The Development of a Java Image Processing Framework

A thesis presented in partial fulfillment of the requirements for the degree of Master of Technology in Computer Systems Engineering at Massey University, Palmerston North, New Zealand.

Jesse Louis McLaughlin

2000

Abstract

Practical computer-based teaching methods are often used in conjunction with theory-based lecture sessions and textbooks when teaching image processing. In kind, electronic or on-line image processing courses commonly provide both theoretical and interactive components, however these are often disparate in that the software use to provide each component is independent rather than integrated. It is less common to find electronic instructional resources for image processing that integrate theoretical textual and practical interactive content together into one seamless package. An integrated approach has the advantage that the concepts are more easily conveyed and reinforced when taught 'side-by-side' this way.

The World Wide Web offers an attractive medium for delivering an integrated instructional resource on image processing. Applets written in Java may be seamlessly integrated into a hypertext environment. These applets can provide practical demonstrations of image processing concepts along side the relevant hypertext-based theoretical content. One of the major barriers to realising this kind of resource is the development effort required to create the necessary applets. This research demonstrates that the provision of a software framework can significantly reduce the burden of developing these applets. Such a framework provides a common code base that can be drawn upon during applet development, thereby avoiding the need to start from scratch each time a new applet is needed.

The framework's design is modelled on a dataflow view of image processing, allowing applets to be built in terms of interconnections between operations. This design is intended to provide the developer with an intuitive and easy-to-use application programming interface (API) for developing applets. The framework also provides APIs for the programmer to implement new operations and data types, thereby extending the capabilities of the framework. Further, the framework's design is general enough to allow it to be used for developing general purpose image processing programs, or other programs that lend themselves to development using a dataflow language. This thesis shows that the proposed framework achieves its aims through an example application of the development of an applet that demonstrates a thresholding operation.

Table of Contents

Acknowledgments

First and foremost, thanks must go to my two very capable supervisors, Donald Bailey and Wyatt Page, each of whom made very different but equally important contributions to the research presented in this thesis. Thanks must go to Donald for his constant ability to provide expert advice and sound common sense in the face of my more fantastic and quick-conceived ideas. He was, after all, the originator of this research and therefore best qualified to provide the direction for the project, both overall and in its every detail. I would also like to thank Donald for inspiring me in the art of devising useful analogies, which itself proved useful when it came time to write this thesis. Wyatt, for his part (and to our fortune) had been less involved at the time the project was initially formulated, and so was able to bring a more objective perspective to the nature of the problem we had set out to solve. His speciality expertise in a variety of niche areas also proved invaluable on more than one occasion.

I am grateful to acknowledge Massey University's assistance, in the form of a Massey Masterate Scholarship. In kind, Donald's financial contribution to the project is also gratefully acknowledged in full, the sum of which afforded me an additional level of support that I might not have otherwise enjoyed.

I would like to give special thanks to my family. To my parents, for being forever interested in everything I do, and to my extended family, for being such a kind, generous and supportive bunch.

My friend and co-conspirator, David Orchiston, is deserving of thanks; his threats mixed with encouragement were of increasing value as the deadline for this thesis approached.

Finally, I would like to warmly thank Rachelle, not only for the constant companionship and much needed distraction she provided during this project, but also for being brave enough to love an aspiring computer systems engineer.

1. Introduction

1.1 Image Processing

Image processing can be likened to word processing or food processing, only on images. Although the analogy may seem a little tenuous on some points, it can nevertheless prove insightful. The ingredients to make a cake, for example, are combined (processed) according to a recipe and the result (we hope) is something that tastes far better than the original cake constituents. Nothing is gained that is additional to what was originally used to make the cake; the important point is that its form has changed into something much more pleasing to the palate. In a similar fashion, images are processed in order to transform the information contained within them into a form that is much more useful to whomever (or whatever) is going to make use of it. For example, an image may have its contrast enhanced, making it more pleasing to the human eye, or a thresholding operation may be applied to an image so that the number of foreground objects present can be easily counted. Russ (1994) cites the purposes of image processing as being two-fold. These are:

a) to improve the visual appearance of images to a human viewer; and

b) to prepare images for measurement of the features and structures present.

Whereas the tools used in cooking are mixing bowls and measuring spoons, and the raw materials are flour, eggs and cocoa, image processing has quite different requirements. The tools of image processing are not physical implements, but mathematically defined operations. Examples of common image processing operations are adding two images, thresholding an image, image filtering operations such as smoothing or sharpening and edge detection operations; and there are many more (see for example Castleman (1979), Gonzalez & Wintz (1987), Jain (1989) and Russ (1994)).

The 'materials' these tools operate on are images. An image can be described as a graphical, two-dimensional representation of the light intensity distribution of a scene as detected by an appropriate sensor (e.g. a camera)[1]. In the digital sense, images exist as arrays of picture elements, or pixels, each of which stores a discrete measure of the light intensity at a corresponding position in the original scene. Digital images are characterised by resolution, the number of pixels in each dimension, and depth, the number of bits used to represent each pixel.

The amount of raw data in a typical image tends to be high, for example an image 400 pixels wide and 300 pixels high with 24 bits per pixel would occupy just over 350 kilobytes, or the rough equivalent a 50,000 word text document (for comparison, the body of this thesis contains approximately 30,000 words). Processing these volumes of data has commensurate computational requirements, often involving many repetitious calculations. Simply adding two such images together, for example, would require 360,000 addition operations alone. Unsurprisingly, computers have proved an indispensable tool for image processing research and its applications. Image processing operations are conveniently implemented in computer software and many application environments have been built to facilitate human-computer interaction in image processing (Bailey & Hodgson, 1988; Rasure & Williams, 1991; Rasband, 1992). Further, computers have the storage and retrieval mechanisms necessary for managing large data structures such as images, and they can drive visual display units (VDUs) capable of displaying high-resolution, true-colour images, making truly interactive image processing possible.

[1] In general, an image may represent the spatial distribution of any signal that can be appropriately measured, and may also extend into three dimensions.

1.2 Web-Based Instruction

Few would argue that to become a chef, all one needs is a cookbook. Rather what is needed is a kitchen and a lot of time to experiment and gain hands-on experience. It may be argued then that a computer is to an image processing student what a well stocked kitchen is to an aspiring cook. In support of this, Hanna (1994) has observed that image processing is ideally suited to Computer Aided Instruction (CAI). Computers allow interaction with images and image processing algorithms *"using feedback based on the visual perception of the output image."* Similarly, Sonka, *et al.* (1998) emphasise the importance of computers in image processing instruction, noting that *"to maximise learning, students must become fully active participants in the learning process."*

In contrast, 'traditional' methods of teaching image processing have betrayed various weaknesses and deficiencies. Photocopied lecture notes, for example, reproduce processed images at very low quality, and additional images shown using slide projectors often prove too transient in nature to make an impact on the learning process. These problems are made worse when the concepts being demonstrated involve only subtle changes between the original and processed images (Hanna, 1994). Even more hindering is the lack of *interactive* facilities for demonstrating image processing techniques and algorithms to students. Textbooks, for instance, generally provide good theoretical coverage of the subject, however students' understanding could be greatly improved by providing 'hands on' experience with the concepts being taught.

The need for user-interaction and experimentation in the teaching of image processing have long made computers an important tool for image processing instruction (Jensen, 1983). Now, with the advent of the Internet and its golden child, the World Wide Web, researchers and educators have a new and appealing medium within which to provide instructional material. Web-Based Instruction (WBI), as it is called (Khan, 1997), offers advantages to users and developers alike. Among these is the steady proliferation of Web-browser software, which has resulted in widespread familiarity with a standardised user-interface. The ease with which material can be made accessible world-wide is another advantage, and one that cuts through physical and cultural divides as much as it does through platform-dependency issues. The ability to combine a variety of media types

(hypertext, graphics, sound, movies, etc.) into a single document is particularly advantageous, especially for creating stimulating and content-rich learning environments (the high resolution displays required to view these documents are now common-place, which has been a barrier to teaching image processing effectively in the past).

Unfortunately, despite the potential of the Web as a powerful instructional delivery medium, what is done with a significant volume of Web-based instructional material is that conventional materials are simply 'Webified', and many of the features and capabilities of the Web are not utilised. The results of such 'Webification' are often likely to end up in the shovelware category[2]. Web-Based Instruction, according to Khan (1997), *"is a hypermedia-based instructional program which utilizes the attributes and resources of the World Wide Web to create a meaningful learning environment where learning is fostered and supported."* It follows that in order to successfully 'foster and support' image processing instruction in a Web-based environment, interactive features must be provided that can be successfully integrated into that environment. The development of software to support those features is therefore an important step in developing interactive Web-based instructional systems on image processing.

1.3 Electronic Textbooks

Whether the subject is image processing or something quite different, the general term 'Electronic Textbook' (ET) (Barker, 1992; Oliver, *et al.*, 1994; Harger, 1996; Brusilovsky, *et al.*, 1997) is often used to describe this kind of 'enhanced' learning environment. Barker (1992) explains the usefulness of such a metaphor:

> *"Because people are so familiar with conventional books as a means of documenting and distributing information it is convenient for us to carry this 'model' over to the computer domain. We therefore use the term* electronic book *as a generalized metaphor or myth which will project an image (to both designers and users) of the 'instructional computer' as being just like a book. However, this type of book has many special properties and characteristics that make it particularly useful for learning and training tasks."*

[2] Shovelware is a somewhat less than euphemistic term often used to describe the results of simply 'shovelling' a whole lot of printed notes/course material online (possibly in a hypertext format).

To put this in an image processing context, even though conventional image processing textbooks provide raw theory and examples, they suffer due to the inherent linearity and static nature of their content. On the other hand, interactive image processing programs allow experimentation and give feedback, but offer very little in the way of guidance or instruction. An important contribution of computer-based image processing instruction is to provide a medium where theoretical textual and practical interactive content can be combined. The ideal image processing electronic resource would be an integrated reference textbook, tutorial system and image processing package, providing powerful navigation aids and flexible accessibility via a single common user-interface. In this context, the 'special properties and characteristics' that Barker refers to naturally include interactive features that enable users to investigate and experiment with the concepts as they are taught.

Invariably, a major decision in the development of any such learning resource is the choice of what image processing software to use to provide these interactive capabilities. Jordán & Lotufo (1996), for example, describe a Web-based image processing course that relies on the Khoros/Cantata (Rasure & Williams, 1991) image processing package to mix theory with practical exercises. The course is self-guiding, with chapters organised around classes of primitive image processing operations. Lessons within each chapter present fundamentals and illustrative examples relevant to the lesson before guiding the user through a laboratory experiment using Khoros. The laboratory pages contain hyperlinks that activate the Khoros operators named in the text, and at the end of each lab there is a link that opens a Cantata workspace; here the user can see how the operators that were discussed in the lesson work together.

This approach incorporates much of what could be expected in an ideal system, but there are drawbacks. For example, the image processing software employed is Unix-based. On a non-Unix platform, the course loses its 'hands-on' touch that is so important to image processing understanding. Also, the user effectively has to deal with a dual user-interface: a Web-browser delivers the 'theory', and Khoros/Cantata delivers the 'practice', which is a solution that could potentially prove cumbersome because the two are not fully integrated. An opportunity

exists, therefore, to make better use of Kahn's 'attributes and resources' for teaching image processing on the Web.

1.4 The Java Programming Language

Java (Gosling, *et al.*, 1996) owes much of its popularity to its general purpose, internet-centric nature. It is also object-oriented and platform independent, and though it shares much of its lexical structure with C, it has a reputation for promoting a 'clean' programming style and for being less prone to programmer error (Zukowski, 1998). Because Java is object-oriented, software development using Java is best carried out using object-oriented methods (see for example Booch (1994)).

Programs written in Java comprise one or more *classes*, which are in turn grouped hierarchically into *packages*. These programs exist in a platform-neutral *bytecode* format, which is executed by a *Java Virtual Machine* (JVM) running on the desired platform. A Web-browser can be made 'Java-enabled' by incorporating a JVM into its code, allowing it to execute Java programs known as *applets*. These are (typically small) programs written in Java that run within the context of the browser and are displayed as integral parts of a Web-page. A major advantage of Java programs executed in this way is that the browser automatically downloads the necessary bytecodes (which may mean individual classes or entire packages) when the page containing the applet is accessed. Hence there is no installation process associated with running these programs. Further, because the browser/applet architecture has been designed from the outset to ensure that maliciously written applets cannot harm the system they are run on, there should not be any security issues associated with utilising applets for doing image processing.

The most popular Web-browsers are now Java-enabled, and thus able to support Java applets. All applets gain a graphical display context and are able to respond to keystrokes, mouse-clicks, etc., via the encapsulating browser window. The Java language itself provides an extensive range of core classes for use with graphics and user-interface programming. The result is that Java applets are well suited to rendering high-resolution images and providing graphical user-interfaces.

Mukundan (1999) has demonstrated the use of Java for creating 'mini-tutorial' applets that demonstrate several different binary vision algorithms. These applets allow students to conveniently visualise the pixel-level workings of this class of image processing operation. Further, the design can be generalised to work with most other common pixel-level operations. This work shows how Java applets can provide interactive, experimental elements for teaching image processing, and concludes that such an approach would be suitable in a Web-based learning environment.

One of the greatest barriers facing developers of such applets derives from the general-purpose nature of Java. Programming image processing operations, linking them to a user-interface, acquiring input images and displaying the output; all of this may be done in an ad hoc manner. Unfortunately, this kind of approach is highly susceptible to such software engineering pitfalls as poorly conceived software design, repetition of coding effort, unmaintainable code, code that is difficult to extend, and bugs.

1.5 A Framework

An improved approach would be to utilise a software framework to assist the developer in creating the envisaged applets. Essentially, a framework is a software system targeted at aiding the development of programs that address a specific problem domain. A framework can shift much of the development burden associated with writing software away from the developer, allowing him or her to work at a higher level of abstraction and avoid unwanted detail. It might be said that a framework is the culinary equivalent of the difference between the kitchen at home and Burger King's kitchen if you are trying to make a hamburger. The latter is far more efficient for the chosen task, because it specialises. Gamma, *et al.* (1995) illustrate this in an object-oriented context:

> *"A framework is a set of cooperating classes that make up a reusable design for a specific class of software. For example, a framework can be geared toward building graphical editors for different domains in artistic drawing, music composition, and mechanical CAD. Another framework can help you build compilers for different programming languages and target machines. Yet another might help you build financial modelling applications. You*

customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework."

Figure 1.1 illustrates how a framework can be considered as a layer of abstraction. Just as the role of an operating system is to provide an abstraction for its underlying hardware, a framework provides an abstraction atop the developer's programming environment, in this case Java. As shown in the diagram, applets created using such a framework are able to draw on both the framework and the Java programming environment. The framework's contribution is to significantly reduce the development effort involved in creating the desired applets.

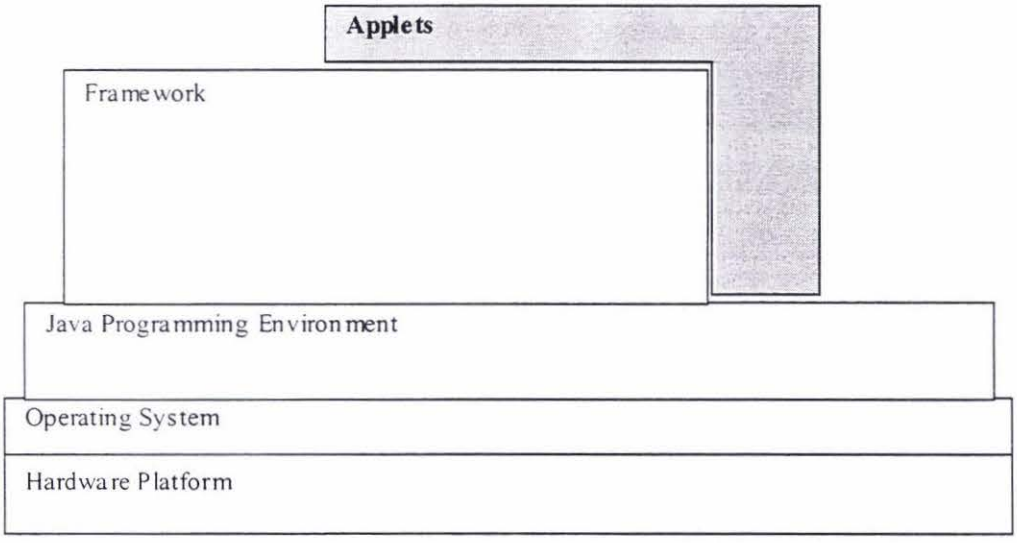

***Fig 1.1:** A framework as a layer of abstraction*

This thesis deals with the design and development of a Java image processing framework. This framework is intended to provide the programmer with a uniform programming interface, reusable components and a well defined architecture within which to create programs that perform image processing tasks. In particular, the framework will provide support for creating 'mini-tutorial' type applets suitable for interactively demonstrating image processing concepts in a Web-based learning environment.

Chapter 2 of this thesis analyses the requirements of the framework and introduces a data oriented model on which the design of the framework rests.

Chapter 3 presents the design of the framework. Alternatives are discussed where they arise and the actual design decisions made are justified. The various aspects and mechanisms necessary to the operation of the framework are described.

Chapter 4 demonstrates an example application of the framework. A mini-tutorial demonstrating the concept of a threshold operation on a greyscale image is devised, and an applet is built to accomplish this demonstration.

Finally, chapter 5 concludes the thesis and discusses the potential for future work, with three examples given.

2. Framework Requirements

In the previous chapter, the potential for Java applets to provide the interactive elements of a Web-based image processing course was identified. The idea of a software framework was then introduced as a way of easing the programming burden associated with creating these applets. This chapter explores the requirements of such a framework in terms of the nature and purpose of image processing software generally, and its application in a Web-based instructional environment in particular. Of significant importance is the introduction of a data oriented model for representing image processing tasks.

2.1 Image Processing Software

One of the major roles of image processing software is to provide a vehicle for the execution of image processing algorithms. Returning to the food processing analogy, an image processing algorithm may be likened to a recipe from a cookbook. A recipe specifies which ingredients to use when and what to do with them. Similarly, an image processing algorithm specifies the type and number of inputs, and sets out which image processing operations should be applied and in what order, so that the desired outputs are achieved.

2.1.1 Categories of Image Processing Software

Image processing software to realise these algorithms can be divided into three broad categories. First, there is application specific software for which an appropriate image processing algorithm or set of algorithms has been devised in order to accomplish a given task. For example, a machine vision system may use such software to locate edges and holes accurately for quality control purposes or robotic guidance. In this case, the software tends to be highly optimised, taking into account process specific factors and hardware platform constraints. There is generally little requirement for any sort of user-interface or user-oriented features. Processing speed is usually the key goal.

The second category comprises software for image processing algorithm development. This kind of software is used to *develop* the algorithms that are put to use in applications such as the machine vision example given above. Here the emphasis is on interactivity and flexibility, so that alternative solutions can be quickly and conveniently investigated during the search for an algorithm to solve a given image processing problem. These 'algorithm development environments' provide (often feature-rich) user-interfaces for combining operations and specifying inputs/parameters, and for viewing the results (including intermediate results). Such software is typically described as consisting of a 'supervisory system' for scheduling and invoking image processing operations together with a library containing the operations themselves (Cady, *et al.*, 1981). In addition, a range of data types used in image processing is generally supported, including different image types as well as scalars, histograms, chain codes, convolution kernels, lists, etc. Bailey & Hodgson (1988) make explicit the major requirements of any image processing algorithm development environment. Namely, such a system should provide a wide range of imaging operations; should be easily extensible when new operations are needed; should support a variety of data types found in image processing, not just images; and should be interactive.

The third category may be considered to lie somewhere in between the above two. This kind of image processing software provides a convenient tool for 'one-off' image processing tasks, for example touching up a scanned photograph or composing collages from multiple source images. A characteristic of software of this type is the absence of any need to remember or recall sequences of operations

(i.e. an algorithm). This distinguishes it from being focused only on algorithm representation and manipulation, and from being simply a canned-algorithm geared toward a specific job. Using this kind of software, the user needs only invoke the necessary operations once and the job is done.

