A User Friendly, Type-Safe, Graphical Shell

Project Design and Implementation Report

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

January 4, 2005

Contents

Chapter 1

Introduction

This is the second of three reports in relation to our 3rd year project *"A User Friendly, Type-Safe, Graphical Shell"*, or *"Kevlar"*, which is the working name given to the system that is now in production.

The traditional command shell, while being an extremely useful tool for power users, presents a difficult interface for the beginner to use and learn. There are several reasons for this, as outlined below:

USABILITY ISSUES WITH EXISTING SHELLS

- **Lack of consistent help.** When learning and using any system, having help available that is relavant 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.
- **Lack of 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.
- **Lack of 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.
- **Lack of pipe semantics.** For beginners, 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].
- **Lack of 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. A two (or higher dimensioned) model/view of the pipelines would be helpful in visualising the processes being run.
- **Lack of 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].

The aim of this project is to address each of the problems above, and in doing so create a new variant of the traditional shell with usability at its center, but to still provide support for the rapid development of program composition that the old shell model supplied.

This report is to give an indication of the progress that has been made during the first half of the coding time on the project, and to highlight any major design feature and/or changes that have taken place.

[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 echos 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/`>

Chapter 2

Project Progress

This section will outline the progress made by the group relative to the specifications that were described in the first report.

Additionally this section will comment upon the first three ”internal milestones”, and outline suggested milestone targets for the following three weeks.

2.1 Specifications

2.1.1 Framework Specifications

MINIMUM (GRADE B)

- **Can connect programs together using pipes between their input and output nodes.** This has been achieved and tested.
- **Pipes should have a flow direction.** This is implicit within the design for the shell system. The implementation of pipes does have a notion of 'flow direction'.
- **Pipes should be type checked.** This has been achieved and tested.
- **Should run valid pipelines.** This is currently in progress.
- **Should handle program crashes.** As the development work on running valid pipelines is in progress, this has yet to be done. However the design now launches programs in external processes, so the shell has a level of protection from crashed programs.
- **Should pick up newly registered types and programs from a central repository.** This has been achieved and tested.

  The natural extension point to this task, which is the removal of registered types and programs at run-time, has been taken into consideration during the design of the Loaders[1]

EXTENDED (GRADE A)

- **Pipes should have the fuctionality of being templated.** Pipes can be templated, and tests of the type checker show they are handled correctly.

2.1.2 GUI Specificaitons

MINIMUM (GRADE B)

- **Should provide a graphical representation of pipelines that is easy to understand.** This has been achieved.

[1] For a complete description of the Loaders component, please see Section 6.2.

- **Should allow the user to construct pipelines using the mouse.** This has been achieved.
- **Should allow the user to construct pipelines using the keyboard.** This is in the design stage.
- **Should gives the user feedback on construction errors.** This is in the planning stage.
- **Should restrict the user's ability to enter incorrect argument values.** This is in the design stage.
- **Should give the user feedback on invalid arguments.** This is in the planning stage.
- **Should visualise certain pipeline output and errors.** This is in the design stage.
- **Should allow the user to save and load pipelines.** This is currently being implemented.
- **Should follow a design process that encourages high usability.** This is ongoing during the development of the GUI.

EXTENDED (GRADE A)

- **Should allow the user to search for programs to achieve a task.** This is currently being implemented.
- **Should provide context-sensitive help based upon the current program, argument or node.** This has been achieved.
- **Should allow for construction of basic macros.** This is in the planning stage.

2.1.3 Program Types and Tools

MINIMUM (GRADE B)

- **Programs can define input and output nodes that are named and typed.** This has been achieved.
- **Programs have arguments, some of which may change its IO function.** This has been achieved in the model.
- **Programs can validate their arguments before runtime.** This is implicit in the design.
- **A core set of programs should be provided to demonstrate the system.** This is to be done.
- **Third parties should be able to write programs for the framework.** The design allows this (the method of writing programs would be the same for third parties as it is for the group).
- **Programs should have human-useable names.** The design allows for programs to have human-usable names as well as internal system names.
- **Types should be in a type hierarchy.** The design enforces this since all types must be derived from `IType`.
- **Users should be able to define their own types.** This has been achieved.
- **A basic set of types should be provided for demonstration.** This is in progress.

EXTENDED (GRADE A)

- **Programs should export help information.** This has been achieved.
- **Programs can accept and produce templated types.** This has been achieved.

2.2 Internal Milestones

As described in the first report, internal milestones have been set on a weekly basis to ensure that the required level of work is done on the project.

Most of the targets specified for the first three milestones have been met. The adopted agile design process evolved the milstones based upon the success of the previous itereation. The milestone targets for the next three weeks are as planned in the first report and are repeated below:

- **12th November.** Submission of Report Two: Project Design And Implementation.
- **20th November.** Internal Milestone Four.

  Programs and pipelines should be able to be executed in the system.
- **27th November.** Internal Milestone Five.

  All Minimum Specifications are met. Progress towards Extended Specifications nearing completion.
- **4th December.** Internal Milestone Six.

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

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

Chapter 3

Development Environment

For the development of this project, it was decided that all group members should have a simple and consistent method of working on code and reports, across any platforms they choose, so that minimum effort is required to work on the project whether at home or at University.

The following sections describe the environment decided upon, and an overview of the 'build-system' that has been written to make conversion from code to jar artifacts as flexible and scalable as necessary.

3.1 Programming Language

The primary language chosen for the development of this project is Java (J2SE 5.0)[1]. This is because it is a language all group members know and are confident with, however the choice to adopt J2SE 5.0 also means the group have the opportunity to learn the new Java features (particularly generics and enumerations).

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

THE JAVA J2SE 5.0 API (AND EXTENSIONS) AND THEIR APPLICATIONS TO 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.** A design change between the first report and the present one means that 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.
- **Security.** A consiquence of the design change mentioned above is the availability of Java's highly customisable security policy mechanism which potentially allows the user to specify fine-grained permissions for the processes run from the shell.
- **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.

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

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

3.2 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 in the 'Build System' section below.

3.3 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, 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.

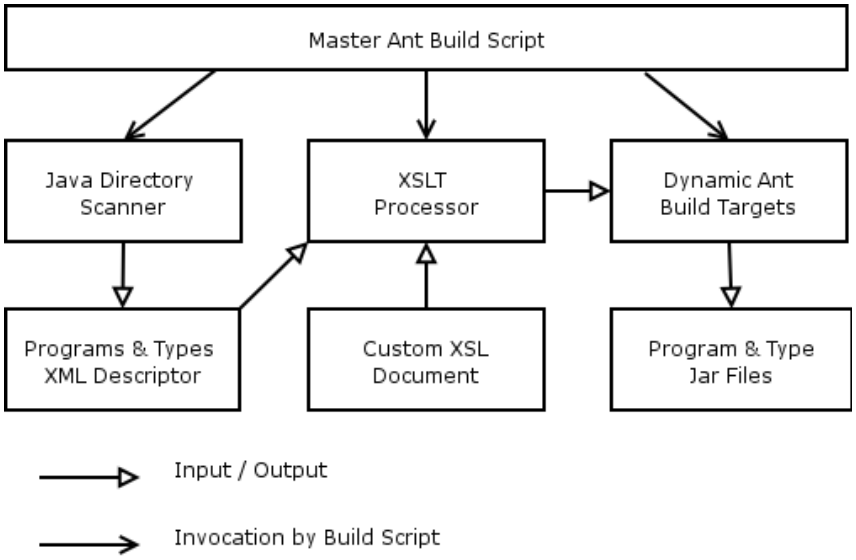
[3] <http://www.eclipse.org/swt>
[4] <http://www.eclipse.org>
[5] <http://www.cvshome.org>
[6] <http://ant.apache.org>

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 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 3.1 Program and Type satellite jar generation by the Build System



3.4 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.

3.5 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

[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".

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 web-navigatable version of the project source code.

