvas and type 's'. This will bring up a list of all programs available in Kevlar that start with letter s. At this point you can either navigate down until you find "show" and press enter or you can type 'h', which will result in only "Show" being left in the list now, so you can press enter to select it. The "Show" program has now been added to the canvas.

Next step is to connect "List-Files" to "Show". To do that, simply click "Filenames" node in "List-Files" and then click on the "Data" node of the "Show". The pipeline will appear between the two programs. At this point your pipeline should look like this:



To execute a pipeline, click on the execute button on the toolbar or press Ctrl+Enter. When the arrow at the bottom of the canvas turns orange, indicating that pipeline has finished running, click on it to expand an output pane which will display pipeline's output.

9.1.2 Constructing more complex pipelines.

When constructing more complex pipelines, you can greatly benefit from using Help system available in Kevlar. To activate Help, simply right-click on any program, argument or IO node that you want to learn more about and the Help window will popup next to it explaining how and what it is used for.

Another feature of Kevlar that you will find very useful is that you can recognize that your pipeline has errors during the construction stage. All invalid entries in your pipeline are highlighted by being displayed in red. If such entries exist in your pipeline, you will not be able to execute your pipeline, which is also indicated to you by showing that the execute button is disabled.

9.2 Toolbar

This section describes what the toolbar is used for, how to use it and what the toolbar buttons are for.

9.2.1 Toolbar buttons overview

The toolbar is used mostly as means of quickly accessing Kevlar's task panes and to execute constructed pipelines when using the mouse.

To activate any of the toolbar buttons simply point your mouse pointer to the icon, corresponding to the action you with to perform and left-click on it.

Figure 9.1 . Kevlar's toolbar

The toolbar contains the following buttons (left to right on the diagram above):

- **Clear the canvas.** If this button is pressed all the programs and pipes that are currently on the canvas will be removed without being saved. This option can be used if you want to start construction of a new pipeline and do not wish to use the programs you have placed on the canvas during the construction of the previous pipeline.
- **Load.** If this button is pressed the Load Pipelines task pane becomes visible on the right hand side of the canvas. If you had a different pane open before pressing this button it will switch to Load Pipelines view. (See Load pane section for more information).
- **Save.** If this button is pressed the task pane will switch to or open a Save Pipelines view. Like with the load button, if you had a different task pane view open it will be switched to Save Pipelines view.(See Save pane section for more information).
- **Execute.** Pressing this button will cause the pipeline you've constructed to execute. When execution is finished it will be indicated by the orange arrow at the bottom of the canvas. Pressing that

arrow will expand the show pane, which will contain the result outputed by your pipeline. However, if your pipeline contain error, pressing execute button will not run your pipeline (execute button will be disabled).

- **Stop execution.** If this button is pressed during an execution ot the pipeline, the execution wiil stop.
- **Show pane.** Pressing this button with slide the show pane in or out.
- **History.** This button switches the task pane view to the History view, which contains history of the pipelines executed in the current session. (See History pane section for more information).
- **Programs.** This button will switch the task pane view to the Program view. The Program view allows you to see the programs available in Kevlar organised in the directories according to their functionality. (See Programs pane section for more information).
- **Search.** Pressing this button will switch the task pane view to the Search view. This view can be used for the keyword search for programs available in Kevlar. (See Search pane section for more information).
- **Directory.** This button will swith the task pane view into a Directory view, which can be used to browse the directories available on your computer and also to view the content of directories. (See Directory pane section for more information).

9.3 Task panes overview

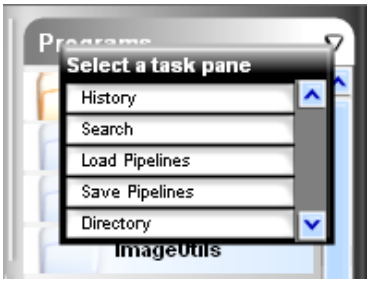
This section introduces task panes that Kevlar uses. Task panes are used to assist you in performing operations such as saving or loading the pipelines you have constructed or searching for the right program to use.

When you start Kevlar, none of the task panes are visible so you can have as much of the canvas space for pipeline construction as possible. However, if you want to use features task panes have to offer, you can open them. An opened task pane is displayed at the right side of the canvas. There are three ways in which you can open and close task panes:

- **Using expand button.** Task pane view can be opend by clicking on an arrow expand button at the right side of the canvas. The task pane slides out showing the Programs pane.
- **Using keyboard shorcuts.** Each task pane view has a keyboard shortcut that can be used to access it. Using those shortcuts will open the task pane in the view the shortcut is assocciated with. (See Keyboard Shortcuts section for more information.)
- **Using toolbar.** The toolbar can also be used to access task panes by clicking on the icon corresponding to the task pane view you want to open. More on this in Toolbar overview section.

If you want to hide the task pane, just click the expand button or press the same keyboard shortcut again.

Once you have task pane view open, you might want to switch between panes. This can be done just be using toolbar buttons or keyboard shorcuts. However, you can also switch between task panes by clicking on the arrow button at the top right corner of the your current task pane. This will bring up a drop down list of all the task panes available in Kevlar.



To switch to a different task pane, just select it from the list and click on it. The task pane will switch to the view you have selected.

9.3.1 Programs pane

Programs pane can be used to access Kevlar's programs, which are organized into directories depending on the program's functionality. The Programs pane shows a directory tree of Kevlar's programs.

To access a program, go into a directory containing the program you are interested in by clicking on it, select the program you wanted to get, click on it and drag it onto the canvas. The program will appear on the canvas, showing its arguments and IO nodes ready for you to use.

9.4 Search Pane

Search pane is a very useful tool in finding which program to use in your pipeline if you don't know its name but know what functionality you want the program to achieve.

9.4.1 Using Search pane

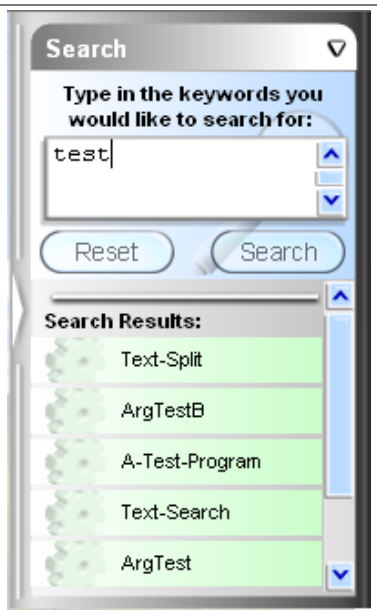
Once you opened a Search pane you can start using it. All you have to do is to type in the keywords you want to search for and press search button (or return). If the search words you've entered don't match any of the programs' keywords the search results will be empty. For example, if you try searching for "hello", the search will return nothing and your Search pane will look like this:

Figure 9.2 . Search pane when nothing have matched the keywords entered.