2.1.2 Software for Interactive Image Processing Instruction

Existing Web-based image processing courses tend to rely on software in the second of these categories to supply the interactive aspects of their content (see for example Jordán & Lotufo (1996) or Fisher, *et al.* (1996)). Algorithm development environments make available a wide and extensible range of imaging operations, plus they provide the means to interactively experiment with these operations either in isolation or in arbitrary combinations. This allows any number of image processing concepts to be demonstrated to the user. Unfortunately, this pairing of Web-based course material with external image processing software in order to create an interactive learning environment has at least four main disadvantages.

Resulting learning environment is not seamless. In addition to the Web-browser program, the user must also run the image processing software that has been chosen to accompany it. From the user's perspective, this approach effectively creates a 'dual' user-interface. In the simplest possible scenario, these would each occupy a single window on the user's desktop. The user must then either arrange these windows so that they share the available desktop area, or must bring each to the foreground as necessary (thereby hiding the other). In the former case, the reduced size of the windows may have adverse effects on the usability of the respective programs. In the latter, the user may well find the arrangement too cumbersome if he or she has to swap views too often, and so may become discouraged with the learning process. Regardless, the courseware designer faces difficulty in trying to coordinate the theoretical and practical components of such a course because the delivery medium for each is separate. If the two could be combined the result would be a far more seamless learning environment.

Need to acquire and install the necessary software. Depending on the manner in which the course material is distributed, the user may or may not have access to the image processing software that goes with it. For example, a CD-ROM

containing the course could also contain the accompanying software. However if the material is being accessed over the Internet, the software would first have to be downloaded and installed, which could be off-putting especially for those with slow connection speeds. The host computer may also have certain access privileges preventing new software being installed, especially in the case of a shared system. Platform dependencies are a further drawback since the software may not have been written for a particular user's hardware and operating system, even though the Web-based component is still accessible. A final complicating factor is the need to comply with any licensing requirements associated with the use of a separate image processing program. Even 'free' software may be subject to a license that affects how it can be distributed.

Users may not follow instructions properly. An image processing concept may be well covered in theoretical terms, but to demonstrate it experimentally requires certain steps to be performed by the user, using the software provided. For instance these steps may consist of a sequence of commands to invoke, followed by suggestions on which of a number of parameters could be varied and how to accomplish this. Imparting this sort information to the user is not guaranteed to be an error-free process, especially if the user is unfamiliar with the software's user-interface. Any mistakes that are made could lead to either frustration on the user's part or unintended results that do not relate to the concept being taught. In the latter case, this may even cause confusion and thus hinder the user's learning experience.

User-interface is quite general. If it is known that a particular image processing concept is to be demonstrated, then it is almost certain that a user-interface can be designed that is specifically tailored for getting that concept across to the user. For example, the demonstration of a look-up table operation would benefit greatly if the user could specify graphically the function to use to apply the transformation. Unfortunately, it is likely to be very difficult (or even impossible) to provide these kinds of customised user-interface facilities within an algorithm development environment in every case. By its general nature, such an environment cannot be expected to account for the specific user-interface requirements of every image processing concept that it might ever be called upon to demonstrate. The result is that the user is presented with a very general user-

interface regardless of whatever concept the courseware designer may wish to demonstrate.

2.1.3 Java Applets for Interactive Image Processing Instruction

What this thesis proposes is the use of Java applets to take on the role of providing the interactive content for an 'improved' Web-based electronic textbook on image processing. Whereas before this 'practical' content was delivered using a separate image processing application, the envisaged applets will obviate the need for any such additional software, and will overcome many of the drawbacks associated with the old approach.

For instance, because applets can be displayed as integral parts of a Web-page (using a Java-enabled browser), they offer a far more seamless solution. Interactive image processing demonstrations can be literally interleaved with text-based content, providing better coordination between the practical and theoretical aspects of the course material, and hence stronger reinforcement of the concepts being taught. There is also no need to install any additional software. Instead, the Java-enabled browser automatically downloads the applet code as necessary (regardless of whether the course material is being accessed locally or over the Internet). Finally, the fact that a different applet can be used for each new demonstration means that: a) its user interface can be customised to best suit that demonstration's particular needs; and b) there is a much reduced chance of user error since he or she is no longer dependent on a general purpose style of user-interface.

The major drawback of this new approach is that there is a significant programming burden associated with actually developing the required applets. In the worst case, each applet for demonstrating a new image processing concept would have to be created from scratch. Producing the applets to cover even a moderate range of concepts using such an approach would quickly become a very tiresome effort. It is of course likely that after a short time at least some reusable code would emerge due to similarities in the way related concepts are found to be best demonstrated. However, it is even more likely that the development effort would remain ad hoc in nature and thus an unnecessary burden on the programmer, regardless of the number of applets produced.

An additional drawback is that the courseware designer may well lack the programming skills necessary to develop these applets. Knowledge of Java in general and applet development in particular is required, as is experience with actually implementing image processing operations. If existing image processing software is employed, designers need not acquire or call upon any additional programming expertise for developing the interactive elements of their course material.

In order to make the proposed applet-based approach more feasible, it is necessary to better equip the developer for creating these applets, and a way to do this is to provide a framework within which software of the required nature can be more easily and efficiently developed. While such a framework, on its own, can not entirely remove the need to write any code, it can, if properly designed, a) significantly reduce the amount of original code needed; and b) greatly simplify the effort required to generate that code.

In the previous chapter, the general nature of a software framework was discussed. This research aims to develop an 'image processing framework' which will assist in the development of programs for performing image processing tasks. A specific feature of this framework will be support for creating Java applets that incorporate the framework's image processing capabilities. In this way, developers can use the framework for creating applet-based image processing demonstrations for use in a Web-based learning environment.

2.1.4 The Basis of an Image Processing Framework

As a starting point for the development of the desired framework, it may in fact be useful to turn to the image processing algorithm development software the framework is intended to supersede. Designers of this kind of software often incorporate some manner of underlying framework into their designs, even if it is only implicit in nature. An underlying software structure such as this is essential in handling design complexity, and for ensuring that a design is extensible (which is an important feature of algorithm development software).

However, whereas with the design of such software the use of a framework is a means to an end, a mark of good software engineering practice, the framework

proposed in this research can be considered the end in itself. Rather than merely providing support for some larger application environment, the purpose of the framework is to explicitly aid developers in writing their own image processing programs. In particular, the framework will be used to create interactive applet-based image processing demonstrations.

The important observation to make is that, in the pursuit of an appropriate design for the framework, much can (and will) be drawn from the design of algorithm development software itself. For example, in section 2.2 below, a number of components typically found in such software are identified as key to the framework's image processing capabilities. Then, section 2.3 investigates a data oriented style of algorithm representation and finds it useful as a basis for the framework's overall design. Image processing algorithm development software is important precisely because it embodies the kind of functionality courseware designers need to deliver interactive-style content. Through the use of an explicit framework that embodies this same functionality, applet-based image processing demonstrations can be developed to provide the equivalent in interactive content, but in a form that avoids the drawbacks of using stand-alone algorithm development software.

2.2 Framework Components

As with algorithm development software, the framework will also need to support an extensible library of image processing operations, support a range of data types, and provide a supervisory system to schedule and invoke operations, and to maintain data in memory as it is processed. This 'core functionality' embodied by the framework represents the reusable code base upon which programs utilising the image processing capabilities of the framework can be built. In addition, the framework will need to add support for incorporating this functionality into the envisaged applets.

In this section, a number of application programming interfaces (APIs) are identified and discussed. These APIs essentially provide the means by which developers can make use of the framework's capabilities. The various components of the framework are able to work together because they conform to these APIs. It may be argued that these APIs are what makes the framework an

explicit software system, rather than just a purpose built platform for some image processing application.

2.2.1 Extensible Image Processing Operation Library

A kitchen lacking all manner of mixing bowls, chopping knives, baking dishes, etc., is fairly useless, but it is still a kitchen. Similarly, the proposed framework may be considered complete even without an associated library of image processing operations. What the framework must provide is a standard facility for implementing new operations as they are needed. This implies an API that allows a programmer to define the various attributes of a new imaging operation. Operations implemented in this way will be of immediate use within the framework, and may be re-used by distributing them among other framework users. Thus the operations may be considered distinct from the framework itself; the framework's design only needs to account for the necessary API.

Implementing a new image processing operation means specifying:

- The number of its inputs, the number of its outputs, their associated types, and the mathematical transformation or algorithm to apply. For example, an operation that adds a pair of images together would take two input images and produce a single output image by adding pixels in corresponding positions.

- Default values for one or more of its inputs, allowing its outputs to be evaluated using these defaults whenever actual input data is not available.

- Error conditions, for example it may or may not be an error to add images with different dimensions.

A well designed API would give the programmer control over all of these aspects of implementing new image processing operations.

This is a large part of the programmer's 'low-level' view of the framework. The corresponding 'high-level' view is a separate API (discussed further in section 2.2.3) that hides all of these implementation details, exposing only what is needed to actually use the framework to build image processing programs. Roman, *et al.* (1998) demonstrate this kind of separation, though their work relates to an image processing library rather than to a framework. They describe an object-oriented

class library written in C++ that represents a range of imaging operations in hierarchical fashion. In this way, a clean programming interface is exposed while neatly encapsulating the library's implementation details, thus hiding them from the user.

2.2.2 Data Types

In image processing terms, a data type refers to a particular class of storage where the internal structure of the data being stored is well defined. In addition, data types commonly have one or more attributes associated with them, which are used to further define a particular instance of data of that type. For example, the data to represent an image may be defined as being stored as a two-dimensional array of pixels. Any particular image would then have both width and height attributes associated with it to define its dimensions.

In fact, an image is not so much a single data type as it is a whole family of related types. This is because the pixels in an image may be stored, and therefore interpreted, in a variety of different ways. For example, a grey scale image may be stored using one byte for each pixel, giving 256 distinct grey levels. Alternatively, an RGB colour image may be stored using a 32-bit integer per pixel, using 8 bits for each of its red, green and blue components, with the remaining 8 bits unused. This demonstrates that a data type is defined not only by its internal storage structure and its various attributes, as indicated above, but also by the meaning that is attached to the data it stores. Only when all of this knowledge about an instance of a particular data type is known is it possible to perform processing on that data in a meaningful way.

There are many different types of data that are often used in image processing, not all of which are image types. For example, a histogram is a commonly used data type for gathering statistical information on an image. Histograms are used to store the frequencies with which pixels of each possible value (or range of values) occur within an image. A particular histogram is attributed with the number of 'bins' it contains, which determines how the pixel values are grouped when their frequencies are counted. In fact, even a solitary integer qualifies as a distinct type of data. Though it would have no attributes (in the sense described here), an

integer data type would still need to specify how the integer was actually stored (e.g. 32-bits).

Having well defined data types is a prerequisite for being able to implement image processing operations in a consistent manner within the framework. They effectively provide a 'protocol' that allows data to be passed unambiguously between one operation and another, with no confusion as to what is expected at any point. Among the defining attributes of an operation are the types of data expected at each of its inputs, and similarly the data types produced at its outputs. For the framework to be useful, a diverse range of data types is needed to support whatever image processing operations may be required.

As with the image processing operations in the previous section, these data types may be considered distinct from the framework itself. Again, what is required is an API that will allow whatever data types are needed to be created; that is, they are also extensible (this API also forms a part of the framework's low-level view). An important point concerning data types is that the internal representation used for the data they type is arbitrary, i.e. it is entirely up to whomever defines any particular type. Since the framework's data types are distinct from the framework, it has no way itself of knowing about the nature of a particular data type's internal structure. It is only the developer who implements a new operation, which uses a given data type, who must know about how to access that type of data. Hence each data type must also specify its own 'data access' API for the benefit of the operations that use it. Because the data types exist on their own, this Data Access API lies wholly outside the framework.

2.2.3 *A Supervisory System*

For the framework to be more than just a library of image processing operations and a collection of data types, it needs to provide the 'glue' that will allow these components to work together to perform image processing tasks. This glue makes the framework what it is, a complete image processing system ready to be customised to whatever application is required. The role of the framework's supervisory component is to *manage* whatever processing the developer wants carried out.

The 'richness' of the functionality provided by this supervisory system will impact significantly on the framework's ability to reduce the effort involved in developing new image processing programs. The major responsibilities of the supervisory system are two-fold. These are *scheduling*, which involves:

- Scheduling and invoking operations in the required order. For example, if a particular operation's parameters are modified, or its inputs change, it will need to be re-invoked, as will all operations that depend (directly or indirectly) on that operation's outputs.

- Managing the use of multiple threads[3], for instance to invoke multiple independent operations simultaneously, or to increase the responsiveness of the program's user-interface. These will need to be allocated, assigned and synchronised correctly.

and *data passing*, which involves:

- Passing intermediate data between dependent operations. Old data has to be discarded and newly processed data made available to the operations that are expecting it. This may involve allocating additional storage in memory and reclaiming unused memory.

- Handling automatic type conversion of data. This enables an operation's inputs that normally expect a certain data type to intelligently deal with alternatively typed data. For example, an input expecting a histogram type could accept an image type if a histogram can automatically be derived from that image and then presented in its place (automatic conversions can only apply where a sensible conversion exists, and has been defined by the respective data types).

Depending on the capabilities of the supervisory system, the responsibility for correctly handling all of this detail may rest with the developer, with the framework, or it may be shared in some proportion between the two.

[3] A *thread of control* is a common programming abstraction that enables a program to carry out multiple tasks as if they were being performed simultaneously. Whether or not this actually occurs is determined by the number of physical processors available to execute the program.

The goal of the framework proposed herein is to assume as much of this responsibility as possible. That is, to unburden the developer so that he or she need only concentrate on the high-level design of his or her program, and all but ignore the lower level details. The justification for this is that the framework is intended to be easy to use. Courseware developers may not necessarily be proficient programmers, and therefore the framework aims to hide much of the necessary details associated with writing image processing programs. This is intended to make applet development using the framework as simple a process as possible.

There are a number of implications arising from this approach. For instance, the design and development of the framework will be necessarily complex. By abstracting this complexity away from the developer, there remains far less opportunity for programmer error, be it due to inexperience, lack of the appropriate skills, or simply ad hoc design. As a corollary of this, the framework will have to impose a fairly rigid architecture on the programs that use it. Only in this way will it be able to simultaneously provide a high-level abstraction for the developer *and* coordinate the various low-level details necessary to support the developer's programs. This in turn implies an inherent limitation on the scope available to developers for imposing their own software designs using the framework. Figure 2.1 illustrates the proposed framework's intended architectural complexity.

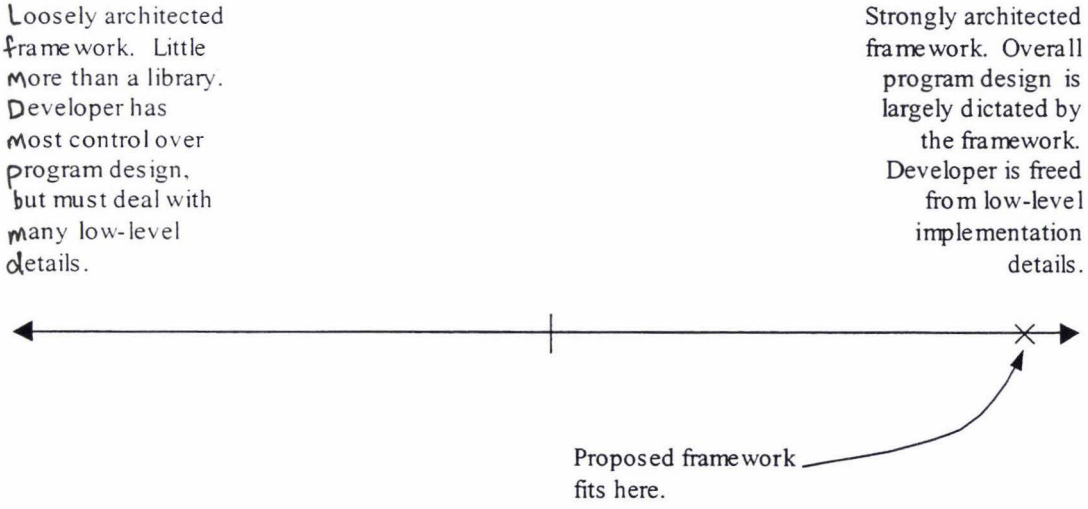

Fig 2.1: Continuum of framework architectual complexity

The API that exposes the capabilities of the framework's supervisory system, and thereby the capabilities of the framework as a whole, is its Developer API. From the perspective of the developer who wants to utilise the framework for creating image processing programs, this is the framework's most visible aspect. It provides the framework's high-level view (alluded to in subsection 2.2.1) which allows the developer to call upon the framework to carry out the required image processing tasks. The applet support discussed in the next subsection is also conceptually at the level of the framework's Developer API, and so may be thought of as supplementing it for the specific purpose of applet development using the framework. Figure 2.2 below gives a structural representation of the framework as has been discussed this far.

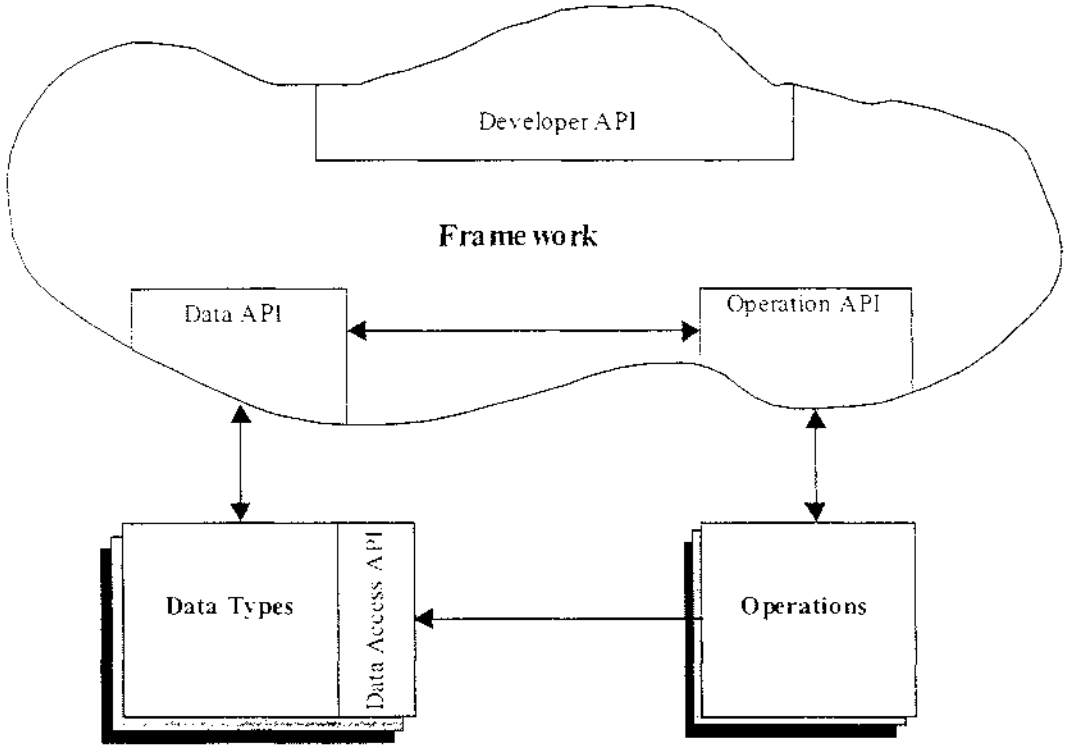

Fig 2.2: Idealised representation of the framework

In this diagram, the amorphous blob represents the framework, with everything inside it subject to the framework's supervisory system. External to this are the framework's data types and operations, which are defined via the relevant framework APIs. These APIs, including the framework's Developer API, are in fact part of the framework. The Data Access APIs, however, are data type dependent (there is one for each data type) and are therefore not part of the framework.