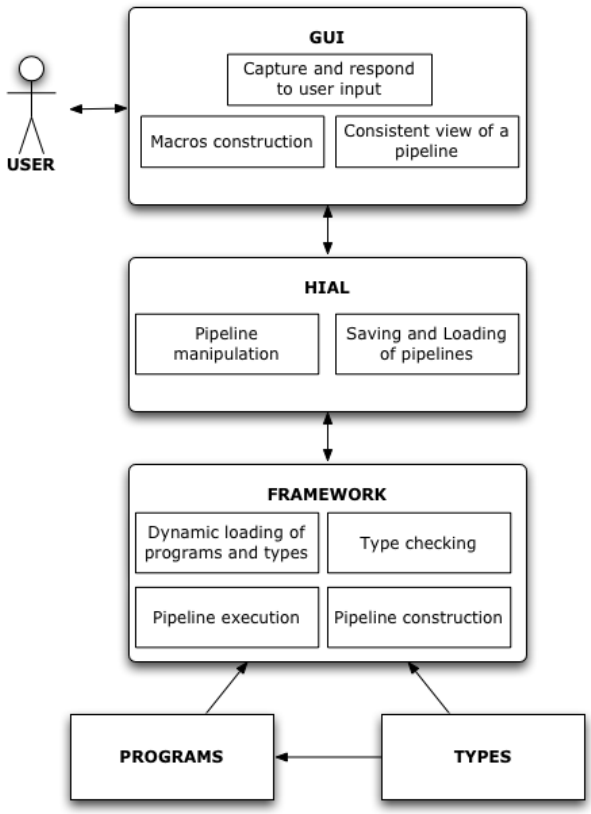
[8] <http://maven.apache.org>

Chapter 4

Component Overview

In the initial specification, Kevlar was described in terms of four main components; the Programs and Types, the Graphical User Interface (GUI), the Human Interface Abstraction Layer (HIAL) and the Framework. The roles of these layers have not changed since the first design, and communication interfaces between each component have now been specified, allowing each to be developed independently. In the following sections, the design of each layer is discussed in more detail, along with an outline of its implementation, how it communicates with the other components in the system and how it meets the sections of the specification completed so far.

Figure 4.1 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 5

Programs

5.1 Overview of the program infrastructure design

5.1.1 Purpose

This section explains the design of the components that are necessary to support programs in the shell.

5.1.2 Program structure

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

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 have 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.

5.1.3 Program meta-information

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.

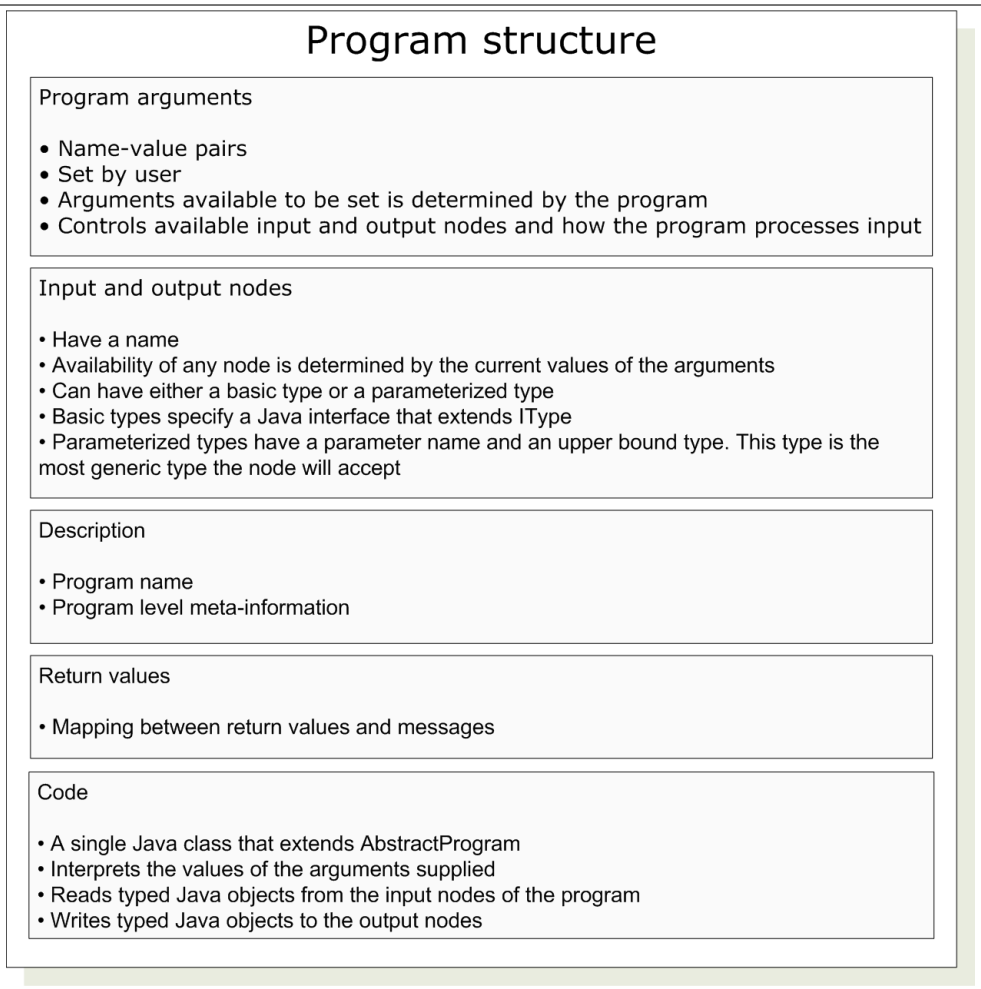
  To allow different versions of programs to be distinguished from one another, version information should be available for all programs. This would standardize the method for getting the version information from a program.

  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

Figure 5.1 Overview of the structure of a program



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 arbitrairly 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 'task' argument would always be available for the user to set. The task argument would be of set type, which means it might be represented in the GUI as a dropdown menu. If the user chooses 'resize' for task, the width and height arguments become available for the user to set.

- **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.

- **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.

- **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.

- **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.

5.2 Overview of Program Implementation

5.2.1 Initial Implementation

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.[1] [2]

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.

5.2.2 Secure implementation

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 implement `javax.swing.JApplet`[3] 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 mechanizms are used to communicate the high-level objects that travel across the pipes to the child processes and back.

However, there are a few programs that have to be run inside the JVM of the shell itself. These correspond to the 'builtin' programs often found in other shells, and perform actions that it would be very

[1] The Isolation API <`http://jcp.org/en/jsr/detail?id=121`> should address some of these problems.

[2] For a more complete discussion of this problem, and some suggested solutions involving byte-code analysis, please see <`www.ida.liu.se/~almhe/publications/CSES-2004.pdf`>.

[3] or `java.applet.Applet` for those who prefer awt.

difficult to safely allow child processes to do. These actions include changing the shell's current working directory, and setting environment variables that can affect all other running programs. To get around this problem, there are programs that are part of the Kevlar source tree that are not loaded as plug-in's dynamically, but are loaded separately using normal Java creation methods. When these processes are to be executed, they are run in their own thread. Since they are shell built-in code, written by the group, they can be trusted.

As an aside, the same executing-untrusted-code issue also applies to Types that are loaded into the system. In this regard it is the user's responsibility to only load new types that they trust. Since there should be a much smaller set of types compared to programs (and by definition the type tree should be standard to common programs so it can be reused properly) this should be less of a problem.