However, if the search has found some matching program entries for the keywords you have entered, the list of results wiil be displayed on the Search pane. For instance, if you search for "test", the search will return a list of programs that match your keyword and the Search pane will look something like:

If the search has been successful and a list of programs is returned, you can use those programs by simply dragging them onto the canvas. However, if you are still not sure which program to use you might want to use Kevlar's help system. Just right click on the program you wish to find out more about and the help will appear describing the program selected.

Figure 9.3 . Search pane when a list of entries matching entered keyword has been returned.



9.5 Directory Pane

This section describes why Kevlar has Directory pane and how to use it.

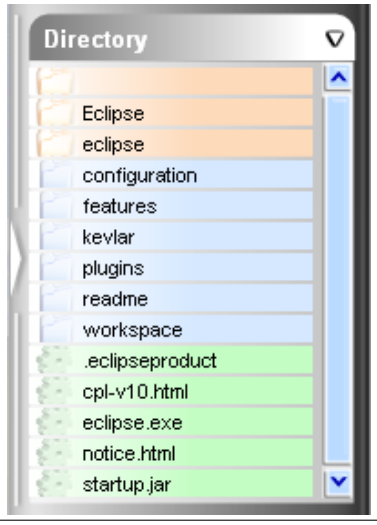
9.5.1 Motivation behind the Directory pane

Directory pane can be used to navigate and browse the directories available on your computer. It is also a quick way to perform operations of changing the directory and listing its contents, which are very often performed in a shell.

9.5.2 Using the Directory pane

First of all, open the Directory pane view if you don't have it already open. Your Kevlar should look something like that:

Figure 9.4 . A screen shot of a directory pane.



All entries in the Directory pane are colour coded to make it easier for you to navigate through your directory tree. Orange entires represent parent direcotries, blue entries represent sub-directories and green entries - files within the directory you are currently in.

To navigate through directories, simly click on the directory you want to go to. The Directory pane will refresh showing you the contents of the direcotory you clicked on and also all the parents of that directory.

A word of warning: navigating through directories using Directory pane changes your current working directory, which may affect execution of your pipelines and therefore the results returned!

9.6 Save Pane

Save pane is used to save constructed pipelines onto disk.

To save a pipeline to disk you need to perform the following operations. First of all open task pane in save pipelines view. Next enter the name of the file you want to save your pipeline in into the textbox. Now, select a directory you want your file to be save into from a directory tree and click "Save".

If saving of your pipeline was successful, the saving pane will close. However, if a problem was encoutered with your save operation, the save pane will stay open and the reason for saving failure will be displayed at the top of the panel.

9.7 Load Pane

Load pane is used to load previously saved pipelines onto the canvas.

To load a pipeline from disk you have to open a task pane in load pipeline view, navigate to the directory that contains the file you want to load. Those files are represented as thumbnails of pipelines that are saved within them. In order to load a pipeline, all you have to do is to click on the thumbnail showing the pipeline you want to load.

The loaded pipeline will appear on the canvas. Loaded pipeline can be manipulated in the same way as any other pipeline.

9.8 History Pane

History pane is used to keep a list of pipelines executed in the current session. Every time you press execute button or a key shortcut to execute a pipeline, it is added to the History pane. The motivation behind History pane is that you can store and load pipelines that you have constructed in this session to/from the pane without saving them to disk.

9.8.1 Using the History pane

If you have just started Kevlar and haven't executed any pipelines yet and opened History pane it will be empty and you will see something like this:

If we now execute a pipeline, lets say List-Files, the History pane will contain a thumbnail of the pipeline just executed:

The more pipelines you execute, the more thumbnails the history pane will show. To use previously executed pipelines, just click on the thumbnail of the pipeline you want to use and it will appear on the canvas ready for you to use again.

Figure 9.5 . History pane when no pipelines have been executed yet

9.9 Keyboard Shortcuts

All of Kevlar's functionality can be accessed through the use of keyboard shortcuts. This section aims to explain how shortcuts can be used to construct and execute pipelines as well as access features provided by the task pane and show pane.

9.9.1 General Shortcuts

The following shortcut keys can be used at all points in the program.

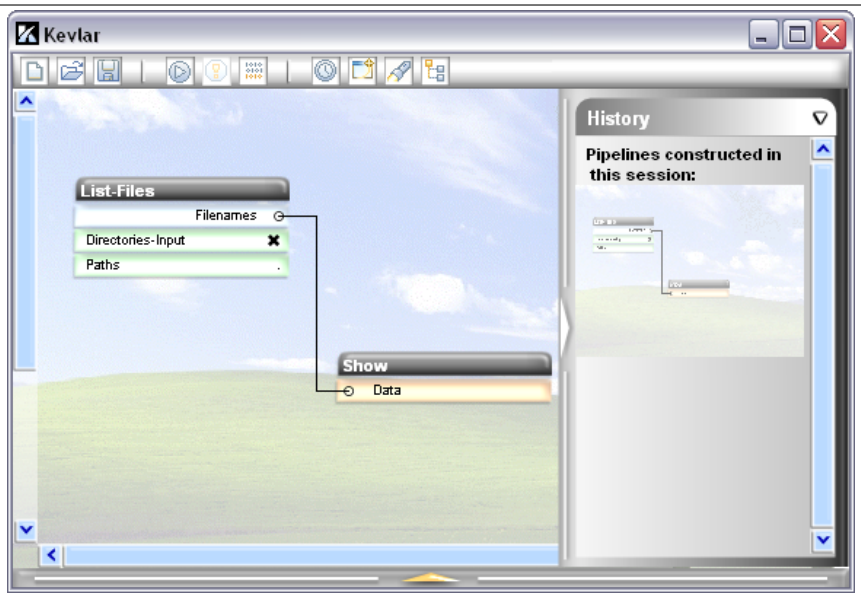
Key combination	Action
Ctrl+Enter	Executes the current pipeline
Ctrl+T	Terminates execution of the current pipeline
Ctrl+N	Starts a new pipeline (clears existing programs and pipes)
Ctrl+A	Selects all programs
Ctrl+O	Shows/hides the output pane

9.9.2 Pipeline Construction Shortcuts

Note: To construct a pipeline using the keyboard, you must first make sure that no task panes currently have keyboard focus by pressing the **Escape** key.

- **Add a program to the pipeline.** If you don't know the name of the program you want to add, you may wish to search for it using the search pane (see Section 9.9.5). Otherwise, bring up the program auto-complete box by typing the first few letters of the program's name. A list of available programs beginning with those letters should appear. You can narrow down the list by typing more letters, expand the list by deleting some letters (pressing **Backspace**), or highlight the program you want by using the **Up** or **Down** cursor keys. When the correct program is highlighted, you can add it by pressing **Enter**. The pipeline view should scroll to where the program is placed, and the program should be *selected*. If you want to add multiple programs in one go, you can use **Spacebar** to select a program, which will add it without selecting it (so that you can start adding another program straight away).