The double arrows between the framework and its external components (data types and operations) represent the fact that they are defined in terms of their respective APIs. Further, these APIs are the means by which these components interact and communicate with the framework (and in particular the framework's supervisory system). The double arrow connecting the Data API and the Operation API represents the supervisory system's means of handling details such as passing data between operations, detecting when default values should be used, automatic type conversions, etc. Finally, the diagram shows how an operation gains access to the data associated with a particular instance of a data type through the use of its Data Access API (the single arrow). As a whole, this diagram represents a form of high-level 'roadmap' indicating where the various framework components fit and their relationships to each other.

2.2.4 Applet Support

Java provides support for creating applets as part of its core language features. What the framework needs to provide are the necessary hooks to enable its image processing capabilities to be incorporated into applet form. Support for applet creation can be considered a specialist extension of the framework's Developer API. Among the roles of this extension are:

- Facilitating the integration of the necessary framework classes with Java's core applet classes.
- Handling input events from the applet's user-interface controls, as well as providing the means to view output (e.g. images, other processed data, etc.) via the applet's display context.
- Handling the applet's life-cycle, which is part of the requirements of the applet-browser protocol (this is explained in more depth in the next chapter).

By considering applet support as an extension, rather than as an integral part of the framework, the framework's conceptual foundations are strengthened somewhat. This is because, at its core, the framework's role is to facilitate the creation of programs that carry out image processing tasks. Applets for interactively demonstrating image processing concepts are just one possible

application of the desired framework. By making this important distinction, it is felt that a sounder overall design can be achieved.

Of course, because the desired applets are the motivation for the framework, it is reasonable to expect that the framework's design will be guided somewhat by their specific requirements. For example, demonstrating image processing concepts interactively means providing the user with a responsive system. In this sense it may not be optimal to make the user wait for an entire image to be processed before intermediate results can be viewed. This is especially important where a user is varying a parameter quickly but the recalculation necessary after each change is relatively slow. For this reason, the framework's design must place a priority on responsiveness.

Another example, though from the developer's perspective rather than the user's, is that the applet developer must find the framework straight-forward to use. The fact that the developer may not be an expert programmer has already been discussed. This, again, is a strong argument for the framework assuming much of the responsibility for handling the low-level implementation detail involved in creating image processing programs.

2.3 A Data Oriented Model

A defining aspect of any such software is the scheme it provides for algorithm representation. This is the 'language' that is used to specify image processing algorithms using the software. It could be argued that the choice of scheme will do most to shape that software's user-interface, since this will determine the nature of the tools that are provided for generating and manipulating algorithms using the software.

In turn, the desired framework will require an analogous scheme for representing image processing tasks; and, as with algorithm development software, the choice that is made will have a defining effect on the framework's design. This section discusses the possibilities for a scheme that is appropriate for the framework.

2.3.1 Algorithm Representation

Figures 2.3a and 2.3b below illustrate two alternative ways that image processing algorithms can be represented. The process oriented view emphasises a sequential ordering of operations, thereby giving a specific order in which operations may be invoked to execute the algorithm. Algorithms represented in this way typically appear as sequences of commands, which is most suited to algorithm development environments that offer a command line style of user-interface (for example, Serendip (Wilson, 1987) and VIPS (Bailey & Hodgson, 1988)).

1. Load Image into **A**
2. Let **B** equal **A**
3. Smooth **A**
4. Subtract **A** from **B**
5. Display **B**

Fig 2.3a: *Process oriented view of an algorithm*

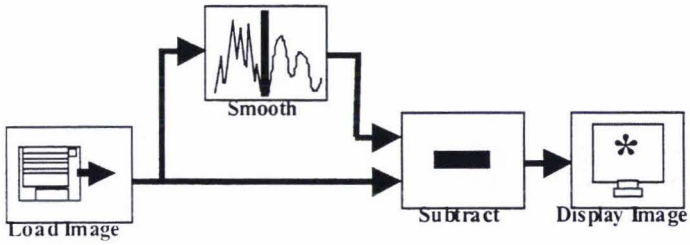

Fig 2.3b: *Data oriented view of the same algorithm*

The data oriented view, on the other hand, emphasises the flow of data through an algorithm. Parallel paths (i.e. those that may be executed concurrently) are made explicit using this representation. Due to the inherently pictorial, visual nature of this representation, it is most suited to algorithm development environments that employ a visual language style of user-interface (for example, HI-VISUAL (Monden, *et al.*, 1984), Cantata (Rasure & Williams, 1991) and OpShop (Ngan, 1992)).

Unsurprisingly, neither of these schemes, or their associated user-interface styles, is wholly 'better' than the other. For example, Ngan (1992) finds that the process oriented view favours a very compact, textual representation of algorithms, and that a text based user-interface allows commands to be issued quickly provided their syntax is well known. On the other hand, the data oriented view leads naturally to a visual style of algorithm representation, which encourages a more

natural, heuristic approach to algorithm development. In terms of user preference between the two, one differentiating factor may be the user's level of experience, where the former style is sometimes favoured by expert users who desire more programmatic control over their environment. Less experienced users, by contrast, may find that a visual interface presents fewer barriers to learning and becoming productive with the image processing software being used.

In carrying out a survey of human computer interface techniques for image processing algorithm development, Ngan notes that a principle motivation behind the development of visual languages is in fact to make programming easier. Representing programs visually offers a very intuitive and user-oriented scheme for specifying the 'instructions' that make up a program. Because data oriented schemes lend themselves to visual representation, they have proved valuable in designing image processing algorithm development environments that are often much easier and more intuitive to use.

Although this research is not concerned with the development of a visual language as such, it is felt that the adoption a data oriented model can nevertheless translate into similar benefits for the user of the framework. The framework's Developer API, for instance, may rightly be considered its 'user-interface'. After all, it is through this API that the intended user of the framework (i.e. the developer) is able to harness the framework's capabilities. By resting the design of the framework on a data oriented model, this research hopes to provide the developer with an intuitive and easy-to-use API for writing image processing programs.

2.3.2 *A Dataflow Framework*

The process oriented form of an algorithm typically employs *variable references* to label its various pieces of data; these are usually given names that convey some meaning about their contents. Using a data oriented scheme, however, the data in an algorithm *flows* between operations via interconnecting links. This connection paradigm is central to the design of the framework. Figure 2.4 illustrates how an interactive image processing demonstration could potentially be represented using this paradigm. An important feature of the paradigm is that the 'operations' constitute not only image processing operations but also user-interface elements, file I/O operations, etc. By considering an operation to be any unit of functionality

that can be 'connected', the framework provides a uniform scheme within which image processing programs can be created. These operations, and the connections between them, are specified using the framework's Developer API.

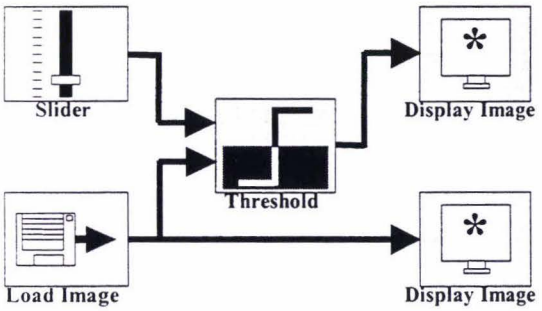

Fig 2.4: Interactively thresholding an image

The diagram in figure 2.4 depicts a threshold operation with two inputs and one output. Of its inputs, one is supplied via a 'load image' operation, which loads the image to be thresholded; the other is connected to a slider control, which supplies the level at which to threshold the image. The two 'display image' operations display the original and thresholded versions of the image. Thus the user can control the threshold level and inspect the results interactively. Figure 2.5 sets out how a simple user-interface could be designed that incorporates the features to support such a demonstration. The role of the framework is to allow the developer to easily implement a design such as this in applet form.

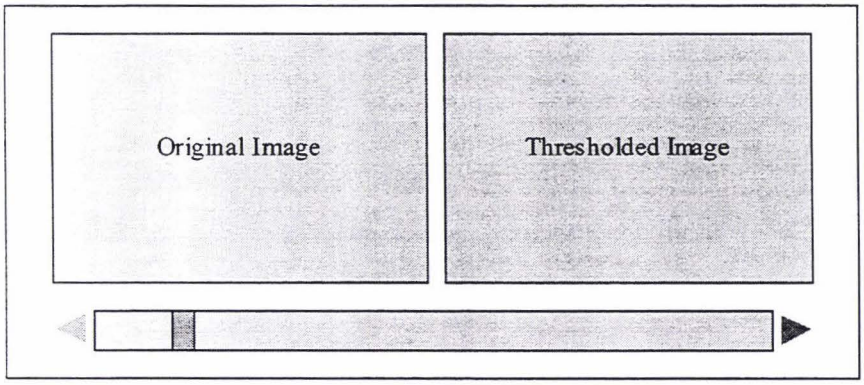

Fig 2.5: User-interface design for demonstrating a threshold operation

As well as providing developers with an intuitive scheme for specifying image processing programs, a data oriented model maps conveniently to an object-oriented design. The operations in figure 2.4, for example, could easily consist of

a set of object instances, each derived from an 'operation' superclass, with the connections between them implemented as object references. The potential for a natural object decomposition is advantageous because Java, the intended implementation language, is object-oriented. A further advantage of adopting a data oriented model is that much of the information required to schedule operations concurrently is inherent in such a model. A framework based on this model would therefore be well positioned to make use of Java's language level support for multi-threading.

A data oriented approach does have disadvantages however. Data persistence, for instance, means that the framework must be carefully designed to avoid over-allocating system resources, especially memory. This problem arises because, given a network of connected operations, the data at each node is conceptually in a unique state. A naïve implementation would therefore maintain each different version of a piece of data in memory as it was processed, which is not optimal especially for large images. Rather, memory needs to be released when it is no longer needed. Unfortunately, determining when this should occur is a non-trivial problem given a data oriented approach.

Another disadvantage of this approach is that there is an inherent difficulty with representing loops and conditional operators in data oriented form, both of which are found in image processing algorithms. For these types of operation, whatever solution is used will tend to be clumsy relative to its process oriented equivalent. Both of these drawbacks lead to a more complex framework design.

2.4 Summary

The initial motivation behind this research is essentially an improvement in the use of image processing software for delivering the interactive content of Web-based image processing courseware. In this chapter, the drawbacks of using a separate algorithm development environment have been identified, a new applet-based approach has been discussed, and an image processing framework has been proposed to assist in developing the applets for implementing this new approach.

What the framework does *not* represent is a complete implementation. Rather, it provides a stepping-stone that brings the programmer closer to an implementation,

without imposing its final form. It achieves this by sitting between the programming environment provided by Java and the various applet implementations. The benefit to the programmer is that there is no need to start from scratch each time an applet for demonstrating a new image processing concept is required. Instead, the framework is employed and its code is re-used, effectively by-passing a large portion of the programming effort involved. Code re-use has the added benefit that there is less opportunity for programmer error and thus less chance for bugs to appear.

The effect of the abstraction provided by the framework is to *hide* a lot of the implementation detail that the developer would otherwise be forced to deal with directly. The framework achieves this by providing an application programming interface (API) which enables the developer to take advantage of its image processing capabilities. This is known as the framework's Developer API. From the developer's perspective, this API is the framework's most visible aspect.

Also identified in this chapter are a number of additional APIs that make up the framework. By conforming to these APIs, the various framework components gain access to, and are able to work with, the functionality provided by the framework as a whole. Also, by conforming to key aspects of these APIs, the framework's external components (operations and data types) are able to be used in a reliable manner by the framework. Having investigated the requirements of the framework in terms of its constituent components, this chapter recommends that design and development be broken into five main areas:

- ***Operations***. Operation API for defining new operations; specifying the processing for an operation to carry out; creating an operation's inputs and outputs.

- ***Data Types***. Data API for defining new data types; Data Access APIs that provide access to the data being typed; common data types found in image processing algorithms.

- ***Data Passing***. Establishing connections between operations; Data API for accessing an operation's inputs and outputs; providing a responsive system; support for default input values and automatic data type conversions.

- ***Scheduling***. Invoking operations; multi-threading; synchronising between dependent operations.

- ***Applet Support***. Developer API; integrating the framework's capabilities with Java's own support for applets.

Underlying the framework's design is a data oriented model. This implies a framework that defines image processing tasks in terms of the operations required to carry out a task, and the connections between them. Data then 'flows' between operations depending on how they have been connected together. The main justification for this approach is that it is intended to lead to a more intuitive and easy-to-use system. The following chapter presents the design of the framework based upon what has been learned this far.

3. Framework Design

3.1 Introduction

At the outset of this research, what was desired was an electronic textbook on image processing that combined a variety of media types, including interactive content, into an integrated learning environment that utilised 'the Web' as its delivery medium. The project was called 'HIPER', standing for Hypermedia Image Processing Electronic Resource. The research in this thesis aims to provide a framework for developing the interactive elements for HIPER. This chapter presents the design of this framework based on what has been learned in the previous chapters, including its various class hierarchies, programming interfaces, and mechanisms.

3.1.1 Tools

In developing the framework, at least three tools played important roles and are worthy of mention here. The first is simply the whiteboard. During development, the whiteboard came to be an indispensable tool for capturing (often abstract) ideas about the form that the design should take. In particular, it provided an extremely useful forum for collaborative design work where two or more people were contributing. In addition to the use of the whiteboard, two software packages were used extensively during the development of the framework; these were:

Borland JBuilder. As an integrated development environment (IDE) for Java, JBuilder (see figure 3.1) proved to be both powerful and easy-to-use. Its features include a sophisticated project management system and browser; seamless two-way visual design editing with full Javabean support; and a host of useful wizards for everything from creating new classes to software deployment to Javadoc support. All code for the framework was developed within the JBuilder environment.

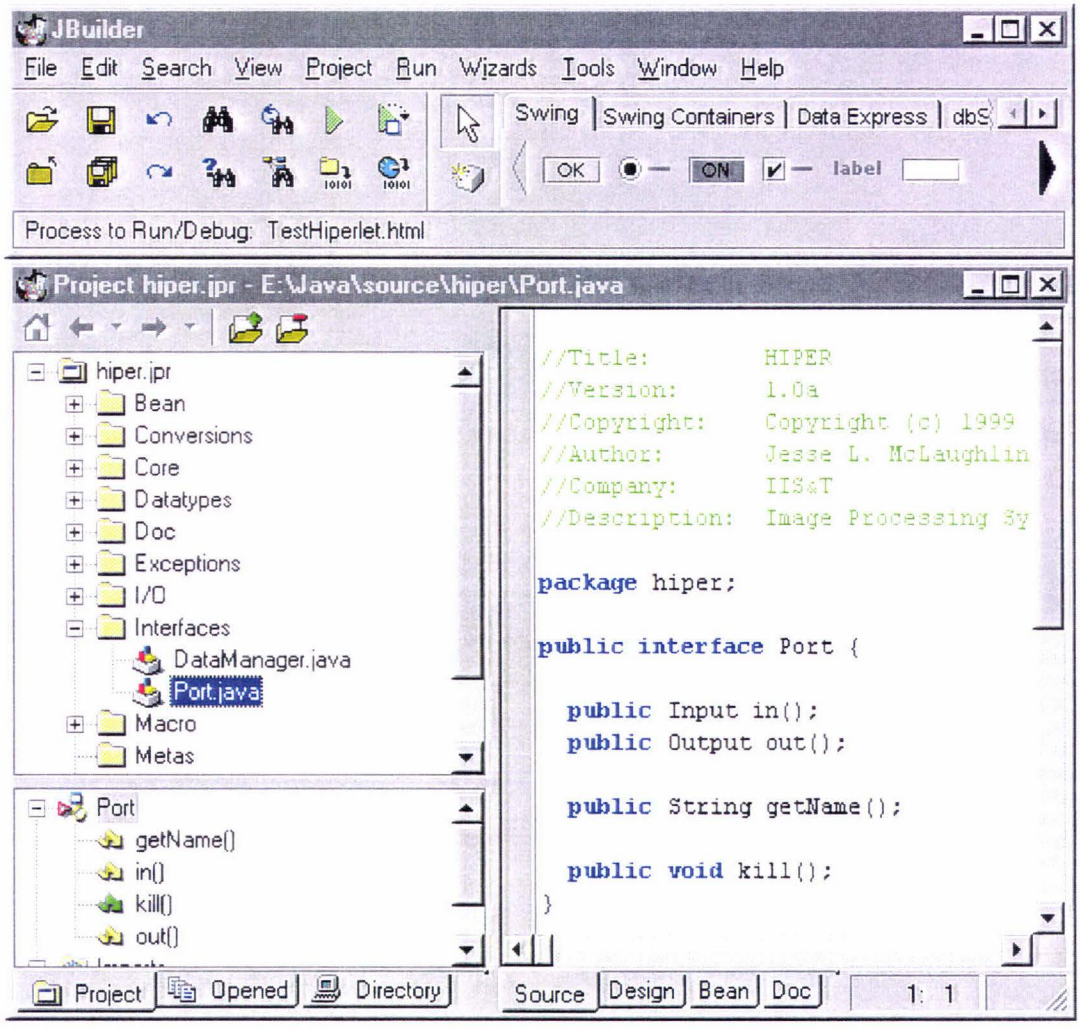

Fig 3.1: *Borland JBuilder*

Rational Rose. The Universal Modeling Language (UML) (Quatrani, 1998; Booch, *et al.*, 1999) is a powerful tool for designing object-oriented systems and software, and was used to formalise many aspects of the framework's design. The Rose software was used primarily as a UML editor. Its features include graphical tools for editing UML models; an intuitive model browser; and, significantly, support for Java in the form of a pre-built model of the Java core classes. Rose's

Java support also includes an automatic Java code parser/generator for round-trip engineering; although this feature, despite being potentially useful, was not used. The following section presents a guide to the UML notation that is used in this thesis.

3.1.2 UML Notation

The *class diagrams* in figures 3.2a to 3.2e (below and over page) each show a different kind of *relationship* that can exist between classes in an object-oriented design. Specialisation simply means that a class is derived from a superclass. This relationship is used to build inheritance hierarchies (subclasses inherit all of the methods and attributes of their superclass, and may add their own). Realisation represents another form of inheritance, though in this case only methods are inherited. This relationship is used where a class *realises* an interface (i.e. a set of methods that the class must implement). Aggregation represents a 'whole-part' relationship. This applies where a class is attributed with another class; the cardinality of this relationship is also sometimes shown, either by a number or by giving the name(s) of the attributed part(s).

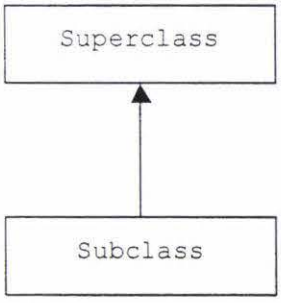

Fig 3.2a: *Specialisation*

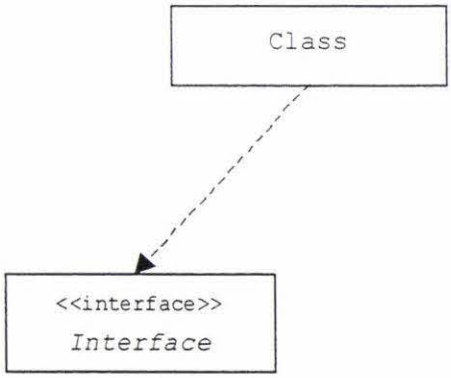

Fig 3.2b: *Realisation*

Fig 3.2c: *Aggregation*

Fig 3.2d: *Association*

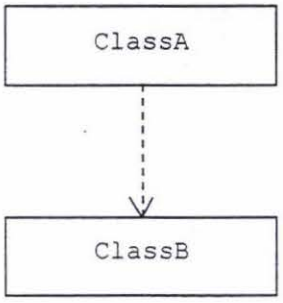

Fig 3.2e: *Dependency*

An association is similar to an aggregation, except that there is no 'whole-part' distinction that can be made at either end (rather, both parties to the relationship have an equal role to play). Finally, a dependency shows where one class depends on another class; this relationship may also be read as 'uses', meaning in this case that ClassA uses ClassB.