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

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

  <!-- 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>

      <!-- 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>
```

5.2.4 Tests for the Programs Component

- **Loading and validation of meta data.** For every program in the framework, load and validate the meta data XML file into data structures. Write the meta data from the data structure into a buffer as XML. Normalize the two XML strings and to resolve ordering differences, take multisets of lines in the two XML strings. Verify the two sets are equal.

Chapter 6

The Framework

6.1 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 is used to validate the pipe connections within a pipeline. It uses the loaded types to construct a Type Tree and exposes various functions to check that pipe connections are type safe. It is the heart of the semantical checker of the pipeline construction system.
- **Contexts.** The framework manages a collection of contexts. Contexts are used to hold the data structure that represents the pipeline. It is able to add new programs to the pipeline and to connect them via pipes. It uses the Type Checker to validate user manipulations of the pipeline for type safety. Furthermore, it initiates the real time execution of the pipeline.

6.2 Loaders Implementation

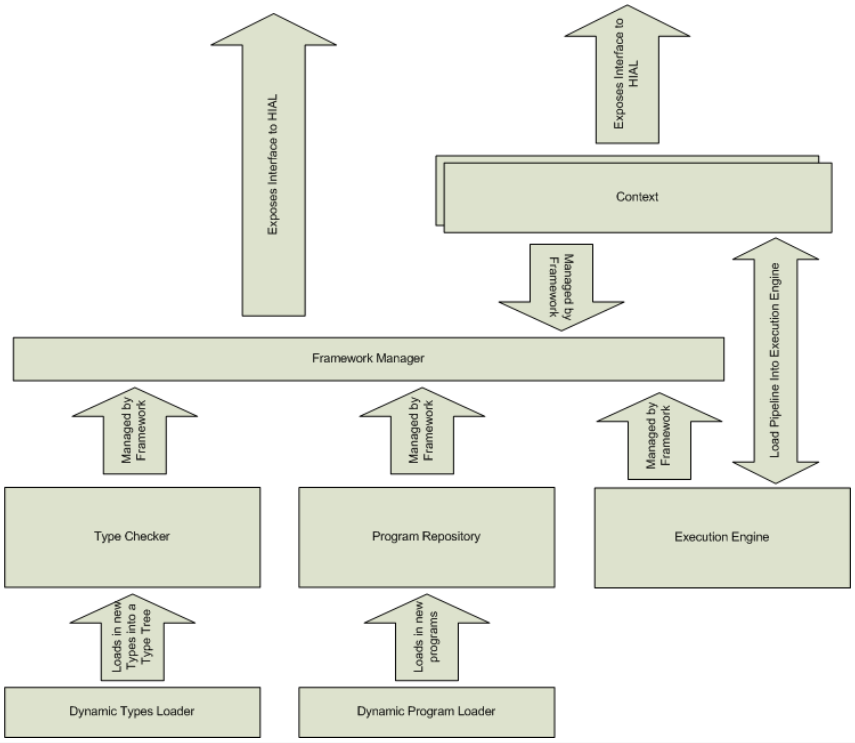
The implementation of the Loaders system is modular in design. It currently consists of four main parts, as outlined below:

COMPONENTS OF THE LOADERS IMPLEMENTATION

- **Directory Watching.** Underlying the loaders is a class that flexibly watches directories, and conditionally sends events about jar files that have been added, removed or updated in that directory to registered `INotifierFilter`s.

  The `INotifierFilter`s can decide if they have an interest in a jar file by analysing its 'Attributes' (jar manifest information) and the `DirectoryWatcher` keeps track of which notifiers have an interest on which jar files in which directories.

Figure 6.1 Block diagram showing the framework components interaction. The main components of the framework



This class has a `run` method that is invoked whenever a re-calculation of all directories should take place. At the moment this occurs every second.

- **Type Loading.** Using the Directory Watcher to locate jar files that export types in the shell system, this keeps track of all the type classes and provides a method for the framework to get at the representation of the type-tree.

  The type tree is checked at construction time to ensure it is valid (all types are interfaces, they all derive from `IType`, and there is no multiple inheritence).

- **Program Loading.** As with the Type Loader, the Program Loader uses the Directory Watcher to locate jar files that potentially have programs in them and loads in their XML descriptor files.

  These descriptor files are wrapped into an interface `IProgram` which is used by the framework and above to represent a program and the different things you can do to it.

  As part of construction of a program, a handle needs to be obtained for the types that it uses, so there is a link between the implmenetation of the Program Loader and the Type Loader (which uses package level visibility to resolve the dependency).

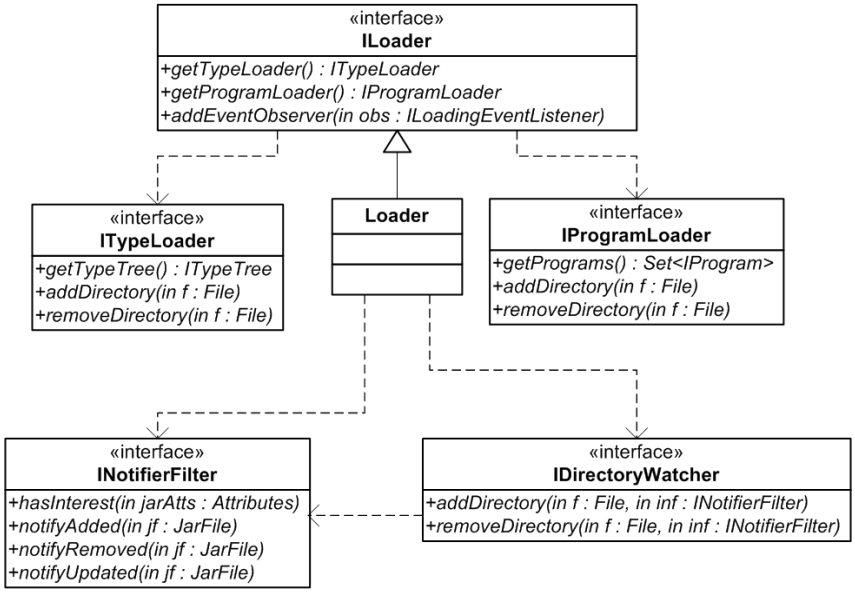
- **Loader Management.** To simplify the API to the rest of the framework, the Loaders section is accessible through a single interface, `ILoader`. `ILoader` allows simple registering to be told when either a Type or a Program event occurs (e.g. a Type has been added or a Program has been removed).

  The `ILoader` interface also provides access to the `ITypeLoader` and `IProgramLoader` so there is flexibility to allow other directories to be scanned for Programs / Types, and also access to the Type-Tree and currently registered Programs.

6.3 Pipeline Construction Implementation

Contexts are used to construct, validate and finally execute the pipelines. The following sections will explain the workings of the contexts in more detail.

Figure 6.2 Part of the Loaders Design



6.3.1 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 a `IContext`.

6.3.2 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 users 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 user simply 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.

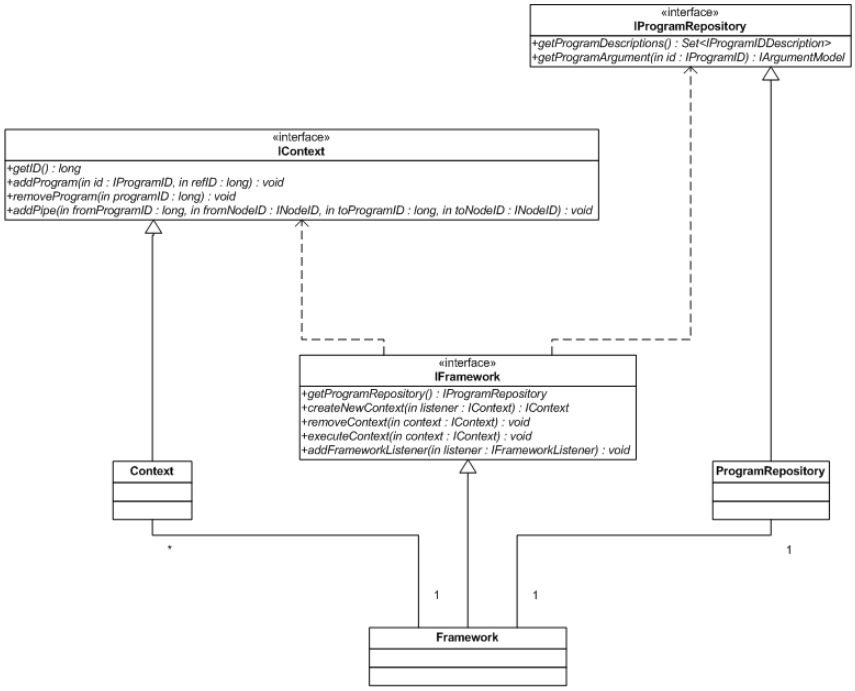
Once the `IProgamID` of the program has been returned, the list of laoded programs and their associated `IProgramID`s is then obtained by calling `getProgramDescriptions()`. The `IContext`'s `addProgram(...)` method can then be called to add the program to the pipeline contained within the `Context` in the form of an `IPipelineProgram`.

6.3.3 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 betwean them. This action will be type checked internally by the `TypeChecker` and will inform the user if this connection is type safe and how it affects the other connected pipes in the pipeline.

Figure 6.3 Framework public interface dependencies



6.4 Type Checking Implementation

Type Checking is the part of the Framework that is in charge of the validation for type safety of the pipeline. After every manipluation of the pipeline by the user the `TypeChecker` will be invoked to validate it and report eventual type mismatches. The `TypeChecker` is hidden to the Client side and will be invoked by the framework side.