Figure 9.6 . History pane after executing one pipeline



- **Edit an argument of a program.** When a program is *selected*, you can edit its arguments by typing the first few letters of the name of the argument you want to alter. This should bring up the program entry auto-complete box, which will contain a list of the program's arguments and nodes that begin with those letters. Use the **Up** and **Down** cursor keys to select the argument you want to edit, and press either **Enter** or **Spacebar**.

  If you're editing a *boolean* value, then selecting the argument will automatically invert the value and return you to the selected program.

  If you're editing a *set* value, then a drop-down box will appear allowing you to change the value. Use the **Up** and **Down** cursor keys to choose the value you want, then press **Enter** or **Escape** to apply the changes and go back to the selected program.

  If you're editing a *single-line text* value, then a text box will appear allowing you to edit the value. You can use all the standard editing keys to edit the text in the text box, then press either **Enter** or **Escape** to apply the changes and go back to the selected program.

  If you're editing a *multi-line text* value, then a multi-line text box will appear, allowing you to edit the value. You can use all the standard editing keys to edit the text in the text box, however pressing **Enter** will insert a newline into the text rather than applying the changes. To apply the changes and go back to the selected program, press the **Escape** key.

- **Select a node of a program.** With the program selected, start typing the first few letters of the name of the node you wish to select. The program entry autocomplete box should appear listing all of the program's arguments and nodes that begin with those letters. Choose the node that you want to select and press **Space** to select the node and return to the selected program. Alternatively, pressing **Enter** will select the node and deselect the current program. If an input-output node pair is selected, a pipe will be connected between them and both nodes will be deselected.

- **Remove a pipe.** To remove a pipe, follow the steps for selecting a node of a program to select one of the pipe's end nodes. When the node is selected, press **Delete** or **Backspace** to remove the pipe.

- **Other program commands.** There are a number of other keyboard shortcuts that can only be used when a program is selected:

Key combination	Action
Up/Down/Left/Right	Move the selected program up/down/left/right
Delete/Backspace	Removes the selected program
Ctrl+E	Expands/contracts the selected program
Ctrl+H	Brings up help on the selected program
Escape	Deselects the current program

9.9.3 Navigating through a Pipeline

While constructing a pipeline, you may want to select existing programs quickly. Kevlar provides two ways of doing this using the keyboard.

- **Tab Traversal.** Pressing the **Tab** key while a program is selected will deselect it and select the next program right of it on the same row. If the currently selected program is the right-most on that row, the next selected will be the left-most program on the row below. Once all programs have been traversed, the top-left most program will be selected.

- **Find-As-You-Type.** Pressing **Ctrl+F** will bring up the find-as-you-type box, which is used to narrow down the programs that you tab between using tab traversal. Start by typing the first letters of the program you want to select. As you type, all programs that do not start with those letters will become *disabled*. If you now press the **Tab** key, you will only select between *enabled* programs, allowing you to select the program you want more quickly. When you've selected the program you want, press **Escape** to close the find-as-you-type box and re-enable all the programs.

9.9.4 Multi-Select Shortcuts

This section explains how to use keyboard shortcuts to select and manipulate groups of programs within a pipeline.

- **Select multiple programs.** To create a multi-selection of programs, start by selecting the first program you wish to add to the multi-select (see Section 9.9.3), then press the **Spacebar** key to create a multi-select around that program. Now use the same pipeline navigation methods to select the next program to be added, and press **Spacebar** to add it to the multi-select. Keep repeating this process until you've added all the programs you want to the multi-select.

- **Multi-Select commands.** When a program multi-select exists, certain commands that previously only applied to the currently selected program now apply to all programs in the multi-select:

Key combination	Action
Up/Down/Left/Right	Move the all selected programs up/down/left/right
Delete/Backspace	Removes all selected programs
Ctrl+E	Expands/contracts all selected programs
Escape	Deselects the multi-select

9.9.5 Task Pane Shortcuts

Keyboard shortcuts also exist for opening and using the functionality of each of the task panes.

- **Opening/Closing a Task Pane.** There are shortcut keys to quickly open any task pane. If you press the shortcut key for a task pane that is already open and has focus, the task pane will contract to the side of the screen.

Key combination	Action
Ctrl+P	Shows/hides the program pane
Ctrl+L	Shows/hides the load pane
Ctrl+S	Shows/hides the save pane
Alt+S	Shows/hides the search pane
Ctrl+H	Shows/hides the history pane

- **Navigating directory/category structures.** On the Program, Load, Save and Directory panes, you can navigate through the directory or category structure by pressing the **Up** and **Down** cursor keys to select a directory or category, and then pressing **Enter** or **Space** to go into that directory or category.
- **Selecting Programs and Pipelines.** On the Program, Load, Save, Search and History panes, once a program or pipeline has been highlighted using the **Up** and **Down** cursor keys, it can be added or loaded by pressing **Space** or **Enter**. If you're adding a program, pressing **Space** will add it but will not select it, whereas pressing **Enter** will add it, select it and contract the task pane.
- **Selecting Task Pane Sections.** On the Save and Search panes, which contain sections for both typing and navigating, you can switch between the two sections by pressing the **Tab** key.

Chapter 10

Appendices

10.1 Type Systems Theory

This section is to be seen as an extention to the explanation of the Type Checker in the Framework chapter. It will clarify some of the vocabulary used in the report as well as give some fundamentals in the field of type theory.

10.1.1 Type Systems

A Type System is a collection of rules that attach semantical constraints to variables of a certain type. In a polymorphic system, one of the constraints are if a type is a subtype of another type in the system. In java, B is a subtype of A if B extends A. B then enherits functions from A. In the Kevlar system, we restrict the type system to checking if type B is of a lower bound to A, meaning that A is an extention to B of some form. What we mean by this is that we check that B is a direct extention of A or an extention of C which is of some form and extention of A. When B is of a lower bound to A we will write this as A $<$ B. These rules will be formulated in form of a so called type tree, which is the discussion point of the next section.

10.1.2 Type tree

As described above we will focus on inheritance properties of types. An easy way to describe these constraints is via a hierarchical structure called a type tree. In a type tree, every node represents a type, and every child node of it represents an immediate subtype. A type tree is a random tree we are familiar with from Computer Science. We call it random as each node contains a random amount of children.

Least Upper Bound. An interesting calculation to make on a type tree is that of the Least Upper Bound. A least upper bound, or LUB, of a set of types contained within a type tree is the most specific ancestor of all the types. By this we deduce two things, first of all the LUB is an ancestor of all the types, ie. there is a path in the tree betwean the LUB and the type in the set. And secondly this ancestor is the most specific one, meaning that its the lowest in the tree of all common ancestors. Notice that in a single inheritance tree the root is the ancestor of all types. This means that for any set of types we calculate the LUB of, the LUB will always exist because the root is always a common ancestor, albeit maybe not the most specific one. The importance of the LUB is that it previdies a way of finding out which type is the most specific and generic type of a set of types, so that we can generalise a set of types with loosing a minimum amount of information (most specific one). This calculation is used a lot by the Kevlar type checker.