A class may reveal its name only, or its attributes and methods may also be shown (see figure 3.3). UML class notation partitions a class three ways. At the top is its class name. If a class' name is in *italics*, then the class is *abstract*. This means that it exists only to be subclassed, not to be instantiated directly. Underneath this are the class' attributes; both their names and types are given. The class' methods are placed furthest down; their names and return types are given, however their parameter types are (usually) omitted. Abstract methods are shown in italics.

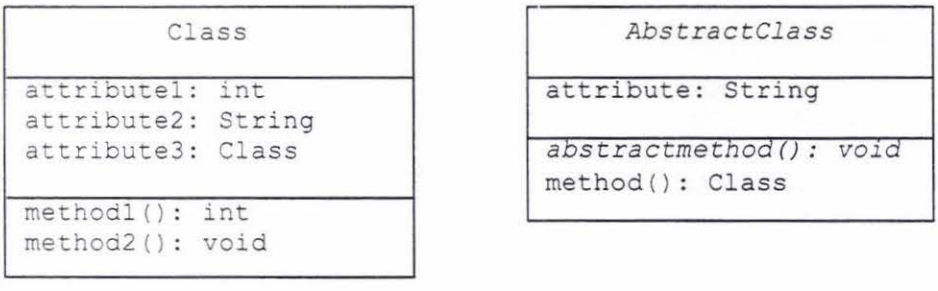

Fig 3.3: *Class Notation*

Figure 3.4 illustrates a UML *object diagram*. This is similar to a class diagram except that it deals with the relationships between objects rather than classes (objects are distinguished from classes by underlining). Each object is labelled with an instance name followed by the class it belongs to. Object diagrams usually only use associations to show the relationships between various objects; they may also show the values of an object's attributes. Because they deal with instances of classes as opposed to the classes themselves, object diagrams effectively represent a *static moment* or *snapshot* during a program's run-time. In this way, they represent a kind of scenario.

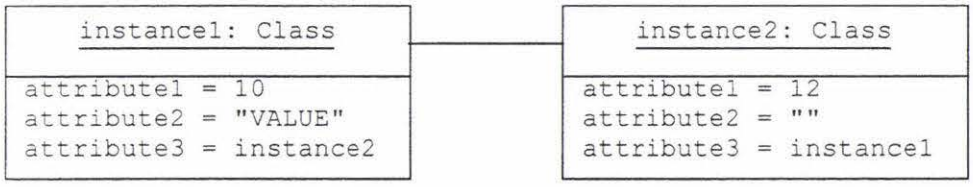

Fig 3.4: *Object Diagram*

A UML *sequence diagram* is another kind of scenario. A sequence diagram demonstrates one possible sequence of method calls among a group of related objects (see figure 3.5). The objects involved in the scenario are arranged in a row at the top of the diagram. The vertical dashed lines descending from each object represent their *life lines*; a line that terminates indicates an object that is no longer needed and can therefore be destroyed. The horizontal arrows in the diagram represent method calls. Each is labelled with the name of the method being called. An object may call a method on itself, which is indicated by an arrow that turns back on itself. The tall thin rectangles represent an object's *focus of control*. These indicate the duration of an action performed by an object, either directly or via a subcall. Nested foci of control indicate that a method is being called from within another method on the same object.

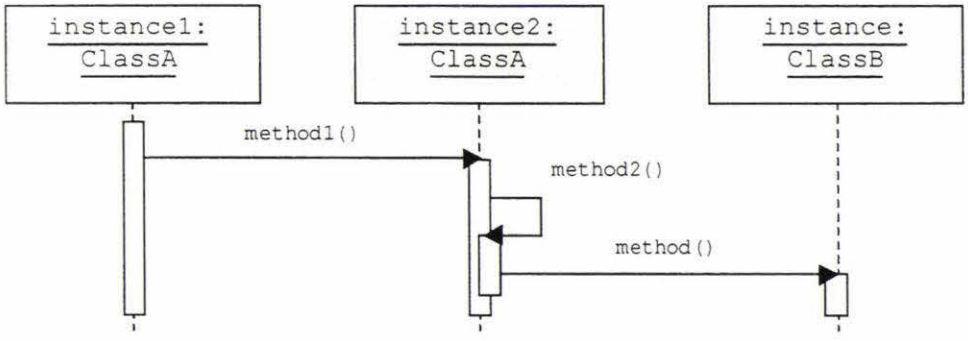

Fig 3.5: *Sequence Diagram*

3.1.3 Methodology

During development, advances in the framework's design tended to proceed in an iterative manner. That is, a problem or sub-problem would be identified, a potential solution would be found, then that solution would be implemented and evaluated to determine whether or not the problem had been solved. The actual steps that took place tended to follow the pattern outlined below:

1. *Identify problem or sub-problem.* This consisted of open, unstructured discussion covering current progress, as yet unaddressed features, or areas that required improvement. If this was a repeat iteration for a given problem then it covered the nature of problem, the previous iterations, reasons for short-comings of current solution, and so on.

2. *Find potential solution.* This was discussion guided by the nature of the identified problem, together with whiteboard sessions. The whiteboard was used to collate ideas using free-form drawings, standard (UML) or arbitrary notations, and pseudo-code. Very often a solution was arrived at in terms of new class/object structures or relationships.

3. *Implement and evaluate.* This step coded, tested, and demonstrated a proposed solution. Depending on the results, either a new problem was selected for step 1, or a new iteration for the current problem was begun, using what had been learned.

Although, the design process was iterative in nature, it is very difficult to convey with accuracy its actual unfolding since a thesis constrains material to be presented in linear form. What is presented here is the result of many *evolutionary* stages, rather than some strictly sequential process. Rather than try to make these stages explicit, alternatives are discussed where they arose, with justification for design decisions given.

This chapter is broken into sections that reflect the five areas identified in the previous chapter. These areas are not necessarily distinct; rather, there is some overlap, since each must work together and is therefore not completely independent. The order in which they are presented is intended to provide the most logical progression through the design and development of the framework.

3.1.4 Definitions

To further assist a proper understanding of this chapter, a fixed width font has been used where the text refers to some concept that actually appears in the framework's implementation, for example it might be a `class`, `attribute` or `method`. This style is also used where example fragments of Java code are used.

In addition, a distinction has been made between the terms *programmer*, *developer*, and *user*. For the purposes of this chapter, these terms hold special meaning. The developer is the person who utilises the services provided by the framework to develop programs, for example, applets for interactively demonstrating image processing concepts. The programmer, on the other hand, is the person who extends the framework, for example by adding new operations or data types, which developers can then use in their programs. Finally, the user is the person who makes use of these programs; this may be a student who is taking advantage of such interactive image processing demonstrations. This distinction is intended to avoid any unnecessary confusion arising from the use of these terms.

3.2 Operations

This section introduces the concept of an operation as the 'building block' of the framework. It then describes a class hierarchy that allows these building blocks to be defined, either by writing code to carry out the required processing, or by grouping previously defined operations to create more complex ones. This part of the framework's design lays the basis for both its extensible image processing operation library and its Developer API.

3.2.1 Operator

In designing a dataflow oriented framework, a natural starting point is a 'processing block' that has inputs and outputs, and that can be connected to other blocks. For this, the framework defines its most central abstraction: the `Operator` (see figure 3.6). In terms of the framework, an `Operator` may be any image processing operation, *or* it may be *any other* unit of functionality that can be 'connected' into a network, for the purpose of representing some image processing task (as illustrated previously in figure 2.4).

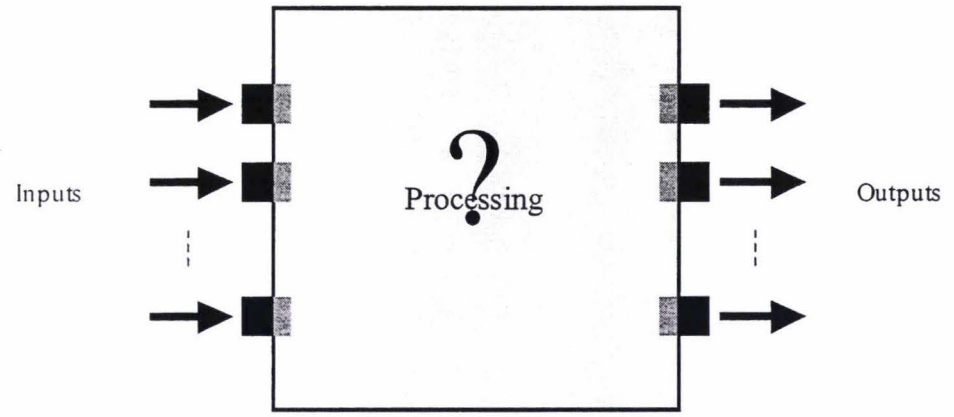

Fig 3.6: *`Operator`*

`Operator`'s principle characteristic is that it can be connected to other `Operator`s, via its inputs and its outputs. As such, it provides the foundation for the framework's Developer API. Using the framework, developers may build or modify their programs by calling on `Operator` objects to establish or dissolve connections amongst themselves. The framework then takes care of processing inputs, passing intermediate data, producing outputs, etc., depending on how the connections between the various `Operator`s in a program have been configured. Figure 3.7 shows how the framework's `Operator` class may be represented using UML[4].

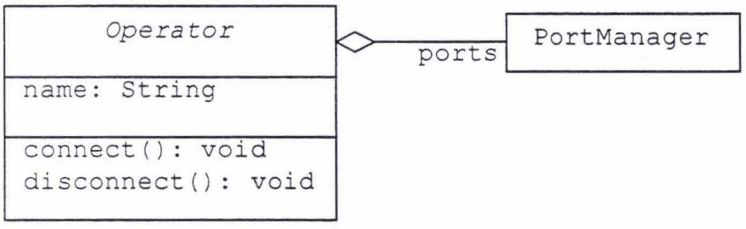

Fig 3.7: *The `Operator` class*

`Operator`'s `name` attribute is essentially a descriptive element. It may be used as a comment field to hold 'human-oriented' information concerning a particular Operator (that is, it's contents are not used by the framework for any specific purpose). At the very least, it enables more meaningful debugging information to be generated during development. The two methods shown here, `connect()` and `disconnect()`, form the basis of the framework's Developer API. These

[4] Note that this is not a complete picture of the `Operator` class, rather just what is relevant at this point. As a modeling language, UML is intended to be used to show only what needs to be viewed at any time, while hiding that which is irrelevant.

methods allow the connections between a group of `Operator`s to be configured to suit a given task.

`Operator` has one additional attribute: its `ports`, which refer to its inputs and outputs collectively. An `Operator` manages its `ports` through an instance of the framework's `PortManager` class. The role of `PortManager` is primarily to allow an `Operator`'s ports to be assigned names (represented as strings). Providing the ability to use names to refer to framework entities, especially those that developers will need to deal with consciously (in this case an `Operator`'s inputs and outputs) offers a developer-friendly strategy for making the framework more intuitive and easy-to-use overall. For instance, it is a more meaningful solution than simply using integers to distinguish among an `Operator`'s inputs or outputs (which was the initial approach taken).

As well as being able to connect to and disconnect from other operations, any particular operation is also characterised by the actual processing it carries out (this is also indicated in figure 3.6 above). Initially, an `Operator`'s processing characteristics were to be defined by the programmer, who would subclass `Operator` and then write the code to implement whatever processing was required. However, it was realised that this is just one of two ways that an `Operator` can be defined. The alternative is to consider a network of connected `Operator`s to be itself a single `Operator`. That is, to 'encapsulate' a number of 'primitive' operations into a single more complex one. Of course, this 'meta operation' would appear identical to any other operation, with its own set of inputs and outputs. For this reason, the ability to define an `Operator`'s processing logic is deferred to a pair of classes that subclass `Operator` (see figure 3.8) [5].

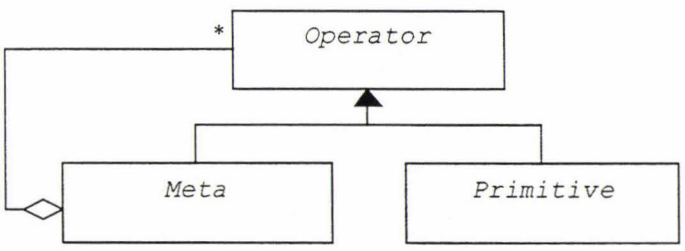

Fig 3.8: *`Operator` class hierarchy*

[5] The `Operator` class hierarchy is an example of the *Composite* design pattern (Gamma, *et al.*, 1995).

The two types of `Operator` are subclassed from `Operator`, so each inherits the ability to have inputs and outputs, and to connect to other `Operators`, etc. The difference is that a `Meta` aggregates a number of existing `Operators` (zero or more, as indicated by the asterisk) in order to define its processing logic, whereas a `Primitive` relies on the programmer to actually write some code that will be used to do its processing. Programmers who wish to extend the number of operations available for use with the framework may do so by subclassing either `Primitive` or `Meta`. It is not possible to subclass `Operator` directly to provide a new operation since `Operator` does not define any way of specifying what processing to carry out. The principle role of these subclasses is to provide the base functionality upon which all operations for use within the framework can be implemented. In this respect, they constitute the basis of the framework's extensible operation library. A comparison between figure 3.9 (in section 3.2.2) and figure 3.11 (in section 3.2.3) illustrates the conceptual differences between these two important subclasses.

3.2.2 *Primitive*

A `Primitive` encapsulates the concept of an operation that cannot (or should not) be broken into any smaller sub-operations. Hence, a programmer who wants to implement such an operation must physically write the code to carry out the required processing. An operation that adds two images is a good example of a `Primitive`. The concept of a `Primitive` as an `Operator` that carries out its own processing is illustrated in figure 3.9.

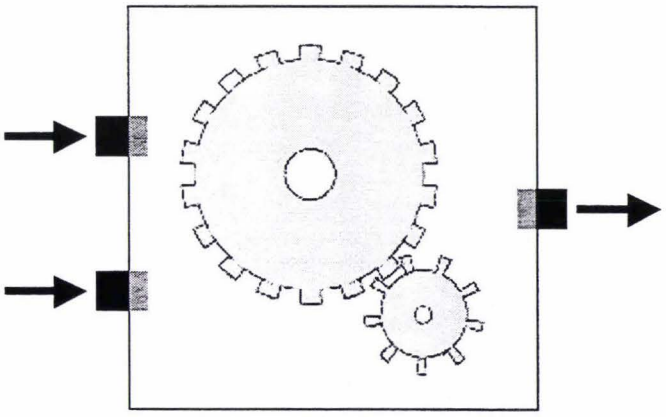

Fig 3.9: *`Primitive`*

`Primitive`s are where all the *actual* processing ultimately takes place, hence this class also provides a vehicle for much of the scheduling logic needed to invoke a particular `Primitive`'s processing code.

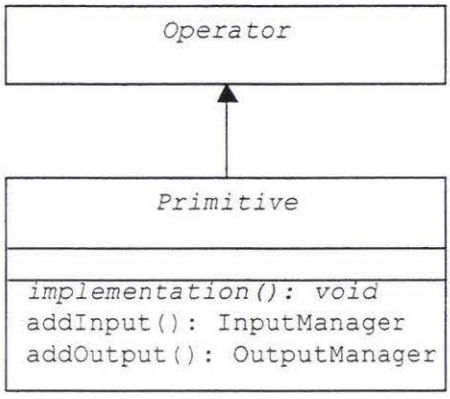

Fig 3.10: *The `Primitive` class*

The `Primitive` (see figure 3.10) class has three methods that are of interest at this point. First, it defines an abstract `implementation()` method, which must be overridden when `Primitive` is subclassed. This method is called by the framework's scheduling logic when a given `Primitive` should begin its processing. Hence this is where the programmer may implement whatever algorithm or logic is required to produce an operation's outputs from its inputs. The framework provides the programmer with access to a `Primitive`'s inputs and outputs via its `InputManager` and `OutputManager` classes respectively. These classes allow the programmer to access incoming data, initialise outputs, etc. These classes and the methods they provide will be described in more detail in section 3.4 on data passing.

Instances of the `InputManager` and `OutputManager` classes are created via calls to a `Primitive`'s `addInput()` and `addOutput()` methods. These methods are intended to be called from a `Primitive`'s constructor, allowing the programmer to set up the number and type of a `Primitive`'s `ports` at the time it is instantiated. In this way, the framework's Operation API defines a two-step process for creating a new `Primitive` operation: 1) subclass `Primitive` and implement the new class's constructor, wherein its inputs and outputs can be created; then 2) implement the new `Primitive`'s `implementation()` method, using the I/O managers created previously to gain access to its inputs and outputs.

3.2.3 Meta

A `Meta` (figure 3.11) relies on previously defined operations to carry out its processing. An example of the use of a `Meta` might be to create an 'unsharp-mask' operation by combining a smoothing operation with a subtraction operation. Note that since a `Meta` is an `Operator` itself, then by definition `Meta`s may contain other `Meta`s, and therefore the relationship is recursive. The framework itself places no intrinsic limitations on the depth to which `Meta`s may be nested.

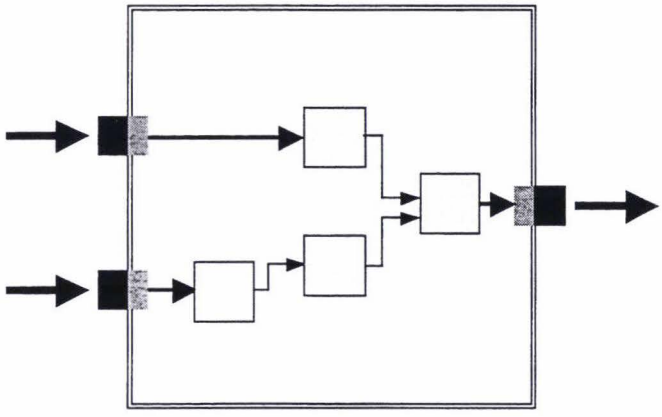

Fig 3.11: *`Meta`*

As shown in figure 3.12, `Meta` defines the methods `addOp()` and `removeOp()`. These are used to add `Operator`s to and remove `Operator`s from a `Meta`. Also present are methods for adding inputs and outputs to a `Meta`. These are different from their counterpart methods in the `Primitive` class in that these methods do not return anything. This is because a `Meta` itself never needs to access its inputs or outputs; rather, all processing that occurs within a `Meta` is performed by its

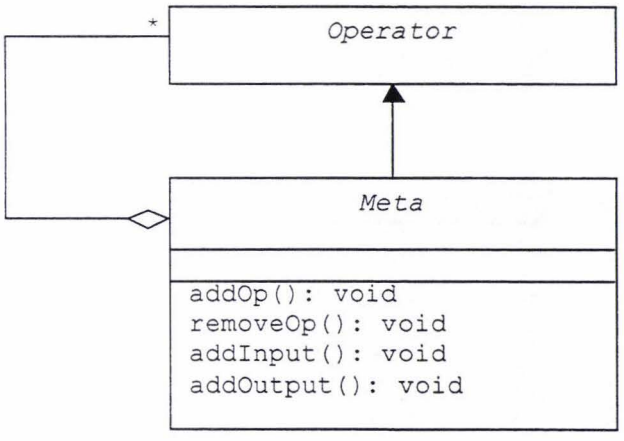

Fig 3.12: *The `Meta` class*

member operations. A new `Meta` may be created by subclassing `Meta` and implementing the new class' constructor. Typically, this constructor would set up the new `Meta`'s inputs and outputs, add its member operations, and then configure the connections between them.

From the outside, `Meta`s and `Primitive`s are indistinguishable, and internally each represents an alternative for defining the processing characteristics of a particular `Operator`. A `Meta`, however, never carries out any processing directly; rather, it's role is to contain (and therefore keep track of) a number of 'encapsulated' operations. As such, this class fills a number of support roles within the framework's design where these characteristics are desired. These roles will be identified and explained in future sections where they become relevant.

3.2.4 Summary

This section has introduced three important (and related) classes belonging to the framework's design; these are `Operator`, `Primitive` and `Meta`. `Operator` provides the basic abstraction of a processing 'block', which can interconnect with other `Operator`s to form a dataflow network for carrying out image processing tasks. `Primitive` provides a vehicle for the raw processing code that will ultimately carry out the work for programs created using the framework. `Meta` provides an 'encapsulation' abstraction that allows groups of interconnected `Operator`s to be treated as though they were a single `Operator`.