6.4.1 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 `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.

6.4.2 Input and output type specification of Programs

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 to. 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`.

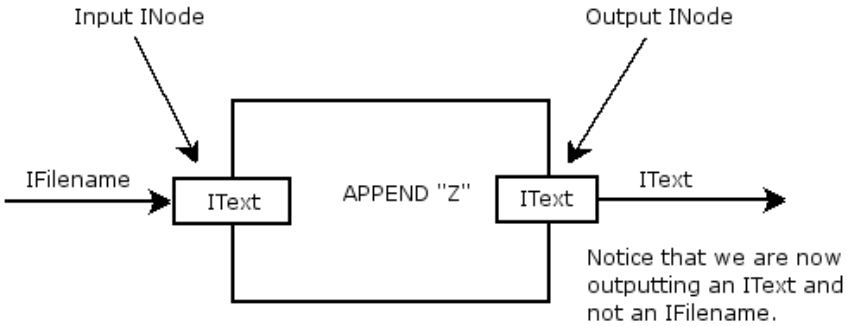
6.4.2.1 IBasicTypeClass

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 one 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 one should associate `IParameteredTypes` to the `INodes`.

Figure 6.4 IBasicClassType upcasting. Diagram of a Command which upcasts `IFilename` to `IText`.



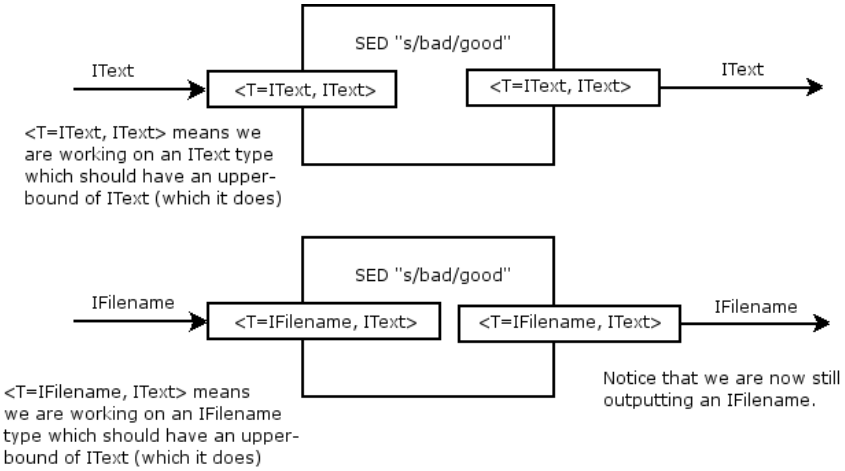
6.4.2.2 IParameteredTypeClass

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 6.5 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.

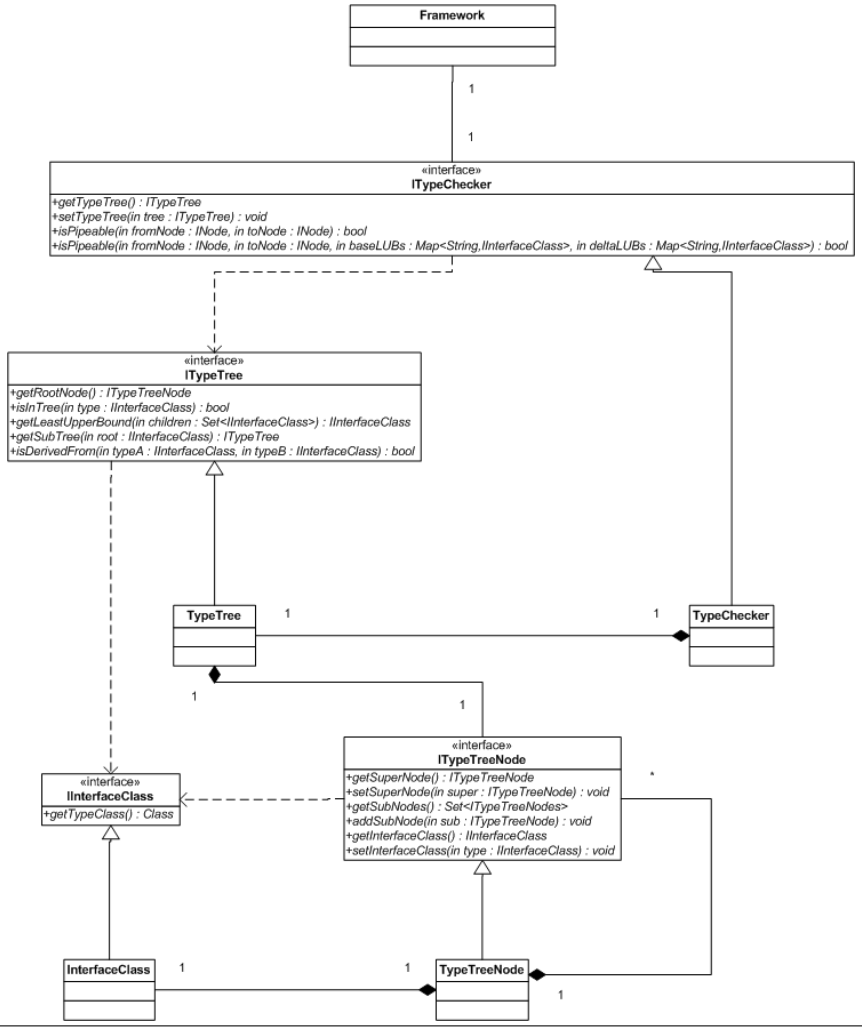


6.4.3 Type Flood Algorithm: The inferrence and type check algorithm

Type Flood is the algorithm that is used by the framework to infer the type of the `IParameterdTypeClass`es and to check if the pipe connections between programs are type safe. It is a recursive algorithm since users are allowed to make 'loop backs' in the pipeline, where a program's output is connected with the input of a program previously used in the pipeline.

The algorithm heavily uses the type tree to perform analysis on the types and to infer the types of the parameters such as finding the nearest common ancestor in the type tree (i.e. least upper bound of the types). This report does not contain a detailed specification of the algorithm, but the final report will contain a Haskell replication of the algorithm to proove some frequently asked questions such as proof of termination and general soundness.

Figure 6.6 Design of the TypeChecker component



6.5 Planned Execution Implementation

As described in Section 5.2.2 (most) programs are now being executed in separate JVM's. As part of this execution, the child processes need a way to communicate with the Kevlar shell in order to pass data across pipes to other programs via the Framework.

In addition to this, the Kevlar shell needs to maintain the environment that the child processes inherit, particularly the current working directory. A limitation of Java is that there is currently no way to change the working directory of a JVM internally, so this has to be kept as a state variable of the execution

system. This in turn leads to the requirement for some builtin programs, as only programs running inside the shell's VM are able to access (and therefore change) this state variable.

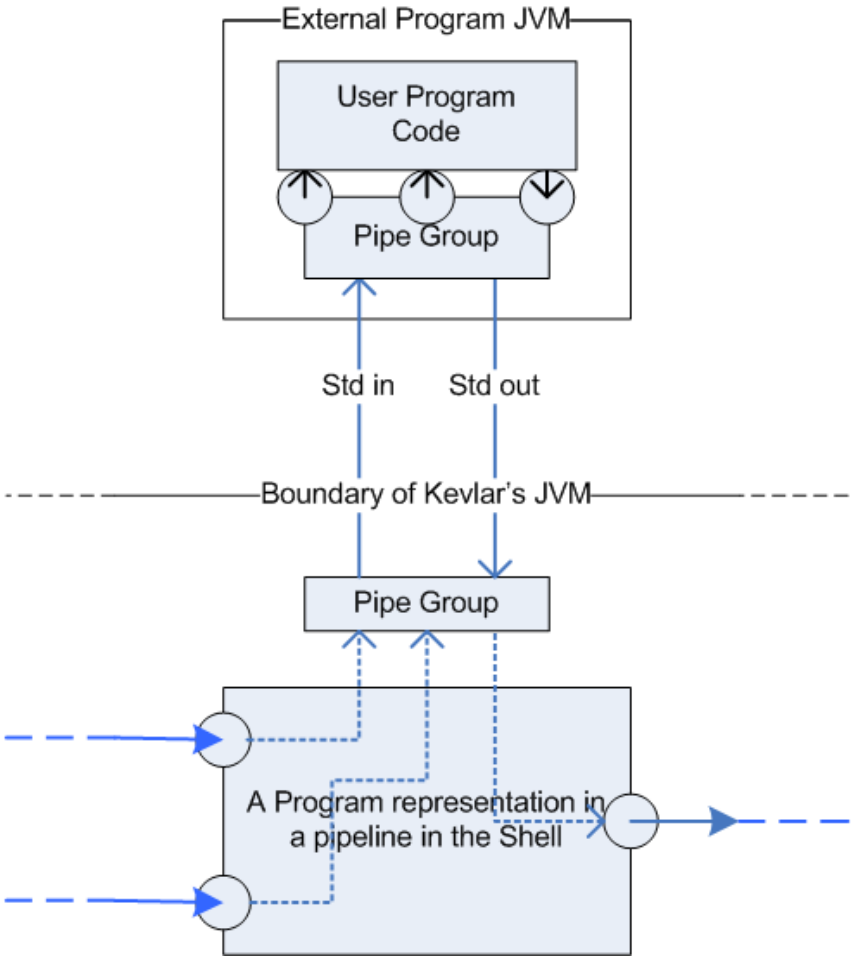
In order to perform execution, there is an `IExecutionEngine` interface that is currently implemented by a class under the singleton design pattern. This use of singleton enforces that only a single environment and current working directory can exist within the Kevlar system at any time.

When a program is requested to be run from the Framework (via a method in the `IProgram` interface), it (the `IProgram` instance), invokes the `IExecutionEngine`. Details of the Java class name of the program to be run, the jar file it is located in, and details of the arguments and pipes are passed to the Execution Engine. These details are used by the Execution Engine to execute the java command with the correct environment, working directory and classpath.

However the class that is 'executed' is not the program that has requested to be run, but instead `AbstractProgram`. The main method of `AbstractProgram` sets up communication with the Execution Engine's pipe handlers in order to allow the high-level pipe communication across standard in and out, and then reads its first argument to find the actual program code to load and run.

If any errors occur in protocol from the child process across standard out and in, it is assumed the program is malicious, and is killed.

Figure 6.7 IPC between Programs in a Pipeline and their JVMs



6.6 Framework Tests

- **Program and Type Loading.** Pre-defined test programs and types are loaded, events fire if a jar file is added, valid type-tree construction, stability if errored program descriptions are found.

  Testing `IDirectoryWatcher`. Initialisiation, adding a directory.

- **Construction of the type tree.** Use the dynamic type loader to construct a tree, and check various functionality, like `isInTree` and `isDerivedFrom`.
- **Construction of the `ProgramRepositry`.** Use the dynamic program loader to populate the `ProgramRepositry`, check that the test programs are loaded correctly.
- **Construction of the Pipeline.** Using the framework components build various pipelines that cover different scenarios. Test for adding programs, adding pipes, removing programs and removing pipes. Test loop back in pipes and different recursive behavior. Check for time of computation.
- **Execution of the Pipeline.** First draft of the `RuntimePipe` behaviour, check that it matches with the semantics of the `ExecutionEngine`.
- **Runtime Pipe Management.** Initialisation, propogation of information across streams, closing pipes, read and put.
- **Execution.** Test communication is correctly set up with a remote JVM, exit code information propogates back correctly.

Chapter 7

Human Interface Abstraction Layer

7.1 Overview of HIAL design