10.1.3 Generics

In recent years a new paradigm has emerged in the polymorphic world. Generics are a separate ideology, different from classic Object Orientation. They offer a way to the programmer to specialize a certain class via parameters. The most straightforward example of generics is a List datastructure. In the past when we used a List in java, internally datanodes would be stored as `Objects`. The reason for this is straightforward, as `Object` is the most generic type, ie. the root of java's type tree, it could be used to represent all other objects in java. There was however an annoying implication to this method, whenever we wanted to retreive an object from the list we would first of all have to know what types were actually stored in the list, and secondly we would have to cast back to the type of the object we added (because in the list it is stored as `Object`). A solution to these problems are generics. Instead of storing data as `Object`, we store it as a parametered type, say T. When we are about to use this parametered List, we fill in what type we want this T to represent. For example we could write `List<String>` to represent a List of String. Once the parameter T has been swapped with the actual type provided, we no longer need to worry about casts since the List will actually hold data of type `String` and no longer of type `Object`. Generics have been available in C++ standard for quite a while now, and Java 1.5 is Sun's newest version of the Java language which introduced generics. Kevlar is written entirely in Java 1.5. Additionally Kevlar also provides the user with the choice of parametered (or generic) types as descibed in the Type Checker chapter.

A special feature of the generics system that Kevlar offers to the users is that the user can specify an upper bound on the parameter. This means that when the user decides to use the parametered class, he can only use a limited amount of types as a parameter - meaning only types of a lower bound or equal to the upper bound of the type parameter. This is a feature that greatly eases the work of the program designer, he doesn't have to worry about which types can be used as a parameter as he knows what the upper bound will be of the parameter and can thus specialise the class for a limitted amount of type parameters.

10.2 Usability study tasks

The usability study in Section 7.2 involved designing 6 tasks for new users of Kevlar to undertake to get experience with using all parts of the system. This appendix contains the 6 tasks and the steps involved.

10.2.1 Task 1.

10.2.1.1 Overall aim

Get a list of files in your current working directory. Learn how to navigate in Kevlar and learn the different ways of getting at programs.

10.2.1.2 Steps

First we want to create a pipeline with one program that lists the contents of the current directory.

- Locate the programs pane on the right and open it.
- Click on 'FileUtils'.
- Click and drag 'List-Files' onto the canvas.

Now we want to connect this program up with another program to show the output of 'List-Files'.

- Click in the middle of the screen and type 's'.

This brings up the 'find-programs-as-you-type' window. 'Show' if you now press 'h', 'Show' will be the only program. Press enter to select it.

- Press enter to select 'Show' which will add it to the screen.

- Click the 'Data' node of 'Show' and then click on the 'Filenames' node of 'List-Files'.

A pipe linking the two programs should appear.

- Press the play / execute button on the toolbar
- Or press Ctrl+Enter together to run the pipeline.

10.2.1.3 Correct result

The show pane at the bottom should change to yellow. Click the arrow to expand the window and see the contents of the current directory.

10.2.2 Task 2.

10.2.2.1 Overall aim

To save the pipeline made in task 1.

10.2.2.2 Steps

If you did not complete task 1. successfully, please still attempt this task as though you had.

First, we want to open the 'Save Pipelines' panel.

- With the task pane on the right open, click the small arrowhead at the top right.
- Click 'Save Pipelines'.
- Or, click the save icon in the toolbar.
- Choose a directory to save the pipeline in and remember it.
- Type in a name and click save.

10.2.2.3 Correct result

The file will appear in the directory that was saved to.

10.2.3 Task 3.

10.2.3.1 Overall aim

To load the pipeline saved in task 2.

10.2.3.2 Steps

If you did not complete task 2. successfully you may skip this task

- Click the 'new' button, the first button on the toolbar.
- Open the 'Load pipelines' panel
- Navigate to where you saved the pipeline in task 2.
- Click on the thumbnail.

10.2.3.3 Correct result

The pipeline saved in task 2 will be loaded and the programs will appear on the canvas.

10.2.4 Task 4.

10.2.4.1 Overall aim

To search for programs and use program arguments. We want to list all files that have the letter 'a' in their name.

10.2.4.2 Steps

First we are going to search for a program that filters out text that does not match a given pattern. Under linux, there is another program to do this called "grep".

- Open the 'Search' panel.
- Type in 'grep' and press 'Search'.

One program should appear called 'Text-Search'. If you click on that program, help about it should appear.we want to connect this program up with another program to show the output of 'List-Files'.

- Drag and drop this program onto the canvas
- Add a 'List-Files' program and a 'Show' program to the canvas too.

We want to connect 'List-Files' to 'Text-Search', then 'Text-Search' to 'Show'.

- Connect List-Files's 'Filenames' to Text-Search's 'In'.
- Connect 'Text-Search"s 'Out' to Show's 'Data'.

If you run this pipeline, nothing should appear in the show pane. That is because we need to set a pattern for Text-Search. Note that the pattern '.*a.*' means: Contains an 'a'. '.*' means: match anything.

- # Set the 'pattern' argument of 'Text-Search' to '.*a.*'. Do this by clicking the argument and typing in the box that appears.
- Run the pipeline.

The output panel at the bottom will now list files in the current directory that contain the letter 'a'. Note you can change the current directory using the Directory panel. (Ctrl-D to bring up)

Notice that the output pane tab is called "Text-Search". This is because 'Show' is connected to 'Text-Search'. This could be set to something more useful. To change the name, we need to add an argument to 'Show'.

- Click the black bar at the top of 'Show' to bring up options.
- Click the '+' button on 'Show'.
- Select 'Name' from the list of arguments.
- Change the 'Name' from "Default" to "A Files".
- Run the pipeline

10.2.4.3 Correct result

The output pane contains a tab called "A Files" and lists files with the letter 'a' in them.

10.2.5 Task 5.

10.2.5.1 Overall aim

To use arguments to change nodes of programs. To build a pipeline to list the contents of just the first sub-directories of the current directory.

10.2.5.2 Steps

If you did not successfully complete Task 4. you may skip this task.

Continuing with the pipeline from Task 4...

- Change the 'Directories-Input' argument of 'List-Files' to a tick by clicking it.

Notice a new input node appears on 'List-Files' called 'Directories'.

- Create a second List-Files program.
- Create a connection (A pipe) from the new List-Files program's output into the 'Directories' input of the other List-Files program.
- Run the pipeline

This pipeline works by getting the directories in the current directory, and then using another List-Files to get the files and directories in those directories.

It is even possible to connect List-Files to itself to find all sub-directories.

You can use the 'Directory' pane to set the current directory. Press Ctrl-D to bring this up.