The `Operator` class' `connect()` and `disconnect()` methods provide the basis for the framework's Developer API. These methods allow the developer to specify the interconnections between the various operations in a program and thus the nature of the processing to carry out. Both `Primitive` and `Meta` provide methods for constructing new operations based on their respective classes. This Operation API will be expanded on when the framework's important `InputManager` and `OutputManager` classes are described further in section 3.4 on data passing.

3.3 Data Types

This section describes how the framework represents data types generically, and at the same time allows data types with arbitrary structure, storage requirements and access schemes to be defined and used. Some common types that are frequently used in image processing programs are then identified and discussed.

3.3.1 Data

In principle, the data flowing between operations may be completely arbitrary in both meaning and structure. Hence, the framework itself can make few assumptions about what is actually being passed. Nevertheless, some scheme is required for representing the various types of data that will be used by programs built using the framework. The basis of this scheme is a `Data` superclass that represents a generic data type, but does not define any attributes for storing data or referencing allocated memory; rather, this is left to subclasses of `Data`, which in turn allocate whatever storage they require (see figure 3.13).

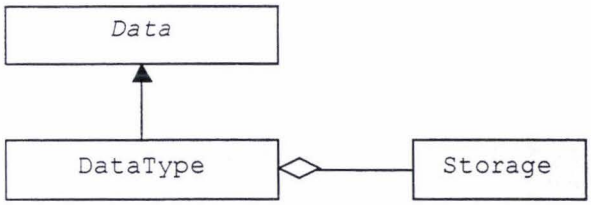

Fig 3.13: `Data`

This diagram represents the general scheme that is used by the framework to define new data types. Subclasses of `Data` are in turn attributed with another class whose responsibility is to actually store the data that is being typed. This important feature of the framework's design is fundamental to the operation of its data passing mechanisms. The reasons for this separation between attributes and actual storage will be discussed in section 3.4 where these mechanisms are explained.

Within this general scheme, the `Data` class represents a generic data type, and defines a Data API to which all subclasses of `Data` must conform (see figure 3.14). The role of a particular subclass is then to define any attributes that are associated with its data type (for instance a histogram class might declare an

integer to specify the number of bins it has) and to declare a Data Access API that provides access to the actual data. A separate class that is appropriate for the data being typed must be used to store this data. Examples of how this scheme is put into practice are given in the following subsection where some data types common to image processing are described. The four abstract methods the `Data` class declares constitute the framework's Data API for defining new data types.

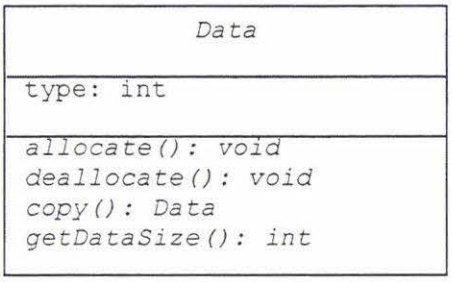

Fig 3.14: *The* `Data` *class*

The `allocate()` and `deallocate()` methods provide hooks that allow the framework to tell a particular `Data` object when it should allocate or deallocate the storage space it requires for its data. As such, they provide an opportunity for the framework to manage the allocation of memory resources that a program uses. Currently, the framework implements only the simplest of allocation/deallocation schemes (as discussed in the following section). However, the fact that they are part of the framework's Data API means that there is scope for a more advanced approach, which could be implemented without affecting the way that existing data types function within the framework. Also discussed in the next section are `Data`'s `copy()` and `getDataSize()` methods, both of which facilitate the framework's data passing mechanisms.

The `Data` class also declares one attribute: its `type`, which is represented as an integer. The framework uses this value to distinguish between different types, and for this purpose it assumes that all the different data types in use have been enumerated with a unique integer value. This is useful because it simplifies many situations where type checking is required, for example ensuring I/O compatibility or performing automatic type conversions.

3.3.2 Common Types

The number of data types common to the majority of image processing tasks is not large, and under normal circumstances it is unlikely that a new type would ever be required beyond this number. Each data type may be thought of as a protocol that allows operations to communicate with one another. A new data type would imply a new kind of problem or task that needed solving, and hence the need for new operations to utilise that data type. Figure 3.15 below demonstrates a class hierarchy of data types, which are all subclassed from `Data`.

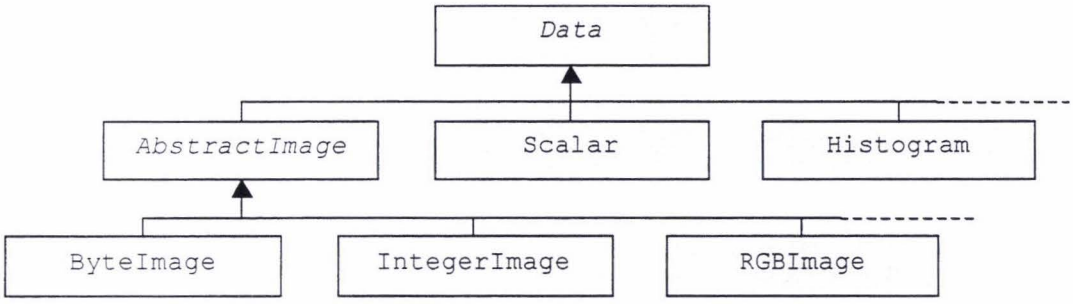

***Fig 3.15:** An example class hierarchy for* `Data`

In fact, this hierarchy is based on what was actually used for testing purposes during the framework's development. The dashed lines at the right of the figure indicate that this class hierarchy would in practice be extended to include other types.

Classes for data types such as `Scalar` and `Histogram` are directly subclassed from Data. The `AbstractImage` class, however, forms an intermediate stage in defining a range of different image types, of which the `ByteImage` class is one example. The following subsections discuss how data types such as these may be created for use with the framework, given what has already been covered concerning their generic properties. These examples are intended to assist in understanding the framework's Data API design.

3.3.2.1 Scalar

`Scalar` is a basic type that represents a single number that may take on any integer or real value. This type may appear, for example, at the output of a slider control operation. Initially, it was thought that separate data types would be used for both *integer* and *real* values. However, the `Scalar` class effectively combines these two so that the value stored by any particular `Scalar` may be interpreted in

either way, depending on the programmer's requirements. The advantage of this approach is that neither the programmer nor the framework need ever perform any type conversions between integers and reals, as this is handled implicitly by `Scalar` itself. Figure 3.16 shows how the `Scalar` class may be defined using UML.

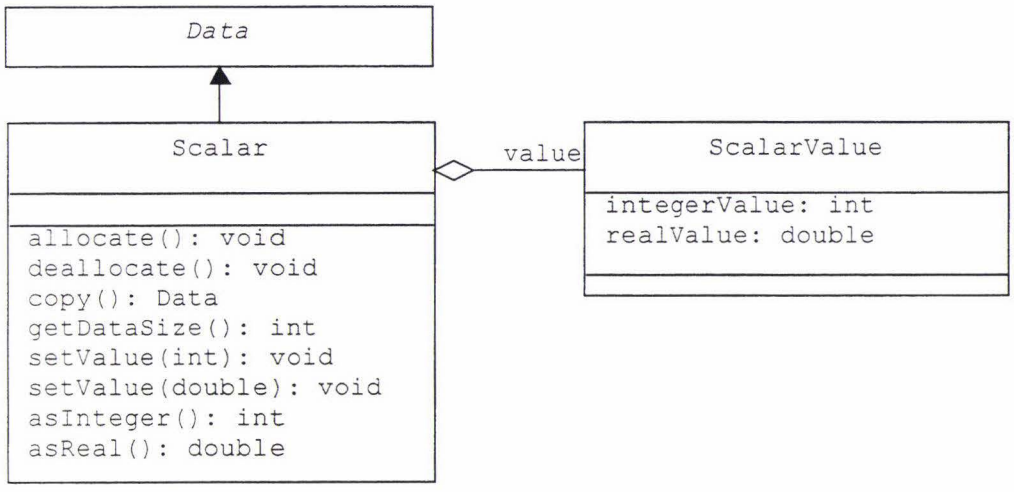

Fig 3.16: *The `Scalar` class*

`Scalar` implements the four abstract methods specified by its `Data` superclass, and adds four of its own; and, as required by the framework, it is attributed with a class (named `ScalarValue`) whose purpose is to store a `Scalar`'s actual data (which is just a single value). These properties of `Scalar` are now discussed.

The `ScalarValue` class stores its value in both integer and real form simultaneously by declaring both an `int` and a `double` for this purpose (these are Java's primitive types for integers and reals respectively). `Scalar` then defines methods to *get* and *set* this value; these are: `setValue(int)`, which sets the value as an integer; `setValue(double)`, which sets the value as a real; `asInteger()`, which returns its value as an integer; and `asReal()`, which returns its value as a real. Note that the *setter* methods change both integer and real internal representations simultaneously. Hence any conversion is done only once, at the time the value set. These four methods constitute `Scalar`'s own Data Access API.

The `Scalar` class always references an instance of `ScalarValue` (its data store) so it's `allocate()` and `deallocate()` methods need not do anything (`Scalar`'s storage is in effect permanently allocated). This is not an issue for this data type

since the space required is only a few bytes. However, these methods are put to proper use when larger data structures become necessary, for example in the case of images. As before, discussion of the `copy()` and `getDataSize()` methods is delayed.

3.3.2.2 ByteImage

`ByteImage` is an image class whose pixels are represented using bytes. This data type would be suitable, for instance, for processing greyscale images (where 256 distinct grey levels may be used). Of course, this class can be expected to have certain properties in common with other image types, for example it would need to have both a width and a height, and some means of setting these values. For this reason, an `AbstractImage` class is used to encapsulate all that is shared between different image types. Subclasses of `AbstractImage` (of which `ByteImage` is one example, as demonstrated in figure 3.17 below) customise this class further depending on the actual storage requirements of their particular image type.

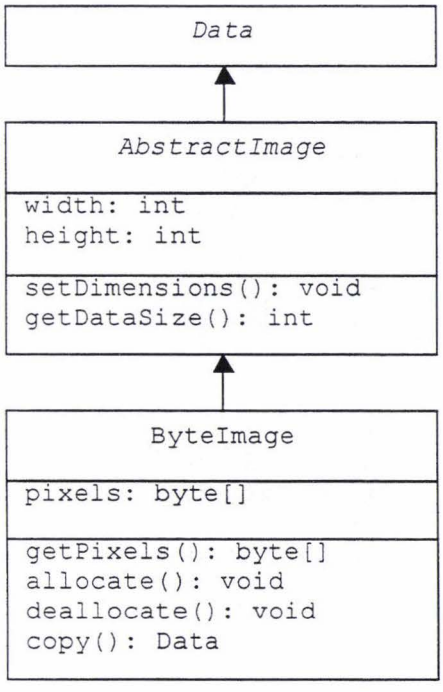

***Fig 3.17:** The `AbstractImage` and `ByteImage` classes*

The `ByteImage` class relies on an array of bytes to store its `pixels`. In Java, an array reference (of type `byte[]`, in this case) is equivalent to an object reference (i.e. arrays are treated as objects in themselves). In this way, `ByteImage` fulfils its requirement to aggregate a class for the purpose of storing its data. This class'

Data Access API consists of a single method: `getPixels()`, which returns a reference to its `pixels` array. The programmer may then manipulate the image data by reading and writing to and from this array.

This is a minimalist Data Access API in that the role of the `ByteImage` class is reduced to merely 'ferrying' the data between operations. Once an instance of `ByteImage` has made the journey from the output of one operation to the input of the next, its `pixels` array may be used directly without any further need to reference the original `ByteImage` instance. This approach is simple and leads to a minimum of overhead for accessing individual pixels (this is because Java has no equivalent to C/C++ style pointers, and therefore array elements must be accessed via their indices).

It is not the only approach however, and an alternative would be to provide method access to an image's pixel data (for instance via *getPixel()* and *setPixel()* methods). This would provide a better encapsulation of the concept of an image as a two-dimensional array of pixels. However, there are efficiency issues that must be taken account of when considering alternatives such as these. During this project, a number of these alternatives were investigated to find the most optimal. Also considered were the trade-offs between convenience of use and memory requirements with respect to alternative pixel representations. Such alternatives arise from Java's lack of an unsigned byte primitive type. The results of these investigations and the conclusions that were drawn are included in appendix 1.

`ByteImage`'s `allocate()` and `deallocate()` methods allocate and deallocate the pixel array referenced by its `pixels` attribute. Allocation simply involves calculating the size of the array to allocate, and declaring a new array of that size. Since `pixels` is defined as a one dimensional array, the required size is equal to the number of pixels in the image (width multiplied by height). Deallocation is even simpler; because of Java's garbage-collected mechanism, the array may be deallocated by assigning a null reference to `pixels`.

3.3.3 Summary

The abstract `Data` class is the superclass for all data types that may be implemented within the framework. Its principle role is to allow the framework to pass data between operations in a generic manner, independent of what form that

data actually takes. Because a given data type may be arbitrary in both meaning and structure, responsibility for storing that data is deferred to concrete subclasses of `Data`. These subclasses must conform to a Data API, which allows the framework to handle different data types in a uniform way. An important requirement of the framework is that any particular `Data` subclass must store its data via a class that is appropriate for the role. This requirement facilitates the framework's data passing mechanisms.

There are a number of common data types found in image processing. These form a class hierarchy, with the `Data` class at its root; `Scalar` and `Histogram` are two examples of these. There are also several different image types, for example `ByteImage` and `RGBImage`. These have their own superclass, `AbstractImage`, which encapsulates the properties that are shared by all images.

3.4 Data Passing

This section describes a very significant aspect of the framework's design: its data passing mechanisms. These mechanisms embody the main function of a data flow system: to allow data to flow between interconnected operations. Data passing also deals with inputs that require default values, automatic type conversions and other issues, all of which are discussed below.

3.4.1 Input and Output

The basis of the framework's data passing scheme is the *Observer* design pattern (Gamma, *et al.*, 1995). This is a way to implement a one-to-many relationship between classes that require knowledge about each other. Note that data does not actually flow between operations as such, but between the inputs and outputs of those operations. This is an important distinction because any operation may have multiple inputs and outputs, each of which represents one half of an independent 'conduit' through which data may flow. An operation itself represents the point where (possibly multiple) inputs are brought together and combined, the results of which appear at the operation's (possibly multiple) outputs. The framework's `Input` and `Output` classes take responsibility for connecting operations together (see figure 3.18).

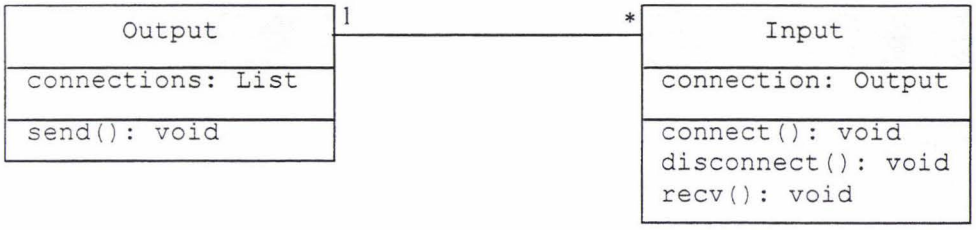

Fig 3.18: *The* `Input` *and* `Output` *classes*

This diagram shows that the cardinality of the relationship is one-to-many in that every `Output` may be connected to many `Input`s, while every `Input` is connected to just one `Output`. This relationship is implemented by `Input` via its `connection` attribute (of type `Output`), and by `Output` via its `connections` attribute (of type `List`). Each `Output` has a dynamically re-sizeable list of `Input`s.

`Input`'s `connect()` and `disconnect()` methods serve to make or break connections between `Input`/`Output` pairs. A call to `connect()` causes a given `Input` to begin *observing* a given `Output`. Conversely, a call to `disconnect()` causes an `Input` to cease observing its `Output`. Connections are always initiated or terminated from the 'input end'. This design feature simplifies the task of ensuring connections are made and broken cleanly (by preserving the semantics of the connection). For instance, the framework must ensure that an `Input` never references an `Output` that does not in turn reference it back. Such a situation would represent some kind of 'half-connection', which would invariably lead to confusion and undefined behaviour in programs created using the framework.

3.4.2 Progress

Once a connection between an `Input` and an `Output` has been established, the `send()` and `recv()` methods (see figure 3.18 above) take responsibility for communicating between the two. This pair of methods represents a variation on the *Observer* pattern that allows a *granular* approach to the passing of data between operations. This aspect of the framework's design is intended to provide the *responsiveness* that was identified as being important in the previous chapter. In order to understand this approach, consider two extreme cases in the example of an image being passed from one operation to another:

a) The image is passed in its entirety. In this case, all of the image must actually be available (processing must have been completed) *before* it can be

passed on. If the process that is generating the image is slow, then it may be some time before this can happen.

b) The image is passed pixel-by-pixel. That is, as soon as a new pixel is generated, it is made available to whatever processes are interested in the data. This is equivalent to passing the image in its most granular form.

Neither of these cases is ideal. The former limits the responsiveness of the system; using this approach, 'downstream' operations will always have to wait for 'upstream' operations to complete their processing before they may begin any of their own. In the latter case, the communication overhead would slow the actual data processing down to such a degree that it would outweigh the intended gains in responsiveness.

The framework addresses these issues by adopting an intermediate approach. For this purpose it introduces the concept of a connection's *progress*. The progress of any connection is an integer value that represents the number of *units of data* that are currently available from that connection. Of course, because the data flowing between operations may be of any type, these units are arbitrary and are part of the definition of the particular data type. This means that for a given type of data, some sensible meaning must be attached to exactly what 'a unit' represents. This implies that operations must interpret this meaning in the same way for a particular data type, which in turn reinforces the concept of a data type as being a protocol that allows operations to communicate between themselves in meaningful fashion.

As an example of this, a convenient unit of data for an image type is a single row of pixels. In this way, a new unit would become available after every row was processed. This would allow an image to be displayed row-by-row, for instance. A user could then decide early on whether or not to adjust a particular input, because the feedback would be *progressive*, and thus more responsive. Other data types must also attach some meaning to the units that their respective data is broken into. The `Scalar` type, for instance, might have only two levels of availability: 'not available' and 'available', depending on whether or not its value had been processed at any given time.

The role of the `send()` and `recv()` methods is to pass the current progress level associated with a given connection. When the progress of a connection advances (due to new data becoming available) the `send()` method of the `Output` of that connection is called, which in turn calls the `recv()` method of each of the `Inputs` that are currently observing it. Each `Input` then has the opportunity to notify the operation it belongs to that some new data is available, and that there may be some further processing to carry out. An operation that processes this data may then produce newly available data at its own outputs. This causes calls to `send()` and `recv()` to propagate throughout the operations in a program.

By assigning special meanings to certain progress levels, this mechanism provides a solution for controlling the various *states* that a connection may be in (see table 3.1). Subsequent subsections will discuss how these states are used to facilitate the actual passing of data between operations.

Connection State	Progress Level
DISCONNECTED	-2
INVALID	-1
VALID	0
AVAILABLE	1+
COMPLETE	∞

Table 3.1: *Progress Levels*

The table shows that a progress level greater than or equal to one means there is data *available* for processing. In this range, the progress level is equal to the number of units that are currently available and that there is more data to come. When processing is finally *complete*, the progress level becomes infinite (this is in conceptual terms only; in practice, the greatest integer value possible is assigned).

A connection may also be in the *invalid* or *valid* states. Invalid indicates not only that there is no data present, but that no data type has yet been associated with the connection. In this state, nothing can flow along this connection. A valid connection means that there is data, but that none is available (i.e. the number of available units is zero). This state may be considered a special case of the *available* state.

Note that in the table above there is also a *disconnected* state; an `Input` that receives this progress level will immediately disconnect itself from the originating

`Output`. This allows an `Output` to request that all of its corresponding inputs should be disconnected via a call to each `Input`'s `disconnect()` method.