7.1.1 Design aims

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.

7.1.2 Retrieval of programs

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, the group decided that the system needs to provide a means to search for programs and look-up programs by category.

The design requires that the HIAL provides a function to find programs by a keyword-based search. Also, to allow a user to look up a program by category, it was decided that would be a program category tree.

The program category tree is a user-organized categorization of programs into folders and sub-folders. For example, there can be a folder for file utilities and image utilities.

7.1.3 Construction of pipelines

As the framework mediator, the HIAL has to wrap up a large library of functionality for constructing pipelines and extracting information about the pipeline for the GUI to use.

Since the HIAL is a mediator, the interface it provides to the GUI should be simpler than the actual interface of the framework.

7.1.4 Saving and loading of pipelines

It is clear that saving of pipelines is functionality that should be factored out of the GUI, as all GUIs that implement our system would be able to reuse the code. However for every program and pipe saved, there is some GUI specific information that needs to be saved with it such as canvas co-ordinates. To solve this problem, a callback interface is used. This callback interface allows the GUI to specify GUI specific data to save and later load.

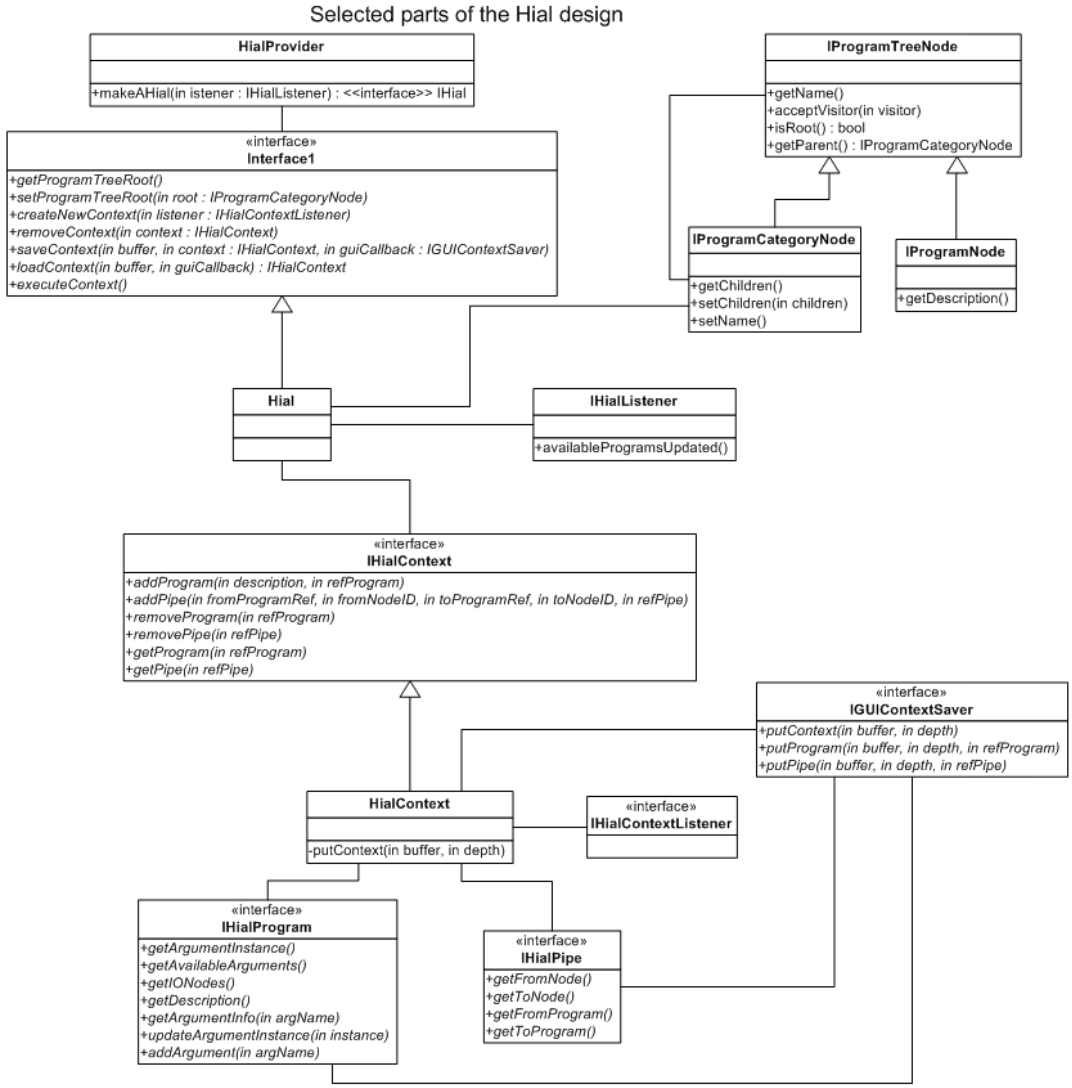
Given this design, the group decided that XML would make an excellent format for the saved information. This means that the GUI can specify the XML it wishes to save and it can be cleanly nested inside a set of GUI specific data tags within the saved file. Also, because of the way XML validation works, it's 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.

On the loading of a context, the HIAL can make no assumptions about the structure of the file, other than what has been validated by the XSD schema. Therefore, it will need to perform semantic validation on all the relevant fields.

7.1.5 Class design

Figure 7.1 Selected parts of the HIAL design



7.2 Overview of HIAL implementation

7.2.1 Retrieval of programs

The programs will be able to provide a set of keywords to allow for keyword searching. However, 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 won't solve the problem in the general case. For example, if the user types 'directories' but a program exports the keyword 'directory'.

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 GUI requests a program category tree from the HIAL on its initialisation and every time it receives a `AvailableProgramsUpdated( ... )` event from the HIAL. This allows the HIAL to manage the dynamic nature of the program loading. If a programmer recompiles a program, this will be reflected in the GUI as soon as the GUI handles the event.

The program-to-category mappings are stored in an XML file but will be editable from the GUI. To avoid having to write to the disk every time a program is moved from one category to another, the program category tree will only be loaded and saved on the open and close events of the system.

7.2.2 Construction of pipelines

The `IHialContext` interface contains functions to construct a pipeline that are typically delegated to a framework context interface. However, the abstraction layer has proved invaluable in creating a simpler interface for the GUI to use over the framework. In particular, there is an invalidation issue which is completely abstracted away by the HIAL. Because the framework needs to only have two primitives for constructing pipelines (add pipe and remove pipe), this means that there are times when pointers to pipes become invalidated because they have been removed and then added again. Therefore users of the framework have to revalidate the pointers between each call, and if they are invalid, they have to get the new pointers from the framework. This is all handled by the HIAL.

Here is an example for programs that solves a similar problem. The call to `IHialProgram` to get the current arguments is delegated to a framework version of the same interface `IPipelineProgram`. This actually results in a call to validate that the program is still valid. If it is, it makes the call; otherwise, it gets a fresh pointer from the framework and makes the call. (There is also an additional check; if the pointer is still not valid, it must mean the program was removed from the context so an exception is thrown.)

Similarly, for other parts of the pipeline construction, the HIAL interface supplies that which is most useful to the GUI while using what the framework is able to provide.

7.2.3 Saving and loading of pipelines

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 XML DOM `Element` object representing the GUI information is simply passed to the GUI.

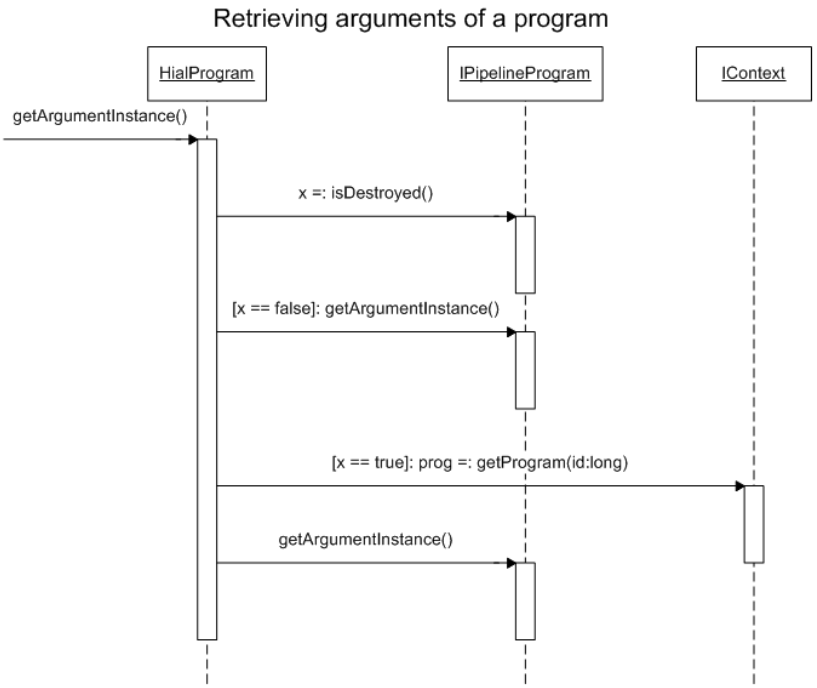
There is a DTD and auto-generated XSD schema for the XML save format, but for readability, an example save file snippet is presented below.

7.2.4 Example save file extract

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

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

Figure 7.2 Sequence diagram showing pipeline construction using the HIAL. The HIAL hides away the revalidation process that the framework requires.