10.2.5.3 Correct result

The output pane contains a tab called "A Files" and lists files with the letter 'a' in them from immediate sub-directories of the current directory.

10.2.6 Task 6.

10.2.6.1 Overall aim

To build an advanced pipeline that takes images in a directory, resizes and rotates them, and shows the before and after images.

10.2.6.2 Steps

Start with a new canvas, by clicking the 'new' button. (First button on the toolbar)

- Press Ctrl-D to bring up the Directory selector.
- Please select a directory on the computer with about 2 to 10 images.

Attempt to build a pipeline that:

- Lists all files in the current directories
- Uses the 'Junction' program to feed the images into two different programs.
- Loads the filenames as images
- Shows the images as they are originally
- Resizes the images to 100 by 100, rotates them by 90 degrees and shows what they would look like.

You will need these programs:

- **List-Files.** Get the filenames of images.
- **Image-Loader.** To convert the filenames into images. It will ignore non image filenames.
- **Junction.** To take the images and allow it to be fed into two other programs.
- **An Image editing program.** To Resize the image, then another one to rotate the image. Please search for a program that can do these tasks.
- **Show.** To display the images as they were before editing, then another to display the images after they have been resized and rotated.
- Run the pipeline.

Note that List-Files should feed into Junction, and Junction should feed into both Show and Image-Edit.

10.2.6.3 Correct result

The last image from the directory will show in both output panel tabs, first as the original image, and resized and rotated in the other tab.

10.3 Usability study results

A Kevlar usability study was initiated, and the following results were collected for the answers to questions. The full set of questions can be found in Section 7.2

10.3.1 Bacground information

10.3.1.1 I know what a unix command line console is.

- **Yes.** 7

  No. 0

10.3.1.2 I have used the command line console to copy files and delete files.

- **Yes.** 7
- **No.** 0

10.3.1.3 I have used the command line console to build a 'pipeline'

- **Yes.** 7
- **No.** 0

10.3.1.4 I have used the unix program 'tee' in the command line console

- **Yes.** 1
- **No.** 6

10.3.2 Task 1.

10.3.2.1 I successfully got the correct result from this task.

- **Yes.** 6
- **No.** 1

10.3.2.2 What problems or annoyances were there while completing this task?

- Help window didn't work properly
- The yellow arrow is maybe too subtle. Running the pipeline takes a few seconds.
- I commonly use the "Enter" key on the numeric keypad rather than the "return" key (they almost always have the same function). However, the "Enter" key does not function in Kevlar, which was momentarily confusing. It might also be better if one could click anywhere in the "Node" elements to join the nodes together, rather than just in the circle?
- The interface of the "find programs as you type" window is not obvious. For example, you cannot use the mouse in selecting an item (e.g. you\'d expect double click to add an item) and you cannot easily tell which is selected when there are only two (although I can see efforts have been made to resolve that by making unselected items smaller)
- only the installation... finally got up and running when [A Kevlar team member] zipped and sent me his installation directory.

10.3.3 Task 2.

10.3.3.1 I successfully got the correct result from this task.

- **Yes.** 6
- **No.** 1

10.3.3.2 If the answer is no, explain which step gave difficulty

- The name I got did not match the filename I inputted. It was prefixed initally with "null", and then on second attempt "programs"

10.3.3.3 What problems or annoyances were there while completing this task?

- No response to clicking Save button to indicate successful save.
- There was no response when clicking on the \'Save\' button to say that the pipeline had been saved. The saved file was called 'nulltest.kvl' although I named it 'test.kvl'.
- No feedback that the pipeline was saved. Again, quite slow.
- There appears to be a bug in the saving process - If I choose the "Kevlar" directory in the save folder selector (the orange thrid orange entry), then enter a filename and click "Save", nothing happens. However, if I choose one of the blue folders (say "programs"), it will save correctly. It would also be useful if there was some feedback to indicate that the save operation succeeded.
- The edit box does not lose focus in the manner you would expect. Pressing the Save button does not confirm anything (one might expect the pane to disappear, for example)

10.3.4 Task 3.

10.3.4.1 I successfully got the correct result from this task.

- **Yes.** 6
- **No.** 1

10.3.4.2 If the answer is no, explain which step gave difficulty

- Nothing happened when pipeline thumbnail was clicked.
- Clicking on the thumbnail didn't to anything.

10.3.4.3 What problems or annoyances were there while completing this task?

- The saved file initially did not appear; had to navigate out of the directory and then back in before it appeared.

10.3.5 Task 4.

10.3.5.1 I successfully got the correct result from this task.

- **Yes.** 7
- **No.** 0

10.3.5.2 What problems or annoyances were there while completing this task?

- There were two text search programs (Text-Search and TextSearch). "Show" program was displayed underneath right hand pane.
- None!
- I got TextSearch and Text-Search. Confusing. To drag Text-Search, i first had to click it because it shows the help which makes dragging impossible. It doesn't happen all the time. Maybe because the program is at times unresponsive? With an empty pattern, it's not clear where to click to enter a new pattern.
- After re-running it, the "A Files" tab did not become selected over "Text-Search", which you might expect. Not a real problem, however.
- if you RIGHT-click on the program, help should appear. oh, do NOT make us click on the circle to select it, clicking anywhere on the name should suffice.

10.3.6 Task 5.

10.3.6.1 I successfully got the correct result from this task.

- **Yes.** 7
- **No.** 0

10.3.6.2 What problems or annoyances were there while completing this task?

- Didn't get correct result first time I clicked run, but did get correct result the second time.
- The edge between List-Files & Text-Search wasn't updated after ticking directories-input.
- The connecting lines get messy when there are about 4 items there. Otherwise, great. (This example really showed the program off well)
-

10.3.7 Task 6.

10.3.7.1 I successfully got the correct result from this task.

- **Yes.** 1
- **No.** 6

10.3.7.2 If the answer is no, explain which step gave difficulty

- When I executed the pipeline, I got the original images to appear, but not the scaled and rotated ones (even when the pipeline had finished executing).
- Resize function wouldn't work; rotate worked on its own, but the image was moved slightly so some parts were missing.
- I created a pipeline "List-Files.Filenames" -> "Image-Loader.Images-Out" -> "Junction.In-1" with "Junction.Out-1" -> "Show.Data" and "Junction.Out-2" -> "ImageEdit.Images-In" -> "Show->Data". The first Show, named "Normal images" showed me my 4 images. The second show, "Resized images", claimed there were no images to show.
- The image editing program did not work. It did, however, work once in the directory of the program on the icons. In all other cases, it failed.

10.3.7.3 What problems or annoyances were there while completing this task?

- The directory scrollbar doesn't behave like a usual scrollbar (you can't page up/down by clicking above/under the slider). Dot files are not hidden.
- Can't change drive in directories pane!