The framework's *progress* mechanism effectively allows information on the *intermediate* progress of data to be passed between operations, thereby enabling them to begin processing that data before the whole data has become completely available. There are limitations to the mechanism, however, which are equally important to understand.

The progress mechanism as described here is effective only when the data is produced in units that map to a monotonically increasing value, since this is how the framework's progress level has been modelled. Take for example the representation of an image with each data unit as a single row in the image. This representation works well with operations that generate images in raster fashion (which is common). However, this approach is not as effective with operations that generate their pixels in a random order.

In this case, the framework's progress mechanism would be forced to delay the availability of the output image data until the whole input image had been processed. While this situation is still handled by the framework, the data can no longer be passed in individual units, and the advantages that this mechanism affords the framework are effectively lost.

Within the framework, it is possible to define a new image type that more intelligently handles the random access processing of images using the framework's progress mechanism. Consider, for instance, that each successive progress level of the new image type represents a new *version* of the entire image as it is processed. This would allow for entire updates to be passed; using this scheme, progress would become *complete* once the final update had been processed.

This demonstrates the arbitrary nature of the meaning that can be attached to a connection's progress level. What is important to realise is that by adopting a new interpretation of this level, a new data type is effectively created. The implication of considering the particular definition of a data unit as being part of a data type is that operations utilising that data type must interpret the arrival of new data

according to how its data units have been defined. Otherwise the meaning of each new unit is lost and unexpected results are certain to occur.

3.4.3 InputManager and OutputManager

Given that the framework provides a mechanism for communicating *the amount* of data that is currently available through a connection, the next step is to show how the data itself is actually passed from an output to a corresponding input. The framework's Data API provides a number of methods that give the programmer access to a `Primitive`'s inputs and outputs. These methods are provided by the framework's `InputManager` and `OutputManager` classes (mentioned earlier) which subclass `Input` and `Output` respectively (see figure 3.19).

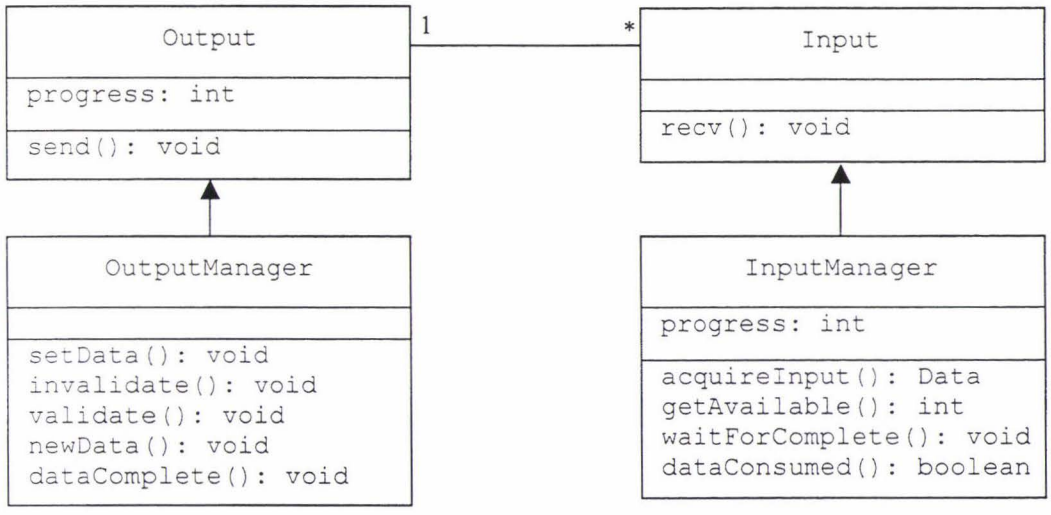

Fig 3.19: *The `InputManager` and `OutputManager` classes*

These methods are used by the programmer within a `Primitive`'s `implementation()` method to control the passing of data at the granularity provided by the framework's *progress* mechanism. An important premise underlying this part of the framework's design is that *the flow of data should be as transparent to the programmer as possible.* This means that the programmer should be able to implement a `Primitive`'s processing code *as if* all of the required input data was available, and place the processed data at its outputs *as soon as* and *as fast as* desired. In reality, various operations will produce and consume data at very different rates, meaning that a `Primitive` will often have to stop and wait for more input data when it is part way through its processing. The

framework's Data API is designed to allow the programmer to remain unaware of when or if such delays occur.

Figure 3.19 also shows that each 'end' of a connection is attributed with its own `progress` level. `Output`'s `progress` attribute tracks the availability of the data as it is produced. At a corresponding input, `InputManager`'s `progress` attribute tracks the amount of data that has been processed so far. In this way, these progress levels are able to advance independently to reflect the rates at which data is produced and then independently consumed. Figure 3.20 reveals the remaining design aspects relied upon by the framework's data passing mechanisms.

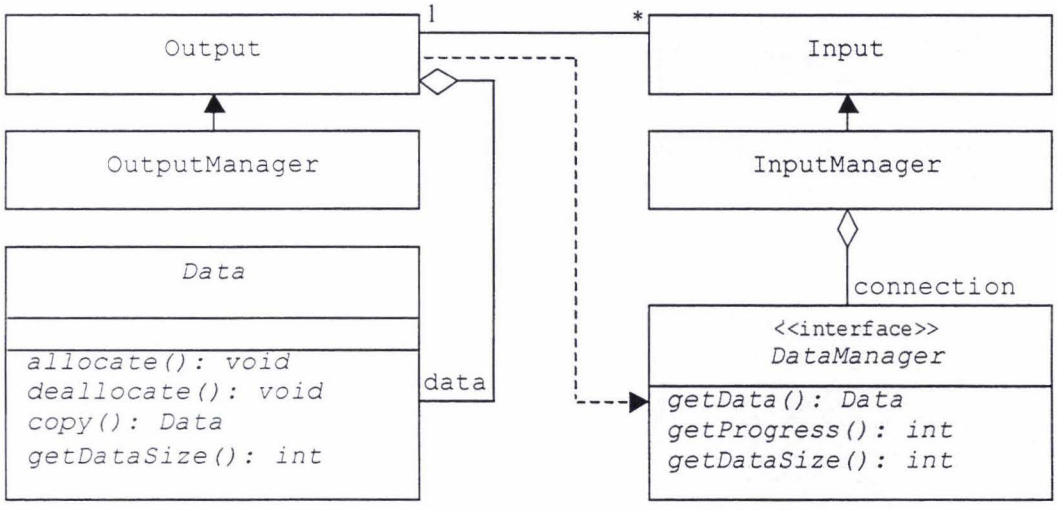

Fig 3.20: *Behind the scenes of the framework's data passing mechanisms*

The `DataManager` interface, which is realised by `Output`, provides the mechanism by which an `InputManager` is able to give access to an `Output`'s `data`. The methods provided by `InputManager` carry out their functions by calling on the `DataManager`. The `DataManager` interface also facilitates the framework's *default data* and *automatic type conversion* features, as will be shown in a later section.

The `Output` class aggregates an instance of `Data`. This is a reference to the actual `data` that will be produced at that `Output`. `OutputManager`'s `setData()` method associates a given instance of `Data` with an `Output`; from then on, this becomes the `data` that is 'managed'. By retaining a reference to this `Data` instance, the programmer may then use the Data Access API of that instance to modify its

contents, with these modifications being accessible by downstream operations via the framework's progress and data passing mechanisms.

The remaining Data API methods are now discussed. Typically, a `Primitive`'s `implementation()` method will follow a pattern similar to that illustrated in the flowchart in figure 3.21. The following subsections discuss the various `InputManager` and `OutputManager` methods that may be used at each step outlined in the flowchart.

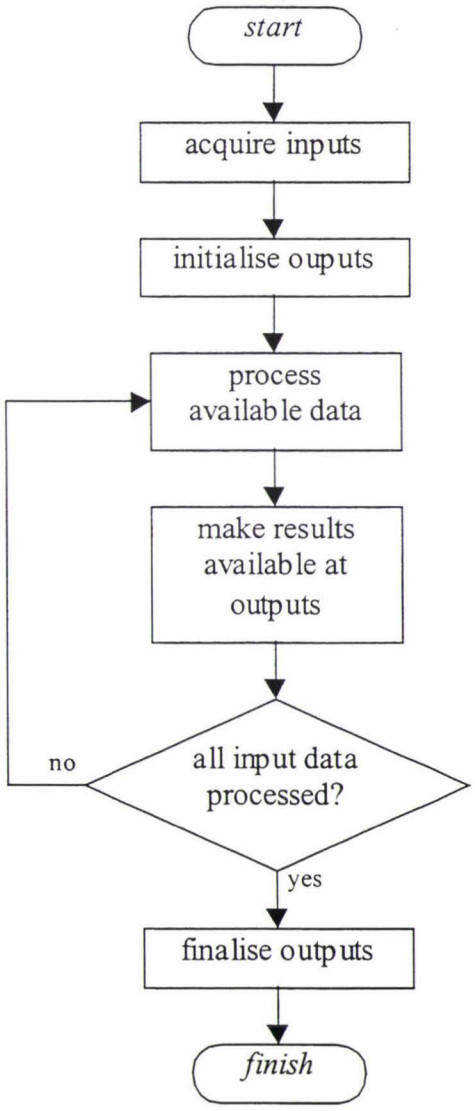

Fig 3.21: *Pattern implemented by a typical* `Primitive`

3.4.3.1 Acquiring Inputs

An `Output`'s `data` attribute references the actual data that is being produced at a given output. `InputManager`'s `acquireInput()` method returns with an instance of `Data` that may be used to access this. Hence this is where the data itself is physically passed from one operation to another. As such, this step may be thought of as *activating* a connection. Once the instance that is acquired has been cast to the appropriate `Data` subclass, its Data Access API may be used to access its contents.

The sequence diagram in figure 3.22 below reveals the sequence of method calls that occur when a `Primitive` successfully attempts to acquire the data at one of its inputs. When these methods unwind, they pass back the instance of `Data` returned by `copy()`, which is finally returned to the programmer by the original call to `acquireInput()`. The programmer may then use this instance to access the contents of the data at the input just acquired. Note that `acquireInput()` does not return until it is able to, which is only when the data becomes *valid*. In the following section on scheduling, it will be revealed how the framework avoids simply 'busy waiting' while the data is waited upon.

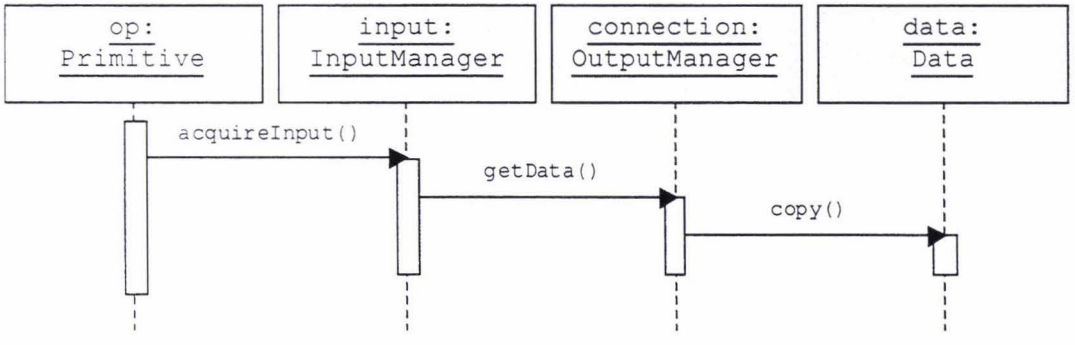

***Fig 3.22:** Acquiring an input*

The key to this mechanism is the `copy()` method, which must be implemented by all subclasses of `Data`. The role of this method is to perform a shallow copy of the instance of `Data` on which it is called. Figures 3.23a and 3.23b illustrate how a call by `op2` to `acquireInput()` provides it with a reference to its required input data. Note that to be successful, there must already be a connection established between the output of `op1` and the input of `op2`, and the connection must be in the VALID state. However, the data itself does not yet need to be available. The `Primitive op2` then has a reference to the `data` that is local to its `implementation()` method (this is indicated in the diagram by the `<<local>>` stereotype[6]). For the duration of this method, the programmer may use this reference to access the `data` as it becomes available from the upstream

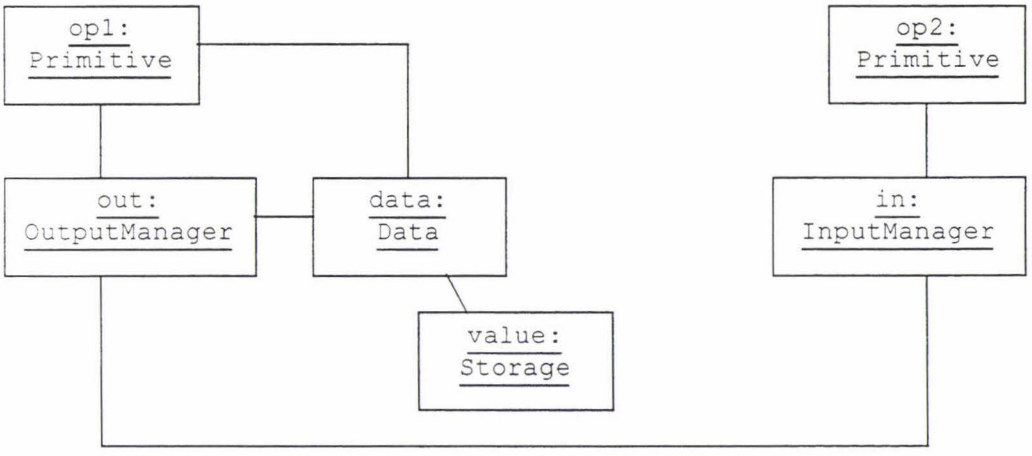

Fig 3.23a: *Before a call to* `acquireInput()`

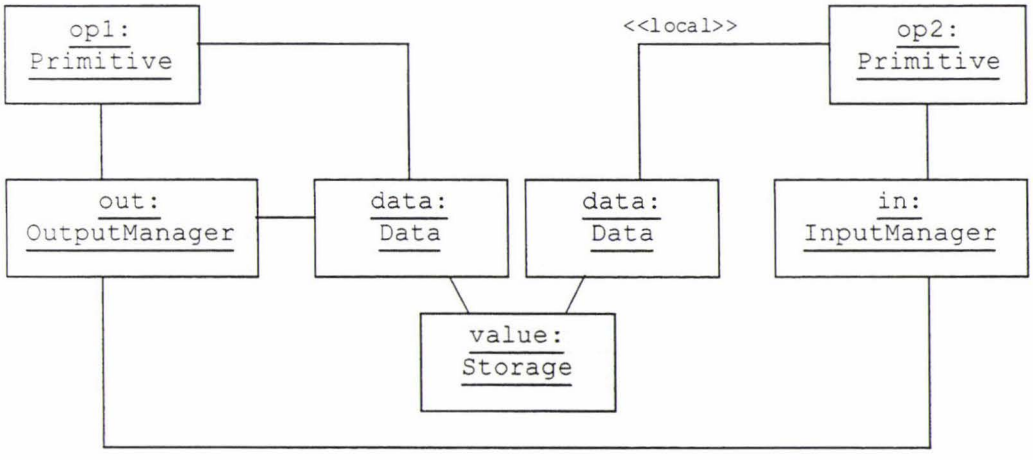

Fig 3.23b: *After a call to* `acquireInput()`

[6] In UML, *stereotypes* may be used wherever additional notation is required to convey some special, problem specific information about a model.

`Primitive`. It is important that this copy is shallow, that is its storage class must have been copied by reference, because this ensures that both ends of the connection have access to the same memory that has been allocated for the `data`.

Because the `Storage` object is shared rather than the `Data` object, the `Primitives` at each end of a connection remain independent. That is, if the `ByteImage` in the above example suddenly needed to be recalculated at different dimensions, then not only would the original instance have to adjust its `width` and `height` attributes accordingly, but its pixel array would need to be reallocated to reflect these new dimensions. This may happen, for instance, because the `Primitive` producing the image finds that its own inputs have been recalculated. Because such a recalculation may occur at any time, each end of the connection must reference its own copy of the `data` instance. Otherwise such changes may result in a `Primitive` that has recently acquired `data` to experience indexing errors.. In this example, such a `Primitive` would have its input invalidated (as described in the next section on scheduling) and would restart its own processing as a result (by re-invoking its `implementation()` method). Since the copied `Data` instance is local to the `implementation()` method, it is discarded when this method gets re-invoked. In Java, this memory would then become eligible for garbage-collection.

The following Java code fragment illustrates the process of acquiring a `Primitive`'s inputs in preparation for carrying out some processing on their data. This `Primitive` has two inputs, one that provides a `ByteImage` and the other that provides a `Scalar`.

```
// acquire inputs and cast to their appropriate Data subclasses
// IMAGE_IN is the InputManager
// image_in is the Data instance that has been aquired via
//  IMAGE_IN
ByteImage image_in = (ByteImage)IMAGE_IN.acquireInput();
Scalar value_in = (Scalar)VALUE_IN.acquireInput();

// get image's dimensions and its pixel array
int width = image_in.getWidth();
int height = image_in.getHeight();
byte[] pixels_in = image_in.getPixels();
```

The first line of code here declares a local reference named `image_in` of type `ByteImage` and assigns it the instance of `Data` that is returned by the call to `acquireInput()`. The cast operation is necessary to convert this `Data` into an

instance of type `ByteImage`. The object named `IMAGE_IN`, on which `acquireInput()` is called, is the instance of `InputManager` previously created by a call to `Primitive`'s `addInput()` method (when the `Primitive` is constructed). The same is done for `value_in` using the `VALUE_IN` input.

Upon successful completion of the call to `acquireInput()`, the `image_in` reference can be used to learn the dimensions of the image, and to gain access to its `pixels` attribute, and thus the image data itself. The `Scalar` type `value_in` has no attributes; once its has been acquired its Data Access API may be used to access its contents. Subsequent subsections will cover how the Data API methods regulate this access so that the expected results are in fact the ones achieved.

3.4.3.2 Initialising Outputs

An input cannot be acquired until the output that it is observing has data that is *valid*. Initialising a `Primitive`'s outputs involve invalidating and then re-validating them. This causes the necessary messages to be sent from a given output to any observing inputs, which in turn notify the operations they belong to that a recalculation is about to take place, and that they may need to reprocess their own input data. Figure 3.24 illustrates the sequence of calls that typically occurs.

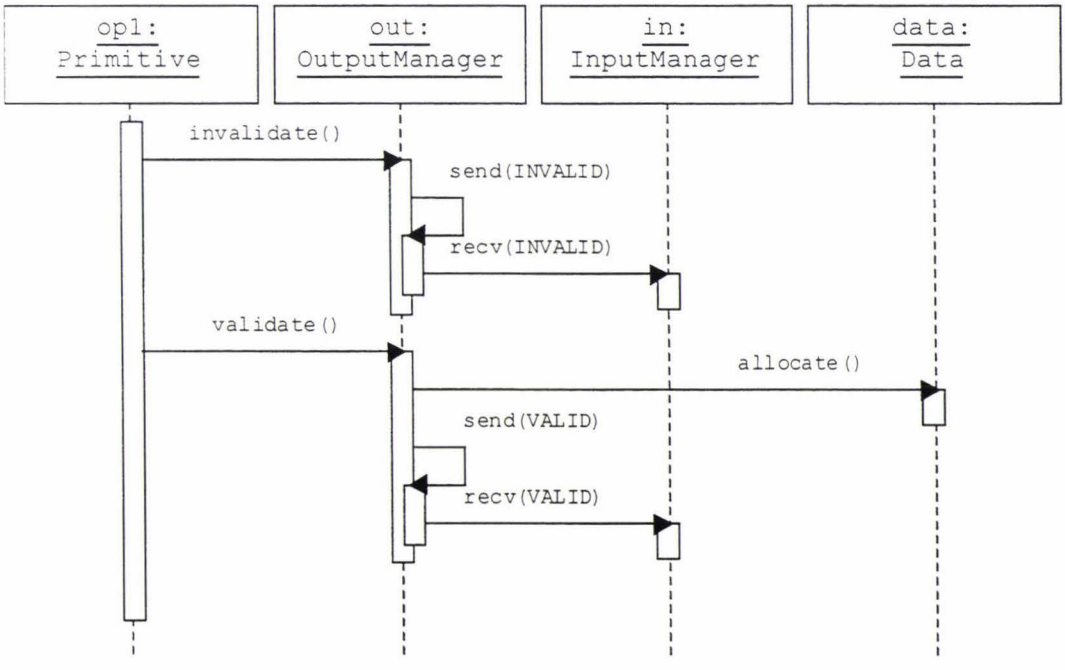

Fig 3.24: *Initialising an output*

Though not shown explicitly here, the `send()` method calls `recv()` *for each* of the `Inputs` its `Output` is connected to. This effectively 'broadcasts' from an output to all of its interested inputs. An `OutputManager` uses this broadcast mechanism to tell each of its inputs (via the progress levels) that their connection is INVALID, and then VALID, via calls to `invalidate()` and `validate()` respectively. The `validate()` method also invokes the `allocate()` method on the `data` it is managing; this ensures that its `data` storage is allocated and in a valid state, so that it may then be acquired at a corresponding input as described above.