```
  <!-- 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>
      <!-- GUI specific program data goes here: eg...
      <x>10</x>
      <y>10</y> -->
    </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>
```

7.2.5 Tests

7.2.5.1 Current tests that verify the correctness of the HIAL implementation

- **Program category tree loading.** Load the program category tree from the HIAL and verify there is a non empty tree of programs. Verify specific programs are in their expected location.
- **Pipeline construction and saving.** Build a simple three program pipeline that should type check correctly. Save the pipeline to a buffer and verify the XML generated is correct.

Chapter 8

The Graphical User Interface

8.1 Choice of library

The decision over which library to use for rendering graphical components in Kevlar was identified early on as a key point in determining the GUI's overall design as well as affecting the look, responsiveness and compatibility of the program across different operating systems. The two main Java graphics libraries were considered for this role:

- **Swing.** The main graphics library supported by Sun and compatible with all platforms on which the Java Runtime Environment is available for. This graphics library is written entirely in Java, and thus tends not adopt the look and feel of the operating system under which the program is running. As a result, Swing applications often look out-of-place and can have poor response times. However, Swing is strongly supported with a large range of widgets and additional extensions available, and has been proven successful in a number of large Java applications (including the NetBeans IDE).
- **SWT.** An unofficial graphics library for Java developed by IBM for use in their Eclipse IDE. SWT wraps native widgets to produce an interface identical to others on the same operating system. Due to the smaller amounts of Java code in the library, GUIs built using SWT also tend to respond faster than their Swing counterparts. However, due to the reliance on native widgets, programs written using SWT are not compatible with all JRE-supported operating systems and an operating-system dependent SWT library must be distributed with the application, since it is not currently part of the JRE distribution.

SWT was chosen as the library to be used on this project due to its advantage in response time over Swing. Since Kevlar is targetted to Windows and Linux machines, SWT's support of these operating systems seems sufficient, and the platform-specific library can easily be included as part of the Kevlar installation program. Using SWT also gives us the possibility of integrating our project into Eclipse as a plug-in at a later date.

8.2 Design Overview

8.2.1 Graphical design

The first stage of graphical interface design involved identifying the information that would need to be displayed to the user, and how the user would interact with it in order to achieve the functionality stated in the specification.

- **Current view of the pipeline.** The main part of the interface involves displaying to the user the current pipeline and allowing methods of editing it. This can be split into several major sub-components:

  - **Programs.** Each program in the pipeline should be represented, displaying either its name (as obtained from the HIAL) or a user-defined alias. The user should be able to move and resize programs within the constraints of the layout algorithm used.
  - **Input Nodes.** Associated with each program should be its input nodes (if any) which denote points of the program to which piped data can be passed from other programs. These points should display their names (as obtained from the HIAL) or a user-defined alias. Input nodes should also be selectable, under the constraint that at most one input node from the entire pipeline can be selected at any one time.
  - **Output Nodes.** Similar to input nodes, a program's output nodes should also be displayed, denoting points of the program from which data can be piped out. At most one output node from the entire pipeline should be selected at any one time.
  - **Arguments.** A program's argument set should be visible to the user, along with suitable widgets for altering the value of each argument. Argument values should appear highlighted if they do not meet the validation requirements specified by the program. Only the arguments which are currently available to the user to be set should be shown.
  - **Pipes.** Upon selecting an input node and an output node, a pipe should appear between the two. Pipes should be represented as lines between input and output nodes, with their path determined by the current layout algorithm. If this pipe is reported by the HIAL as being type-unsafe, it should appear highlighted.

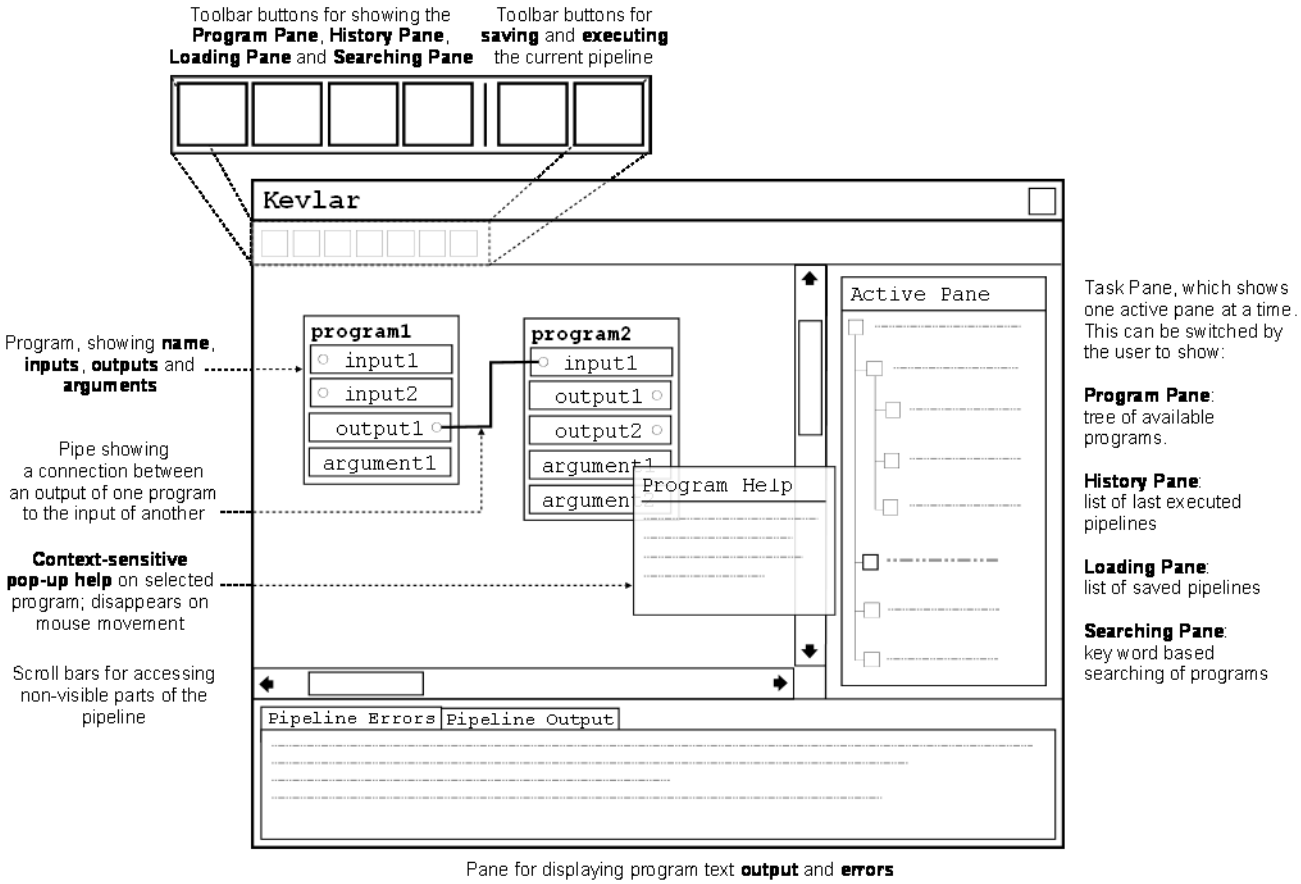
- **Help on selected programs, inputs, outputs and arguments.** When a program, input, output or argument is selected, help relevant to the current selection should appear if available. This help should be navigatable (i.e the user should be able to scroll it if necessary), however it should also be non-intrusive, disappearing once the user has clicked off of the selected item.
- **Tree of available programs.** The user should be able to view the tree of programs that are available to be used in the pipeline. Programs are divided into categories, which may be nested. The user should be able to easily navigate this structure and then drag-and-drop a program from the tree onto the pipeline.
- **List of previously saved pipelines.** The user should be able to view previously saved pipelines and be able to easily distinguish between them. Upon selecting a saved pipeline it should be loaded from disk, replacing the existing pipeline in the pipeline view.
- **Section for entering search terms and viewing search results.** The user should be able to enter keywords into a text box and search for available programs containing those keywords. The search results should be displayed in a list, from which the programs can be dragged and dropped into the current pipeline.
- **A list of the previously executed pipelines.** A history of executed pipelines should be kept, which can be viewed by the user and selected. When a pipeline is selected, it is loaded from disk and replaces the current pipeline in the pipeline view. This is intended to replace the history navigation provided by the up/down cursor keys in the Linux and Windows shells.
- **Errors and text output from executing programs.** Any textual output or exceptions generated by executing programs should be visible to the user.

Having identified the information to be displayed by the graphical interface, a visual design was drafted to contain these data.

This highlighted a potential problem with the interface; the pipeline itself had to contain so much information that it could easily become too large to fit inside the window. Unless enough of the pipeline is visible at any one time, large pipelines could become very difficult to navigate.