10.3.8 Section

Final Questions

10.3.8.1 Which task did you find the hardest?

- **1.** 1
- **2.** 0
- **3.** 0
- **4.** 0
- **5.** 0
- **6.** 6
- **Unsure.** 0

10.3.8.2 Why did you find that task the hardest?

REFERRING TO TASK 6.

- Lots of programs, no nice 'guide through' steps.
- Because the way to connect different parts isn't so clear. Can you connect image output to a junction for instance?
- The junction concept was not immediately obvious - I had expected some sort of "single in, multiple out" system, whereas it seemed to act as a "multiple in, multiple out" which seemed rather pointless. Why would I used a junction to connect In-1 via Out-1 when I could just connect directly?
- Due to the problems encountered.

REFERRING TO TASK 1.

- It took me some time to figure out how the program works, but after the basic "initiation" I found it very intuitive.

10.3.8.3 Which aspects, not mentioned in your answer to Q2, were the most difficult to use, or the most damaging to the user experience?

- Where programs were placed on screen (some were off screen and some were underneath panes).
- Its's sometimes difficult to know which keypresses do what in certain situations.
- probably the most damaging is the lack of native OS UI idioms. Whilst the interface was attractive in it's way, the widgets were rather small, and did not totally confirm to the experience I\'m used to. Use of native OS widgets would have made me feel more "at home" with the application.
- Load and Save dialogs really need fixing.
- having to click on the small circles and click-drag not working for making pipes
- There is a slight lag on my machine, due to the lack of memory, probably due to the Java overheads. However, my machine is below average by modern standards, so it should be much less of a problem for users with powerful PCs.

10.3.8.4 "It is easy for a new user to get used to the system."

- **Strongly Agree.** 0
- **Agree.** 6
- **Neutral.** 1
- **Disagree.** 0
- **Strongly Disagree.** 0

10.3.8.5 "The performance of the system was acceptable. (ie. It ran fast enough.)"

- **Strongly Agree.** 0
- **Agree.** 3
- **Neutral.** 2
- **Disagree.** 1
- **Strongly Disagree.** 1

10.3.8.6 "The user interface was easy to use. The controls (eg. text box, list box) were easy to use."

- **Strongly Agree.** 1
- **Agree.** 3
- **Neutral.** 2
- **Disagree.** 1
- **Strongly Disagree.** 0

10.3.8.7 "I feel the Kevlar's system is an improvement over the classic command line. It is more intuitive."

- **Strongly Agree.** 2
- **Agree.** 1
- **Neutral.** 1
- **Disagree.** 3
- **Strongly Disagree.** 0

10.3.8.8 "I would consider using Kevlar instead of the normal command line."

- **Strongly Agree.** 1
- **Agree.** 3
- **Neutral.** 0
- **Disagree.** 1
- **Strongly Disagree.** 2

10.3.8.9 If you were to redesign Kevlar, what would you like to add or change?

- Make it much faster (on slower machines).
- Make it run fast =)
- As mentioned, a switch to native UI idioms would be appriciated (also, an MDI interface?). A command syntax for creating pipelines would also be interesting, which would render Kevlar as a "Pipeline visualisation tool", rather than a graphical builder which would be far more useful to me.
- Justification of above responses: After working through this document, the program is very easy to use. A user may not be able to use the program too easily without working through such a document, however. The same is also true (but more so) for standard command lines, however. The performance was unacceptable for small tasks, due to the initial loading time. However for tasks which might take longer to execute the performance is not a big problem (since the execution speed was acceptable, I felt) The user interface varied. The load and save dialogs were badly designed; however other dialogs (the output pane, the search dialog, the file trees and the workspace, for example) were very well designed. For complicated tasks, this is a great improvement over the classic command line. I would use Kevlar for large tasks instead of a normal command line. This would be especially true if there were a way of running KVL files without launching the environment. As for the actual question: I like it how it is. Maybe more help for the programs, instead of just a summary. Also, more programs would be needed to make it useful. As mentioned above, the ability to run files without launching the environment would be good.

- a ”duplicate” method so if something was on the canvas already, I wouldn\'t have to go back to programs to find another one to drag out. bigger hit regions for amking pipes a better mechanism for selecting things from dropdown list (I\'m still not sure what it is)... i.e. setting the angle to 90 after you select it, then what?
- One thing to consider would be a new, Kevlar brand of ”man”, which would instruct the user and provide hints and help. It could be intelligently integrated with the GUI.

10.4 Individual Working Hours

This section lists the weekly-working hours for each member of the group, and a rough outline of what that time was spent doing.

WEEK BEGINNING 10TH OCTOBER

- **Tristan.** 26 hours - Build system, project design and planning
- **Daniel.** 17 hours - Project design and planning, Report 1
- **Marc.** 20 hours - Project Log website, GUI design, Report 1
- **Kate.** 14 hours - GUI design, pipe rendering algorith research
- **Steve.** 21 hours - Layout algorithms, researching similar projects, Type System

WEEK BEGINNING 17TH OCTOBER

- **Tristan.** 34 hours - Build system, program and type loading
- **Daniel.** 34 hours - Report 1, argument model, HIAL, program descriptions
- **Marc.** 17 hours - GUI paint core, planning Drag-And-Drop
- **Kate.** 17 hours - GUI implementation
- **Steve.** 28 hours - Type Checker, Report 1, ProgramRepository

WEEK BEGINNING 24TH OCTOBER

- **Tristan.** 23 hours - Program and type loading, planning execution system
- **Daniel.** 31 hours - Type checking, HIAL, XML description and argument models
- **Marc.** 8 hours - Scrollbars and Scrolling. Enjoying Canada
- **Kate.** 14 hours - GUI coding, Drag-And-Drop
- **Steve.** 23 hours - Testing program loading, Type checking, Framework

WEEK BEGINNING 31ST OCTOBER

- **Tristan.** 15 hours - Bugfixing and execution system
- **Daniel.** 11 hours - DTD for programs, testing XML models, pipeline saving
- **Marc.** 23 hours - Graphical representation of programs and pipes. Using HIAL from GUI for program details.
- **Kate.** 13 hours - Contextual help, bugfixing
- **Steve.** 4 hours - Pipe implementation

WEEK BEGINNING 7TH NOVEMBER

- **Tristan.** 20 hours - Report 2
- **Daniel.** 19 hours - Report 2, HIAL
- **Marc.** 23 hours - Report 2, refactoring out Layout
- **Kate.** 18 hours - Report 2

- **Steve.** 22 hours - Report 2, researching Layout

WEEK BEGINNING 14TH NOVEMBER

- **Tristan.** 7 hours - Execution engine
- **Daniel.** 14 hours - HIAL, Needleman Wunch library code, Show Pane
- **Marc.** 16 hours - Arguments, bugfixing, wrapping SWT controls
- **Kate.** 15 hours - Search pane, toolbar, pane switching
- **Steve.** 9 hours - Layout algorithms

WEEK BEGINNING 21ST NOVEMBER

- **Tristan.** 14 hours - Execution engine, programs, wrapping swing components
- **Daniel.** 21 hours - Show Pane, Needleman Wunch, Framework
- **Marc.** 21 hours - Keyboard input, bugfixing, argument editing
- **Kate.** 22 hours - Thumbnail research, history pane, search pane
- **Steve.** 2 hours - Group meetings

WEEK BEGINNING 28TH NOVEMBER

- **Tristan.** 0 hours - Exam revision
- **Daniel.** 0 hours - Exam revision
- **Marc.** 0 hours - Exam revision
- **Kate.** 0 hours - Exam revision
- **Steve.** 3 hours - Unit testing [thats dedication!]