The programmer is thus provided with a simple idiom with which to initialise an output. Three steps are involved: 1) invalidate the output; 2) configure the output data's attributes; and then 3) re-validate the output. Consider the following example, where an image is to be output.

```
// determine dimensions and initialise the output image
// IMAGE_OUT is the OutputManager
// image_out is the Data instance that has been associated
//  with IMAGE_OUT

IMAGE_OUT.invalidate();
image_out.setDimensions(width, height);
IMAGE_OUT.validate();
```

This example presumes that the dimensions of the output image (referenced by `image_out`) have been determined from the `Primitive`'s inputs in some way. For instance, its size could be equal to the size of an input image, which would be the case for a thresholding operation. From here, `IMAGE_OUT` is invalidated, `image_out`'s dimensions are set, and then `IMAGE_OUT` is re-validated.

This idiom ensures that the framework's data passing mechanisms will reset the corresponding inputs of any downstream `Primitives`. Once `validate()` has been called for a particular output, the attributes for the `Data` instance associated with that output should not be changed without first calling `invalidate()`. The `validate()` method lets an `OutputManager` know that its `data` is now available to be acquired.

3.4.3.3 Processing the Available Data

The successful acquisition of an input implies that its connection is in a *valid* state. This means that the data at this input has had its attributes configured, and

that storage has been allocated for its contents. It does not necessarily mean, however, that there is any actual data available to process. The framework's progress mechanism must be used to determine how much data is actually available at any time. For this the programmer uses `InputManager`'s `getAvailable()` and `waitForComplete()` methods. In a previous example, it was shown how a call to `acquireInput()` provides the programmer with an instance of `ByteImage`, which is then used to access that image's pixels. The roles of `getAvailable()` and `waitForComplete()` are to regulate this access so that units of data are not inadvertently processed before they are produced.

The `getAvailable()` method has two variants. The first is parameter-less; it returns with an integer equal to the number of units of data currently available at an input. This variant is suitable when the programmer simply wishes to process as much data as possible. Repeated calls to this variant of `getAvailable()` only return when there is *more* data available than when the previous call was made. The second takes a single integer parameter which specifies the exact unit that is required. This variant returns when at least that unit is available; in this way it provides better control over the handling of the available data. Either or both of these variants may be used, depending on how the programmer wishes to implement a particular processing algorithm. The second variant still returns the number of units currently available. The difference between the two is *when* they

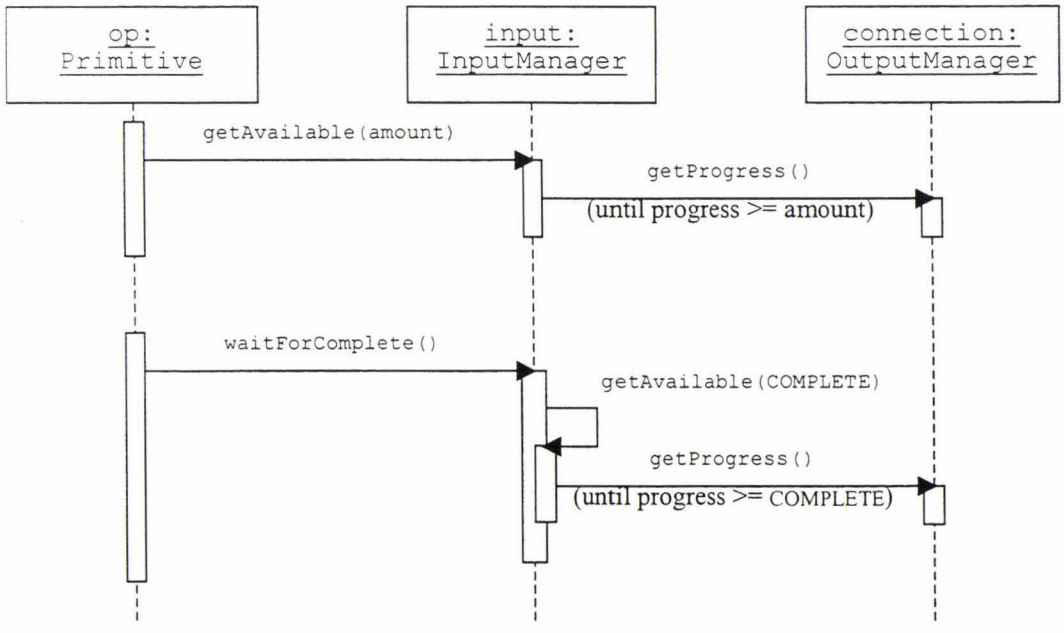

Fig 3.25: *Getting the available data*

return. In fact, if no parameter is provided to this method, it uses the value from `InputManager`'s `progress` attribute, which is updated on every call to `getAvailable()` to reflect the latest information regarding the number of units available at the corresponding output. The sequence diagram in figure 3.25 above provides a typical scenario resulting from a call to either version of `getAvailable()`. The `waitForComplete()` method does not return until all the data at an input has become available.

3.4.3.4 Making the Results Available

As a `Primitive` processes its input data, its outputs should be notified as the results become available so that the progress mechanism works effectively. `OutputManager` provides a single method, `newData()`, for this purpose. This method takes a single integer parameter that specifies the number of new units that have been produced *since the last call to* `newData()`. For instance, a `Primitive` producing an image with a row-by-row data structure would call `newdata(1)` after each row was completed. Alternatively, `newData(10)` could be used after every ten rows; this would reduce the overhead associated with each call to `newData()`, but would also reduce the granularity at which the data was accessible at its output.

This method uses the same broadcast mechanism used by the `invalidate()` and `validate()` methods discussed above with the difference that the `OutputManager`'s `progress` level is incremented by the specified value. The following code fragment shows how the `getAvailable()` and `newData()` methods can be employed. It is assumed that the necessary inputs have been acquired and that all outputs have been initialised using code similar to the previous two examples.

```
// acquire inputs (section 3.4.3.1)
...

// initialise outputs (section 3.4.3.2)
...

// process data
// We require VALUE_IN before we can start any processing
VALUE_IN.waitForComplete();
byte value = (byte)value_in.asInteger();

byte[] pixels_out = image_out.getPixels();
for (int y=0, index=0; y < height; y++) {
  // Wait until row y is available at the input
```

```
    IMAGE_IN.getAvailable(y);
    for (int x=0; x < width; x++, index++)
      // process a pixel
      pixels_out[index] = pixels_in[index] + value;

    // make a row available
    IMAGE_OUT.newData(1);
  }
```

In the context of this operation, the `Primitive`'s `Scalar` input is required before any further processing can take place. The `waitForComplete()` method is called for this input's `InputManager` and when this method returns, the `Scalar`'s value can be accessed.

To process the image, a double for-loop is used to iterate through each pixel in `pixels_in`; each pixel has its value incremented by `value`, and is then placed in its corresponding position in the output image. At the start of each row, a call is made to `getAvailable()` to ensure that the current row is available at the input. Accessing pixels that have not yet been made available may lead to unpredictable results, and should be avoided. At the end of each row, a call is made to `newData()`, which notifies any inputs observing this output that a new row is now complete at the output of this `Primitive`.

3.4.3.5 Checking for More Input Data

Note that in the code example given in the previous subsection, processing finishes once the double for-loop completes. Significantly, no explicit checks are necessary to determine if there is any more input data available. Rather, this is all taken care of by the `waitForComplete()` method (in the case of the `Scalar`) and the `getAvailable()` method (in the case of the `ByteImage`). In particular, this code assumes there is no more image data available once the number of rows that have been processed is equal to the height of the input image (if this assumption doesn't hold then something has gone very wrong!).

No checks are needed in this example because the nature of the input data made such checks unnecessary. It is possible to know when all the data has become available purely from the number of units that have been passed through. This is not the case for all data types, however. An example of such a type is a list type that grows dynamically as it is produced. If each element in such a list represents a single unit, then the number of units will not be known until the operation

producing that list has finished its processing. For situations such as these, `InputManager` provides a `dataConsumed()` method, which returns the boolean value `true` only when the connection is in the *complete* state (and thus the number of units is known), *and* the programmer has previously requested at least this many units (via calls to `getAvailable()`). This second condition ensures that `dataConsumed()` does not return `true` until every unit been requested.

The scenario in figure 3.26 is a variation on the one for `getAvailable()` illustrated in figure 3.25 previously. Note the additional call to `OutputManager`'s `getDataSize()` method, which in turn calls `getDataSize()` on the `data` being managed. Subclasses of `Data` should implement this method to return the number of units of data they contain. For a `ByteImage`, the height of the image should be returned as this is equal to the number of rows it has, and hence its unit count. For a dynamically sized data type, such as a list, the number of units currently available should be returned (in the case of a list this would be the number of elements it contains).

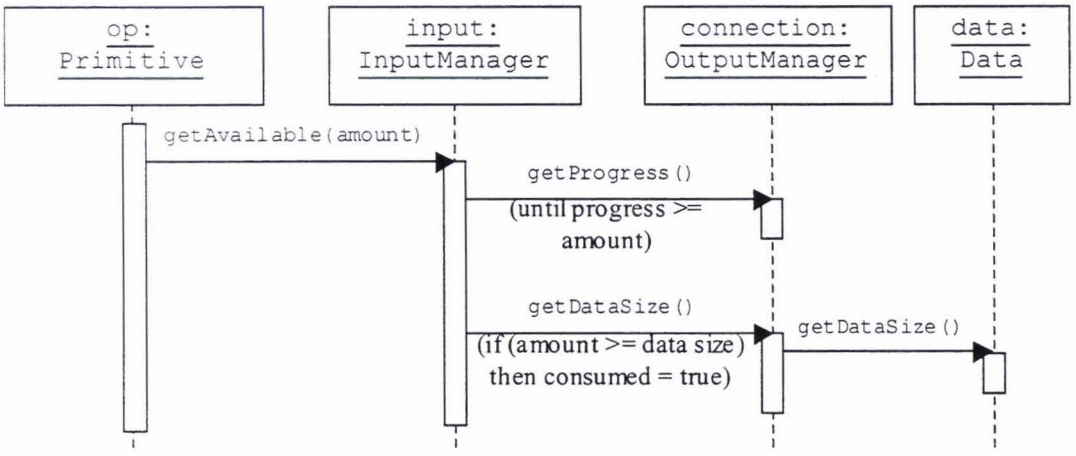

Fig 3.26: *Determining when all input data has been consumed*

The call to `getDataSize()` is invoked for the case where the previous call to `getProgress()` returns COMPLETE. Since the output has been found to be in the *complete* state, the value returned by `getDataSize()` can be relied upon to be the final value for the number of units that the data at this output contains. A comparison is then made between the number of units requested by the original call to `getAvailable()`. If this is greater than or equal to the data's size, then `InputManager` sets an internal flag to indicate that all of the data at this input has been consumed. The `dataConsumed()` method simply returns the value of this

flag. Thus it may be used to determine whether or not all of the data that will ever be produced at a given input has been requested via calls to `getAvailable()`.

Note that there is nothing to stop the `dataConsumed()` method being used to control the processing of data for image types or other types where it is not strictly required. The following example code fragment performs the same processing on the image as in the previous example. As before, it is assumed that the necessary initialisation steps have been completed.

```
// process image data as fast as it becomes available
int n=0, i=0;

while (!IMAGE_IN.dataConsumed()) {
  // request as many more rows as possible
  n = IMAGE_IN.getAvailable();

  // process (n-i) rows starting at row i
  for (index=i*width; index < n*width; index++) {
    pixels_out[index] = pixels_in[index] + value;
  }
  IMAGE_OUT.newData(n-i);
  i = n;
}
```

In this example, `dataConsumed()` is used in conjunction with the parameter-less variant of `getAvailable()`. On each iteration of the while-loop, `getAvailable()` returns with as many new rows as possible. Once all of the data has been requested, `dataConsumed()` returns `true` and the while-loop terminates. Note that this code makes its output image available in chunks, and the size of each chunk depends on how many rows of the input image are returned by each successive call to `getAvailable()`. This may be row-by-row or all at once, or it may be somewhere in between. The code could be modified slightly if a more regular scheme is desired (for instance so that `newData()` was called after each row was processed).

3.4.3.6 Finalising Outputs

Once a `Primitive` has completed producing the data for one of its outputs, it should call `OutputManager`'s `dataComplete()` method to notify all interested inputs that this data is now complete. The following code fragment gives an example of how this method is used.

```
// finalise output
IMAGE_OUT.dataComplete();
```

After a call to `dataComplete()`, the contents of the data at the corresponding output should not be modified any further. Note that to output a `Scalar` type, a call to `newData()` is never needed. The code snippet below shows how a `Scalar` output could be produced. In this example, `SCALAR_OUT` is the `OutputManager` managing the scalar and `value` is the instance of `Scalar` that has been associated with this output.

```
// output scalar value
SCALAR_OUT.invalidate();
SCALAR_OUT.validate();
value.setValue(100);
SCALAR_OUT.dataComplete();
```

In this case, the data is effectively complete as soon as its value has been set.

3.4.4 DataManager

The `DataManager` interface was introduced in the previous section. `DataManager`'s methods are called by instances of `InputManager` to access information about the data at the input. There are several important types of `DataManager` within the framework. The previous subsection described a connection between two `Primitive`s and in this case the `DataManager` was the `OutputManager` of the upstream `Primitive`. Two other classes that implement the `DataManager` interface are `DefaultDataManager` and `ConversionManager`. In each of these cases, the `InputManager` has no knowledge of the fact it is not receiving its data from an actual output, and neither does it need to. The advantage of using an interface is that it simplifies the `InputManager`'s implementation.

3.4.4.1 Default Input Data

Occasionally, a programmer will want to provide one or more of a `Primitive`'s inputs with *default values*. The presence of a default value at an input can allow a `Primitive`'s `implementation()` method to produce valid output data even when that input has no physical connection. For example, a threshold image operation could provide a default value for the input that supplies its threshold level. Default input values are generally only practical if their storage requirements are minimal. It would be unusual, for instance, to find an operation that had a default

image associated with one of its inputs. Figure 3.27 introduces the framework's `DefaultDataManager` class, which facilitates the provision of default values.

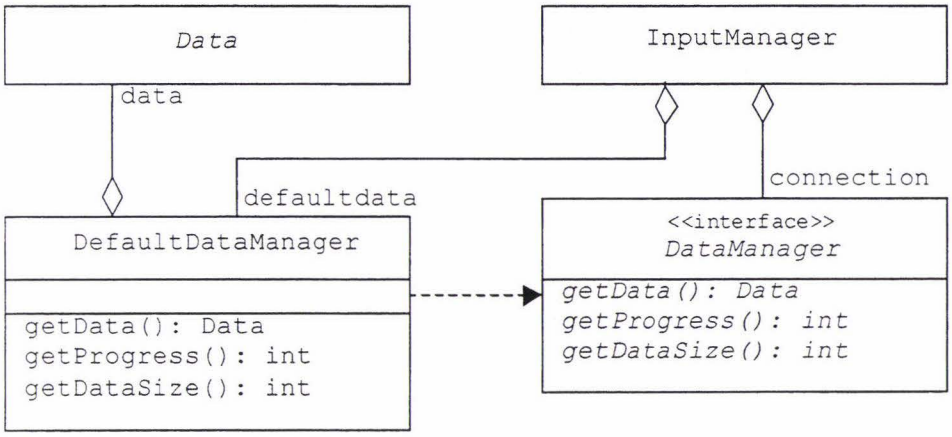

Fig 3.27: *The `DefaultDataManager` class*

In addition to it's `connection` attribute, `InputManager` also aggregates an instance of `DefaultDataManager`, which implements the `DataManager` interface. `DefaultDataManager` aggregates an instance of `Data`, which represents the default data that this `DataManager` will make available to its `InputManager`.

This design greatly simplifies the mechanism that is used for handling default values. It is driven by `InputManager`'s `acquireInput()` method. Previously, figure 3.22 demonstrated the sequence of calls normally triggered by this method. That scenario depended on `InputManager`'s `connection` attribute referencing an instance of `OutputManager`. However, if nothing has physically connected to this input, then `connection` will reference `null`. In this case, the `InputManager`'s `DefaultDataManager` 'fills in' for the missing `OutputManager`; this is achieved as shown here:

```
if (connection == null) connection = defaultdata;
```

The same sequence of calls is then made as was shown in figure 3.22. The difference is that the `DataManager` being used is now an instance of `DefaultDataManager` rather than `OutputManager`.

The `DefaultDataManager` implementation of the `DataManager` interface is trivial. The methods `getData()` and `getDataSize()` simply pass on the appropriate calls to the `DefaultDataManager`'s `data` attribute. The

`getProgress()` method, however, always returns COMPLETE (assuming its data attribute is non-null). The framework's implementation of this method is given below to illustrate this:

```
public int getProgress() {
  if (data == null)
    return INVALID;
  else
    return COMPLETE;
}
```

Hence an input's default data is either completely available (where this `DefaultDataManager` has been assigned some `data` to manage) or invalid (if no `data` has been assigned).

3.4.4.2 Automatic Type Conversions

Automatic type conversion is a convenience feature of the framework. This feature obviates the need to insert explicit 'conversion' operations into a program. The aim of this feature is to provide a completely transparent mechanism for automatically converting from a data type produced at one output to an alternative type expected at a corresponding input. Examples of typical conversions that developers might find useful are colour image to grey scale image, or image to histogram. Note that an automatic conversion should only be defined where some *sensible* conversion actually exists.

This mechanism operates on the same principle as the default data mechanism described in the previous section; that is, `InputManager`'s `acquireInput()` method determines that the `OutputManager` it is connected to is unsuitable and so replaces it. In this case, it is replaced with an instance of the framework's `ConversionManager` class. Figure 3.28 below illustrates the design structure relevant to this mechanism.