To solve this problem, the design was reworked to give the user more control over which information would be visible at any one time. Firstly, the Task Pane would be initially hidden, and would become visible by clicking one of the toolbar buttons for showing a pane. Once shown, it could then be hidden again by the user, freeing up a large amount of space on the right-edge of the window. The same idea was then applied to the lower Output Pane, so that most of the the time the main pipeline view could fill the entire window under the toolbar.

Figure 8.1 Initial draft of graphical interface design. This design shows the initially designed layout of the GUI, including where all the features mentioned in the specification should be placed.



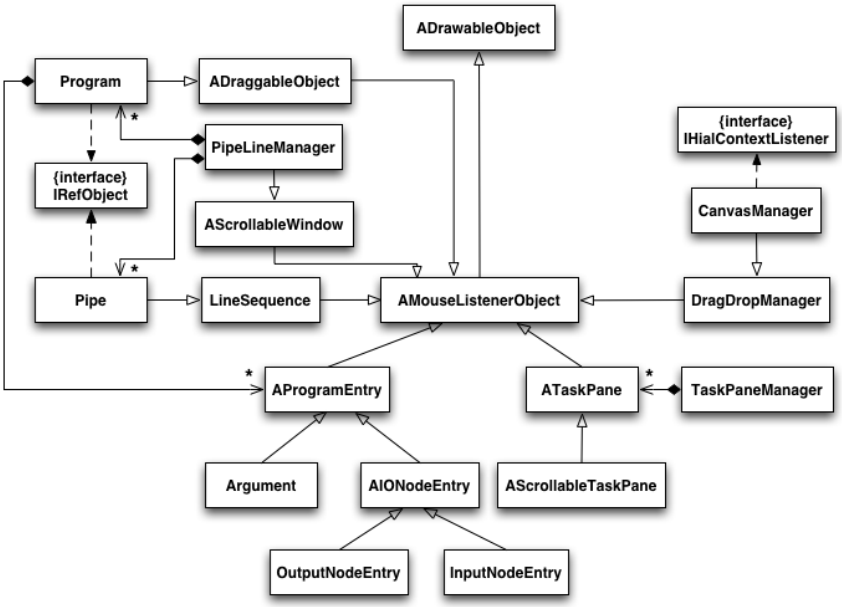
For the pipelines themselves, the design was adjusted to include an option to 'contract' programs into a smaller form, where inputs and outputs would be collected into a single pair of nodes, and arguments would be made completely invisible. By doing this to all programs, the user would be able to see an overview of the pipeline that would take up much less room than showing all the information at once. To see the full detail of a program, the user could then simply click a button to expand the program into its full representation.

8.2.2 Design of Java code using UML

Once an outline of the graphical design had been completed, an initial design of the Java code was created using a UML diagram. Since the graphical design contained many widgets that could not easily be represented using standard SWT widgets, the decision was made to create a new graphics library on top of SWT that would provide a set of new, basic widgets that could then be built upon to represent more complex objects such as the programs and task panes.

- **ADrawableObject.** Primitive objects that simply want to be displayed on the canvas subclass `ADrawableObject`. Other instances of this class can be added as children, where they will be drawn relative to the parent's coordinate system. The root drawable (the `CanvasManager`) listens to redraw events in the canvas and propagates the event through its children. The event is only propagated to drawables within the redraw area specified by SWT, so only areas of the screen that change are actually redrawn, which helps to keep response times down. `ADrawableObject` also provides methods for alerting the drawing system that it needs to be redrawn by passing its absolute area to the `CanvasManager`. Rather than instantly redrawing, the `CanvasManager` instead aggregates the areas from all redraws caused from a single event (such as a mouse click), and then applies them in one go. This prevents 'overdraw', where a section of the canvas is drawn

Figure 8.2 Initial design of GUI code in UML. This UML diagram shows the relationships between the various classes in the first design of the graphical interface.



only to be instantly drawn again by another object.

- **AMouseListenerObject.** This class is subclassed by objects that need to respond to mouse events. The root mouse listener (the `CanvasManager`) listens for mouse events from the canvas, then propagates them through its children. Each child then translates the mouse coordinates to its own local coordinate system before calling the abstract method to perform its response.
- **GUIManager.** This class contains Kevlar's `void main(String[] args)` method, which initialises the GUI's `CanvasManager` and the HIAL. This class is also responsible for requesting a context from the HIAL and providing it to the other GUI objects when they need to obtain information from the Framework.
- **CanvasManager.** This class wraps the SWT canvas used for displaying Kevlar's custom widgets. It captures the canvas' paint and mouse events and propagates them to its drawable children.
- **TaskPaneManager.** Provides methods for switching between available task panes. This passes paint and mouse events to only the active task pane. Also provides methods for expanding and contracting the current pane.
- **ATaskPane.** This is the superclass of all task panes. It provides methods for drawing its name and vertically resizing when the parent window resizes, which are common to all task panes.
- **PipelineManager.** A subclass of `ADrawableObject` which contains all the programs and pipes that comprise the pipeline representation. The drawable resizes itself to be larger than its contents, and shows scrollbars when this size exceeds the size of its parent.
- **Program.** A drawable for displaying the name of a program and containing a list of `AProgramEntry` for representing the program's inputs, outputs and arguments. This drawable resizes itself vertically so that it is always just larger than its children.
- **AProgramEntry.** The superclass of the classes used to display inputs, outputs and arguments.
- **AIONodeEntry.** Contains code for displaying the selectable node icon that is included in both input and output node entries.
- **InputNodeEntry.** Displays the name of the input along with a selectable node icon on the left that can be clicked to connect up a pipe.
- **OutputNodeEntry.** Displays the name of the output along with a selectable node icon on the right that can be clicked to connect up a pipe.

- **Argument.** Displays the name of the argument and its current value.
- **Pipe.** Takes an `InputNodeEntry` and an `OutputNodeEntry` in the constructor and creates a visual representation of a pipe between them.

8.3 Implementation Overview

8.3.1 Custom widgets

In order to represent the programs, pipes and task panes, which cannot easily be mapped to existing SWT widgets, a number of custom widgets were implemented.

- **Line.** A simple drawable which takes two points relative to its parent's coordinate system and draws a line between them. A bounding box is used to avoid redraw outside the range of the line. An instance of this class is used for drawing a single segment of a pipe.
- **LineSequence.** A collection of lines forming a path between two points. This is subclassed by `Pipe` to draw the pipe between an input and output node.
- **RollOverButton.** A button that changes its image when the mouse moves over it. This was required since the design has many custom widgets that accept mouse events but do not look like traditional buttons (an example of this is the selectable node for each input and output point). By being highlighted when the mouse moves over it, the user will know that the object responds to mouse clicks.
- **SelectableButton.** A subclass of `RollOverButton` that changes between two states when clicked. This is intended for objects that can be selected an deselected. The `Node` class, representing the selectable node for each input and output point, is a subclass of this.
- **StretchyBox.** The existing SWT `Composite` and `Canvas` controls did not offer enough flexibility for efficiently implementing the custom widgets needed for representing the pipeline, so instead a lightweight `StretchyBox` control was written. Like `Composite`, this groups controls, but it also resizes itself to be large enough to contain all its children. By extending `ADrawableObject`, `StretchyBox` automatically adopts the background of its parent, allowing for transparency which would be difficult to achieve with existing SWT controls.