WEEK BEGINNING 5TH DECEMBER

- **Tristan.** 0 hours - Exam revision
- **Daniel.** 0 hours - Exam revision
- **Marc.** 0 hours - Exam revision
- **Kate.** 0 hours - Exam revision
- **Steve.** 0 hours - Exam revision

WEEK BEGINNING 12TH DECEMBER

- **Tristan.** 2 hours - Group meetings
- **Daniel.** 2 hours - Group meetings
- **Marc.** 2 hours - Group meetings
- **Kate.** 2 hours - Group meetings
- **Steve.** 2 hours - Group meetings

WEEK BEGINNING 19TH DECEMBER

- **Tristan.** 22 hours - Swing wrapping, programs, rewriting execution
- **Daniel.** 23 hours - Show, Kill / Killall in HIAL, Bugfixing, Argument handling in Framework
- **Marc.** 36 hours - DirectoryPane, TextBox, DropDownBox, MultilineTextBox controls, Macros, Feeling Chirstamssy.
- **Kate.** 31 hours - Saving and Loading panes
- **Steve.** 39 hours - Layout algorithms

WEEK BEGINNING 26TH DECEMBER

- **Tristan.** 48 hours - Final report, bugfixing, new programs

- **Daniel.** 34 hours - Final report, usability website, Show
- **Marc.** 37 hours - Final report, keyboard, bugfixes, graphics
- **Kate.** 49 hours - Final report, saving and loading macros
- **Steve.** 36 hours - Final report, usability survey

WEEK BEGINNING 2ND JANUARY

- **Tristan.** 25 hours (specualtive) - Final touches to report, presentation rehearsal
- **Daniel.** 25 hours (specualtive) - Final touches to report, presentation rehearsal
- **Marc.** 25 hours (specualtive) - Final touches to report, presentation rehearsal
- **Kate.** 25 hours (specualtive) - Final touches to report, presentation rehearsal
- **Steve.** 25 hours (specualtive) - Final touches to report, presentation rehearsal

10.5 Group Meetings

We met regularly throughout the course of this project in order to allocate work, check project progress and discuss our project with our supervisor. In this section, we outline the meetings that took place, stating who attended them and a summary of the meeting's minutes. Please note that this is not intended to be a comprehensive list, since there were often informal meetings between the group members where work was discussed, and sometimes sub-groups would meet to talk about their section of the project (i.e the Framework, HIAL and GUI), which is not included below.

- **Group Meeting: 12/10/04 (7:05pm - 8:05pm, Attended by all).** The first group meeting, where we all attended to go through the project outline and discuss what we hope the product should be capable of. Several use-cases for the project were brainstormed and a name for our product was discussed. Various names were proposed (including 'shelltime' and 'pipesafe'), but eventually the group agreed on 'Kevlar - the bullet-proof shell' since it reflected the idea of a type safe shell that would prevent the user from executing incorrect pipelines.
- **Group Meeting: 13/10/04 (12:05pm - 1:03pm, Attended by all).** The agenda for this meeting was to formalise a specification for the first report, due in the following week, and to assign tasks to each group member. We started by analysing the specification report produced by last year's best group, and came up with section headings for ours. We then split our project into three sections; Human Interface, Programs and Types, and the Framework, and began to decide on requirements for each. Group members then volunteered for the sections they wanted to work on; Tristan and Steve were allocated to the Framework, Dan was allocated to Programs and Types, and Marc and Kate were allocated to the Human Interface. We finished by agreeing to work to a set of six milestones to be completed on a weekly-basis.
- **Group Meeting: 13/10/04 (2:53pm - 4:45pm, Attended by all).** Following from our earlier meeting, we decided to sketch out on a whiteboard an overview of the project's architecture. We realised the need for to split the Human Interface section into two; the Human Interface Abstraction Layer (HIAL) and the Graphical User Interface (GUI). We decided that this would allow the GUI section to concentrate solely on displaying information to the user and capturing user events, while the HIAL would pass information between it and the Framework and handle additional, non-graphical tasks such as saving and loading.
- **Supervisor Meeting: 14/10/04 (4:10pm - 5:00pm, Attended by all).** This was the first meeting with our supervisor, and the main agenda items were discussing the requirements we'd thought of in previous group meetings with our supervisor and planning the specification report. We discussed the need to make our shell 'usable' and how we could quantify 'usability'. We decided that regular integration for testing and a usability study would be appropriate. We planned to release a beta version of the project mid-way to the final deadline and gather feedback that could be used to improve usability.