- **ScrollingWindow.** Provides a window which can accept children positioned at any point greater than (0, 0) and provides scrollbars for viewing areas outside its bounded size. The implementation of this involves a `StretchyBox` which reports back to the `ScrollingWindow` when its size changes, which in turn updates the scrollbars. When the scrollbars are clicked by the user, the `StretchyBox` is moved accordingly. Since the `StretchyBox` is a child of the `ScrollingWindow`, the areas that lie outside of the window's dimensions are not drawn.

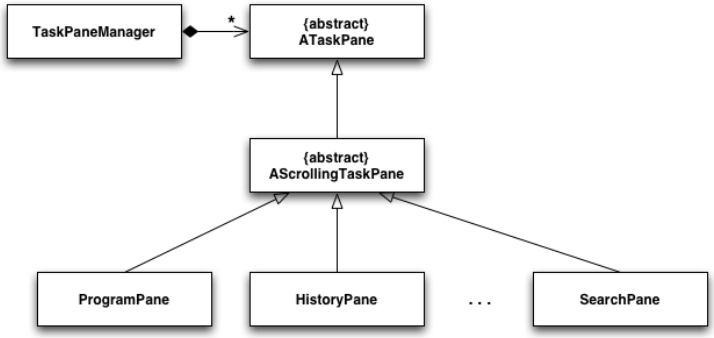
8.3.2 Pipeline implementation

The structure of the pipeline classes outlined in the Design Overview were implemented, allowing the user to view the current state of the pipeline. Since a program can have many input nodes, output nodes and arguments, it was decided that the unused entries should be hidden from the user to conserve window space. A button was added to the top of the program representation which, when clicked, would allow the user to view all the hidden entries so that they can be edited.

8.3.3 Task panes implementation

In the Design Overview it was stated that in order to achieve the functionality described in the specification the user should be able to view the tree of available programs, search for programs, view the list of previously executed pipelines, and load pipelines that had been saved to disk. These have been implemented by extending the `ATaskPane` and `AScrollableTaskPane` classes where appropriate.

Figure 8.3 Task Pane design. A UML diagram showing the subset of the overall design which outlines the relationships between task pane classes.



Each task pane deals with just one functionality aspect. Only one task pane is displayed at any one time and it is up to the user to decide which pane he/she wants to use. Switching between different panes is controlled by the `TaskPaneManager`.

Such architecture allows each task pane to be implemented independently 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.

8.3.4 Implementation of the context sensitive help

Help is available for all relevant components of the pipeline by clicking on the component of interest. Since only one help window is visible at any one time, it made sense to make the `HelpManager` class follow a singleton design pattern, so that any object may call `HelpManager.displayHelp(String name, IHelp help, Point position)` to configure the help pop-up window to display the desired context-sensitive information.

`HelpManager` class makes use of the SWT Browser widget to display help. The Browser widget renders HTML, which allows HTML formatting to be used, including style sheets, on the context of help and therefore make it more user-friendly. The Browser widget also inherits from SWT `Scrollable` class, which makes it navigatable and allows users to scroll if needed. `HelpManager` takes help information passed to it by pipeline components, wraps it in HTML code and uses Browser to display it.

8.3.5 Layout implementation

One of the most difficult features to implement in the graphical interface was a layout algorithm for deciding the rules of where programs could be positioned and the paths that pipes would take to connect input and output nodes. Ideally, the layout should use the user-defined positions of programs where possible, only moving them to avoid program overlap or when the user hasn't defined a position (i.e by using keyboard input to add a program). Pipes should contain only right-angled bends, not overlap programs and avoid crossing other pipes where possible.

Since implementing such a layout algorithm would likely take a large amount of time, and is not pivotal to achieving the goals set out in the specification, a set of interfaces were defined for allowing a variety of layout algorithms to be implemented at a later date. A singleton `LayoutManager` class can be called to get access to the active `ILayout` instance, which is notified when programs are added, moved, resized or removed from the pipeline. It is also told when a pipe is added, removed or has its connection points moved, so that the path of the pipe can be calculated. It then applies its layout by using a callback mechanism to set the positions and paths of the programs and pipes in the pipeline.

Chapter 9

Evaluation Strategy

One of the goals set out in our initial specification was to adpot a design process that encourages good usability. To achieve this, an iterative process has been adopted that produces an internal release of our program, with all sections integrated and functioning at some level, at the end of each week (see Section 2.2). The testing of the individual components of the project is performed by using JUnit, where applicable, in order to highlight minor bugs before integration. Meanwhile, the weekly internal releases allow the integration of all the components to be tested, and also produce an overview of the product's progress as a whole.

At each milestone in the project, the group is able to look at the integrated product and discuss any usability issues that have arisen. This informal evaluation of the project allows for major problems to be highlighted and fixed during the next iteration. However, group project members do not themselves give accurate results for usability evaluation due to their prior knowledge of the internal workings of the product. To address this issue, we plan to produce an external beta release of our product at Internal Milestone Five once all minimum specifications have been completed. This release will be distributed to a number of people across a range of abilities, who will then be questioned about their experience of the system.

To obtain information about specific areas of our system, beta users will be asked to perform items specified on a task list when first using the program. They will then be able to access an online questionnairre which will ask them about the following areas.

- **Previous experience of the existing consoles available.** Users will be questioned about previous experience of the existing Linux or DOS shells in order to determine their level of expertise. This will help us to determine whether existing users find it easy to migrate to our system, and how user-friendly non-shell users find it.
- **Ease of locating programs.** The task list will ask users to use the system to perform a specific job, which will require them to locate the programs needed to fulfil that job. They will then be asked how quickly they were able to locate these programs, giving both comments and a mark on a scale. By correlating this data with the user's experience, we will be able to determine whether Kevlar bridges the gulf of knowledge between experienced users, who will be able to find programs based on their Linux or DOS equivalents, and newcomers, who will use keywords based on their tasks.
- **Ease of setting program arguments.** Certain tasks will require the user to configure arguments. Users will be asked whether they made use of the context-sensitive argument help and, if so, how they found the layout and content of the information.
- **Ease of connecting pipelines.** Users will be asked to connect pipelines between programs, which will make use of the type-checking of pipes. Users will be asked whether or not they found the type checking helpful, and whether they could understand and respond to any errors they received.