- **Group Meeting: 15/10/04 (4:45pm - 5:12pm, Attended by all).** The main agenda item for this meeting was resolving problems that members had thought of with the project's design. We discussed the need for scripts (which eventually became macros) and the need for temporally-linked programs. We decided that these areas posed many problems that would take a very long time to resolve, so moved them to the optional specifications section of the first report. We also decided on the project's directory structure and discussed how each member would contribute to the specification report document.
- **Group Meeting: 18/10/04 (6:00pm - 6:15pm, Attended by all).** The main agenda item was to discuss work done over the weekend and set milestone tasks for Friday. We decided to finish writing the mains sections of the first report on Wednesday to give us time to proof-read it and make alterations before submission on Friday. We also outlined a set of milestones for friday, which included the ability propagate the program list from the Framework to the GUI, in order to show all sections communicating with one another.
- **Supervisor Meeting: 20/10/04 (3:00pm - 3:41pm, Attended by all).** The main agenda item was going through the specification report with our supervisor, who commented on the viability of the requirements we had suggested. We decided to alter some of our requirements in order to make them more achievable and introduced the usability requirement at our supervisor's request.
- **Supervisor Meeting: 27/10/04 (3:00pm - 3:30pm, Attended by all except Marc).** The main agenda item was discussing project progress with our supervisor and fixing requirements for the second milestone. We also discussed the security model we wanted to implement to ensure that programs were only permitted to perform certain operations. The implementation of the type-checking algorithm was also explained.
- **Group Meeting: 02/11/04 (4:50pm - 5:35pm, Attended by all).** This meeting was predominantly concerned with decided the weekly milestones and discussing what Marc had missed while in Canada. We allocated tasks for each people to complete by Friday; Marc was to complete simple pipelines in the GUI, Kate was to work on the help system for programs, Tristan and Steve were to work on pipeline execution and Dan was to work on saving and loading.
- **Supervisor Meeting: 03/11/04 (3:10pm - 4:10pm, Attended by all).** We demonstrated the project to our supervisor for the first time, showing how the user could browse the available programs and drag one into the pipeline. We discussed how this would be accomplished using the keyboard and also how to prove the soundness of the type-checking algorithm by creating a model in Haskell.
- **Group Meeting: 08/11/04 (12:00pm - 1:00pm, Attended by all).** A meeting to discuss the implementation report due in at the end of the week. The report was split into sections based upon the architecture of the project and the general content of each section was planned and allocated to a group member. We also set the deadline that all sections must be written by Thursday so that it could be proof read and altered if necessary before submission on Friday.
- **Supervisor Meeting: 17/11/04 (2:10pm - 3:00pm, Attended by all).** The main agenda items were the testing of the current system and planning what needs to be implemented before we break for exam revision. We discussed the need to come up with more use-cases that would highlight the programs needed to be implemented for the project demonstration, and would allow us to test out system. Also, since the majority of the Framework and HIAL had been completed, Dan and Steve were assigned to work on GUI components such as the show pane and the layout algorithm.
- **Group Meeting: 23/11/04 (12:00pm - 1:00pm, Attended by all).** The main agenda item was finishing the minimum specifications for internal milestone five and demonstration of project to supervisor on Friday. We highlighted the main aspects of the GUI left to be finished were keyboard input and saving/loading, but also realised the need for programs to demonstrate our system once finished. The need to reassess our break for revision was discussed.
- **Supervisor Meeting: 26/11/04 (10:00am - 11:00am, Attended by all).** We demonstrated the project to our supervisor, although some of the minimum specifications had not been fully finished. Since everyone was getting distracted by exams, we decided to suspend work on the project for three weeks and work over Christmas. We also postponed our beta release for the usability study until after the exam period.
- **Supervisor Meeting: 17/12/04 (2:00pm - 3:00pm, Attended by all except Kate).** Exams were over, so we planned our work over Christmas. Our usability study had been postponed until the fol-

lowing Monday, so we planned what needed to be done over the weekend to finish the minimum requirements and produce a usable beta release. The main points to be done were keyboard input and example programs, which were assigned to Tristan and Marc. Since some group members were going home for Christmas, we planned to discuss the remaining requirements and plan the report online over the holiday.

- **Group Meeting: 03/01/05 (9:00am - 9:00pm, Attended by all).** With all group members now back from home, we met to proof read and edit the report, as well as write the presentation slides for submission on Tuesday. All minimum and extended specifications had been completed, and the usability study had gone online, albeit after being postponed to the 31st December. Responses to the usability questionnaire were written up, discussed and changes were made to the project to address some of the problems highlighted. Tuesday and Wednesday were allocated for further group meetings in order to rehearse the presentation.

10.6 The Group Calendar

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
October 3rd			Group Project Introduction		Submission of Project Choices Form	
October 10th	Allocation to Project	Group Meeting	Group Meeting	Supervisor Meeting	Group Meeting	
October 17th	Group Meeting		Supervisor Meeting		Project Specification Report Submission	
October 24th			Supervisor Meeting			
			Marc Hull In Canada			
October 31st		Group Meeting	Supervisor Meeting			
November 7th	Group Meeting				Project Design and Implementation Report Submission	
November 14th			Supervisor Meeting			
November 21st		Group Meeting			Supervisor Meeting	
						Exam Revision
November 28th			Exam Revision			
December 5th			Exam Revision			
December 12th					Last Day Of Term Group Meeting Supervisor Meeting	
		Exams				
December 19th	First Installer Released for Kevlar					
December 26th					Usability Study Online	
January 2nd	Group Meeting	Final Report Submission		Project Presentation		
		Presentation Rehearsal				

10.7 Classic CVS Commit Comments

During the course of this project, we used CVS as our version control system. As part of the system, developers are encouraged to attach comments to each commit.

Collected below are a select few of these comments.

- Committing whilst drunk - too drunk to think of comment
- Hacky tests
- Macros - they're alive...

And the one we love the most

- *** empty log message ***

Project Bibliography

Cited Resources

[HBK04] *MEProf: Modular Extensible Profiling for Eclipse*, Marc Hull, Olav Backmann, and Paul Kelly, <http://www.doc.ic.ac.uk/~phjk/Publications/MEProf-ETXVancouver2004-PressQuality.pdf> , 2004.

[HS01] *Problems Running Untrusted Services as Java Threads*, Almut Herzog and Nahid Shahmehri, <www.ida.liu.se/~almhe/publications/CSES-2004-Preproceedings.pdf> , 2004.

[IBMEX] *SWT Standard Widget Toolkit - Development Resources*, IBM Corporation, <http://dev.eclipse.org/viewcvs/index.cgi/\%7Echeckout\%7E/platform-swt-home/dev.html#snippets>.

[JISO] *JSR 121: Application Isolation API Specification*, , <http://jcp.org/en/jsr/detail?id=121>.

[MANSEL] *Linux Programmer's Manual, the select man-page*, , 2001-02-09.

[PIPER] *The PIPER Project*, , <http://bioinformatics.org/piper/documentation/scientific-apps.html> <http://bioinformatics.org/piper/screenshots/index.html/> .

[PURSUIT] *The Pursuit Project*, , <http://citeseer.ist.psu.edu/modugno94pursuit.html> <http://citeseer.ist.psu.edu/cache/papers/cs/3447/http:zSzzSzpecan.srv.cs.cmu.eduzSzafszSzcs.cmu.eduzSzprojectzSzgarnetzSzwwwzSzpbd-groupzSzpaperszSzcmu-cs-94-109.pdf/modugno94pursuit.pdf> .

[QUE] *Questionnaires in Usability Engineering*, , <http://www.ucc.ie/hfrg/resources/qfaq1.html> .

[SWTDOC] *Eclipse Platform API Specification: Class Canvas*, IBM Corporation, <http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/org/eclipse/swt/package-summary.html>, 2001.

[VISIQ] *VisiQuest Application*, Accusoft Corporation, <http://www.accusoft.com/imaging/visiquest/visiquest_vis_prg.asp> .

[VUFC] *The VUFC Project*, , <http://www.dmst.aueb.gr/dds/pubs/jrnl/2001-SPANDE-VUFC/html/vufc.html> .

[WIKI] *Wikipedia*, , <http://en.wikipedia.org/wiki/Command_line> .

Other Resources

[1] *Effective Java™ Programming Language Guide*, Josua Block, 0-201-31005-8, Copyright © 2001 Sun Microsystems, Inc, Addison-Wesley.

[2] *Java™ 2 Platform, Standard Edtion, v 1.5.0 API Specification*, , <`http://java.sun.com/j2se/1.5.0/docs/api/`>, 2004.