The `ConversionManager` class is the abstract superclass for all conversion managers. Concrete subclasses of `ConversionManager` take responsibility for converting *to* a given type. So, for example, the `ConvertToHistogram` class in figure 3.28 knows how to convert various data types into a histogram. For each type that it knows how to convert to a histogram, `ConvertToHistogram` declares a `convert()` method that takes as its only parameter an instance of that particular data type. In this example, `ConvertToHistogram` knows how to convert from

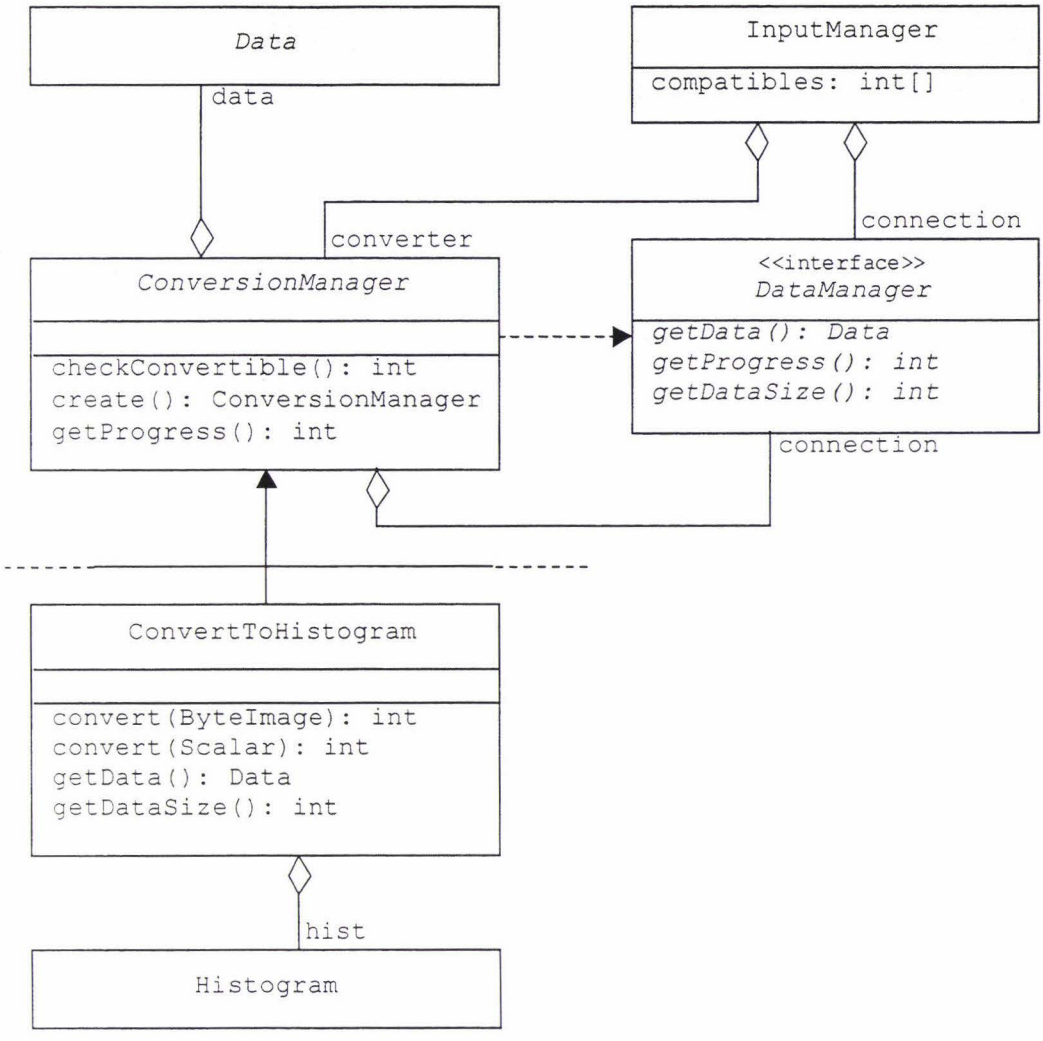

Fig 3.28: *The* `ConversionManager` *class*

both `ByteImage` and `Scalar` types. To convert the former, this conversion manager would simply produce the histogram of the image in question; in the latter case, a histogram could be generated by sampling from a zero-mean normal distribution, using the value of the `Scalar` as its standard deviation[7].

The automatic conversion mechanism can be broken into two stages. The first stage is driven from `InputManager`'s `acquireInput()` method. Once `InputManager` has obtained a copy of the `data` at its corresponding output, its type is compared with all of those in `InputManager`'s `compatibles` array (an array of integers representing the types this `Primitive` is able to process from this input). If the `data`'s type is listed as compatible, then no conversion is necessary

[7] This latter case is included here primarily to demonstrate that, in principle, any conversion that can be imagined is possible.

and the `acquireInput()` method proceeds as described in subsection 3.4.3.1. If it is not, then a check is made to determine whether a `ConversionManager` subclass has been defined that can convert the data to one of the types this input *is* compatible with. The first match that is found is used, and if no match is found then `acquireInput()` simply waits until the data type at this input changes to one that is either compatible, or convertible to a type that is. This change may occur either because a new output gets connected to this input, or because the output that is connected starts producing data of a different type.

`ConversionManager`'s static `checkConvertible()` method performs the search for a suitable conversion manager, and returns an integer representing the type that the data should be converted to. This type is then passed to `create()` (also a static method) which returns an instance of the appropriate `ConversionManager` subclass[8]. Thus the information concerning the conversions that are available is contained within the `ConversionManager` superclass. Once a suitable `ConversionManager` has been created, it is assigned to `InputManager`'s `connection` attribute just as with the `DefaultDataManager` in the previous section. `InputManager` then carries on as normal, unaware that its `connection` has been replaced with an alternative `DataManager`. The `ConversionManager` must be passed the original `DataManager` (through the `create()` method), since this is where it gets the `data` from that needs converting. The object diagrams in figures 3.29a and 3.29b illustrate before and after cases where an `InputManager` has its connection replaced by a `ConversionManager`.

The `ConversionManager` is effectively inserted *in between* an `InputManager` and the `DataManager` that it connects to (in this case an instance of `OutputManager`). As such, it acts like a *proxy*, taking the data from the original output as it becomes available and converting it to the type required by the input. This leads to the second stage of this mechanism, which is where the actual conversion takes place. This stage is driven via `ConversionManager`'s implementation of `getProgress()`. Whenever the programmer invokes `getAvailable()`, `InputManager` calls its `connection`'s `getProgress()` method to determine the number of units currently available at the output it is connected to. When

[8] The `create()` method is an example of the *Factory Method* design pattern (Gamma, *et al.*, 1995).

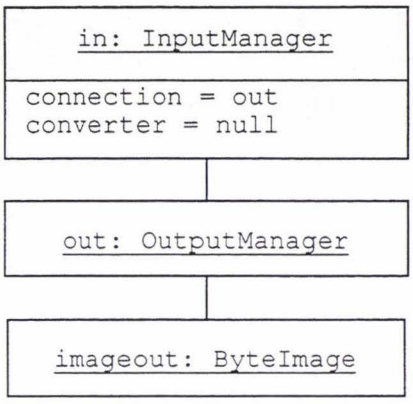

Fig 3.29a: *Before* `ConversionManager` *class is created*

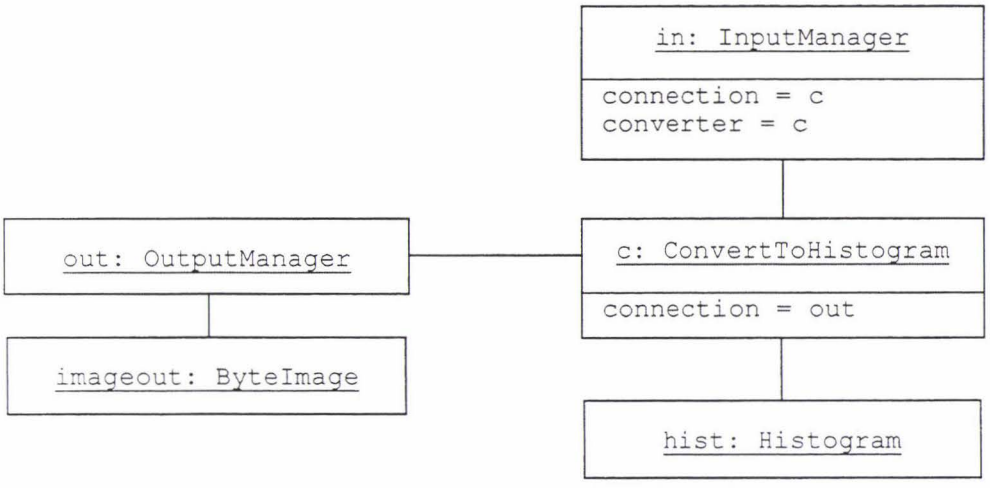

Fig 3.29b: *After* `ConversionManager` *class is created*

`ConversionManager`'s `getProgress()` method is called, this in turn calls the appropriate `convert()` method to convert as much data as it can and returns the amount of converted data that is available as a result.

Note that when a programmer attempts to re-acquire an input that previously came via a `ConversionManager`, the `InputManager` must revert to its original `connection` in order to determine the type of the data being supplied (since it might have changed). If it has changed, then the previous `ConversionManager` is no longer needed and can be discarded. If however, the data type has remained the same (the most likely case) then this `ConversionManager` is still needed. In this case, the `InputManager` must then revert back to its `ConversionManager`. The role of `InputManager`'s `converter` attribute is to store a reference to this `ConversionManager` while it is not being used. In this way, the framework

avoids continually re-creating the same `ConversionManager`, unnecessarily burdening Java's garbage-collector and hence degrade a program's performance.

The framework's `DataManager` interface offers a natural solution for implementing automatic type conversions, and is considered to be the best solution given the framework's overall design. However, the way this mechanism operates has a number of important implications that need to be respected by programmers and developers alike.

The fact that conversions take place at the input end of a connection means that when multiple inputs connect to the same output, there is the potential for a number of otherwise redundant conversion managers to be created. That is, the same conversion may be performed at multiple inputs. Unless these conversions could somehow by shared, this situation would adversely affect programs in terms of both memory requirements and processing efficiency.

One solution in this respect would be to make the `OutputManager` class responsible for handling conversions. It could then easily share the converted data among as many of its inputs as necessary and thus avoid any inefficiencies. However, this approach means that `OutputManager` may be called upon to simultaneously handle a number of different conversions for the data it is managing, depending on the requirements of the inputs to which it is connected. Unfortunately, this increases the complexity of implementing such a solution. The relative advantage of performing the conversion at a connection's input end above is that each `InputManager` only ever needs to handle a single conversion.

A further alternative that was considered during the framework's development was to automatically create and insert special conversion operations as necessary. For example, if one operation produced some data that required conversion so that a second operation could use it, then a conversion operation could be inserted between the two, at the time the conversion was recognised as being required. This solution would involve automatically disconnecting two operations and reconnecting them with the necessary conversion operation in place. It would also involve detecting when this operation was no longer needed and so reverting to the original configuration (discarding the conversion operation at this point).

However, it was decided that this potential solution was not only too complex, but that it went against the spirit of the framework's overall design.

A corollary of using the `DataManager` interface is that it is entirely possible for the framework to automatically convert an input's default data to an alternative type, if this was detected as necessary. Naturally, it would make little sense to set up the conditions where this would occur, but it does demonstrate the elegance of this solution within the framework's design.

3.4.5 Port

The framework utilises the concept of a *port* to model *those points on an operation that may form connections*. As such, an operation's ports refer to its inputs and outputs collectively. Figure 3.30 illustrates the *port* concept.

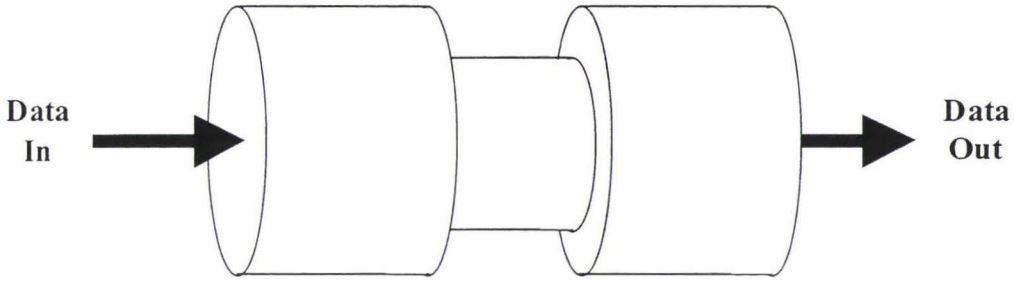

Fig 3.30: *A Port*

The framework realises this concept as an interface, which is implemented by both the `Input` and `Output` classes as shown in figure 3.31 below.

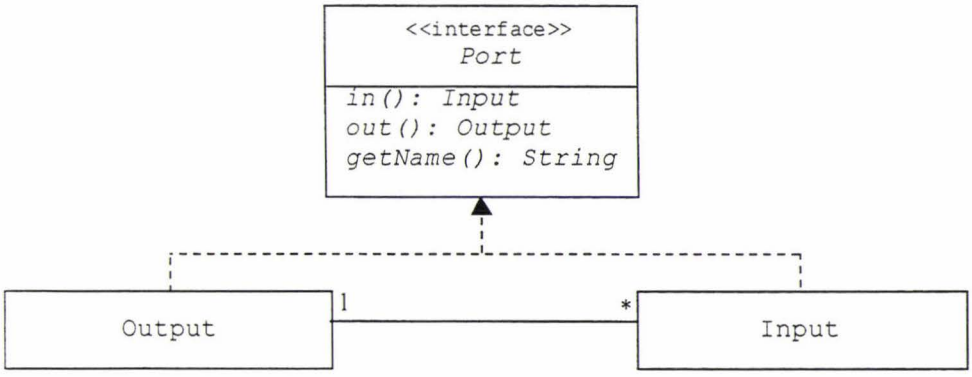

Fig 3.31: *The `Port` interface*

The implementations of `in()` and `out()` for the `Input` and `Output` classes are trivial; for `Input` this is:

```
public Input in() { return this; }
public Output out() { return null; }
```

and similarly for `Output`:

```
public Input in() { return null; }
public Output out() { return this; }
```

Hence, an `Input` is a `Port` that only has an 'in' side (itself), and an `Output` is a `Port` that only has an 'out' side (also itself). The `InputManager` and `OutputManager` classes (which are `Ports` because they subclass `Input` and `Output` respectively) provide the inputs and outputs for `Primitive` operations because they give the *programmer* access to a `Primitive`'s inputs and outputs (via the framework's Data API).

A `Meta`'s `ports`, however, have quite different requirements. For a `Meta`, no programmer access is required since all of its processing is done on the `Meta`'s behalf by its member operations. A `Meta`'s inputs and outputs must simply pass data from external to internal operations (and vice versa). Figure 3.32 below illustrates this concept. Observe that a `Meta`'s inputs (down its left hand side) behave as inputs from the outside, but exhibit the behaviour of *outputs* on the inside. This is most obvious for its lower input, which internally goes on to supply the inputs of *two* of its member operations. Similarly, a `Meta`'s outputs exhibit both input- and output-like behaviour, depending on whether they are considered from the inside or the outside of the `Meta` respectively.

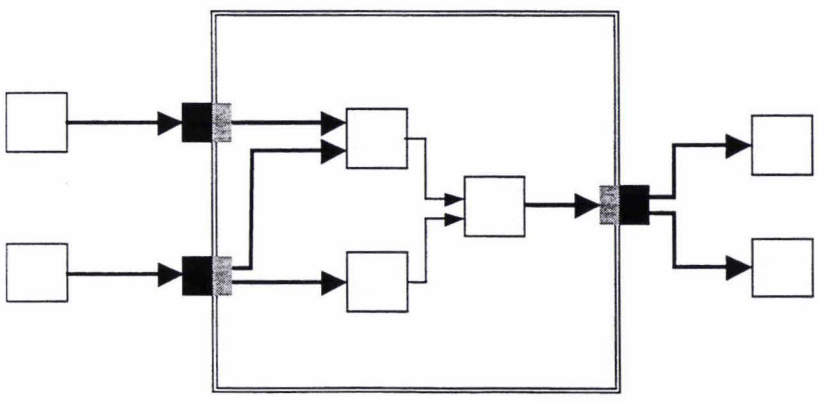

Fig 3.32: *A `Meta` in use*

3.4.6 FeedThru

Rather than try to model these behaviours from scratch, the framework combines the characteristics of its `Input` and `Output` classes into a `FeedThru` class. Figure 3.33 reveals that `FeedThru` is constructed by aggregating an instance of `Input` with an instance of `Output`, and then implementing the framework's `Port` interface.

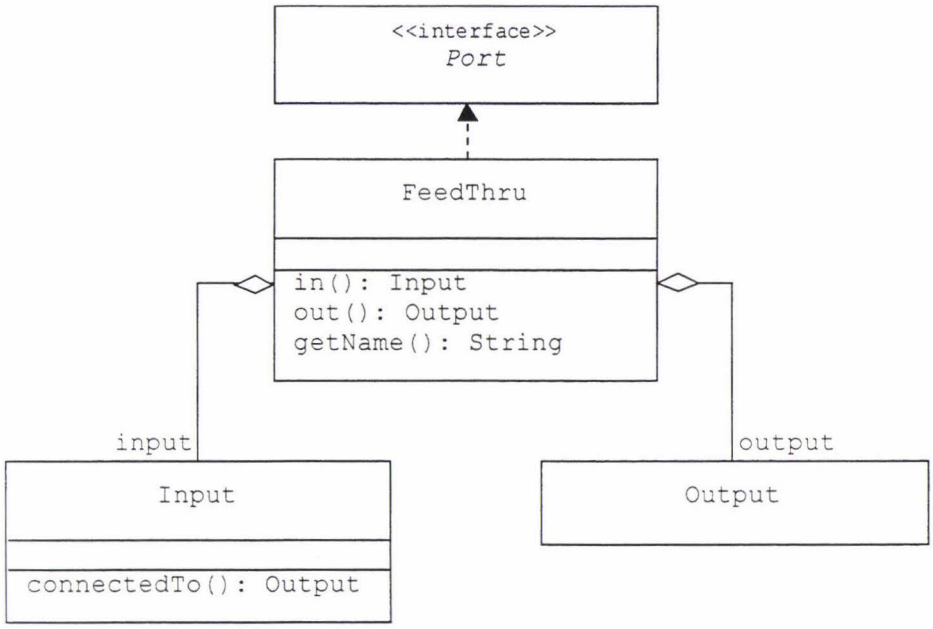

Fig 3.33: *The `FeedThru` class*

Essentially, a `FeedThru` is a junction that an output can connect to, and that in turn can connect to one or more inputs. A `FeedThru` allows data coming into its input to flow out again (via its output), and then on to the inputs that are connected to it, thus it may be used to provide both the inputs and outputs of a `Meta`. The `FeedThru` class is able to provide its functionality through the use of the framework's *binding* relationship, which effectively allows data to flow *from* an input *to* an output; this feature of the framework's design is explained now.

3.4.7 Binding

The data passing mechanisms that have been discussed so far have dealt purely with connections between the output of one operation and one or more inputs of other operations. This one-to-many relationship allowed multiple inputs to observe a single output. The `FeedThru` class, however, relies on a second kind of connection utilised by the framework, referred to as a *binding*. A binding takes

the form of a one-to-one connection that may exist between an input and an output. Figure 3.34 reveals how the binding relationship is implemented within the framework.

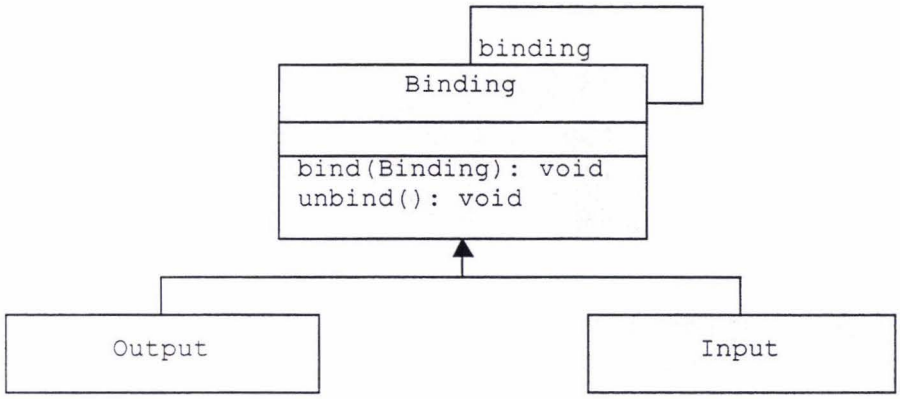

Fig 3.34: *The* `Binding` *class*

The framework's `Binding` class contains one attribute, named `binding`, which is a reference to an instance of its own class to which it is *bound*. The binding relationship is defined to be reciprocal, that is, its semantics dictate that for two `Bindings` *a* and *b*, if *a* is bound to *b* then *b* will be bound to *a* (the exception being that no instance of `Binding` may be bound to itself). `Binding`'s two methods, `bind()` and `unbind()`, may be used to make or break bindings between instances of `Binding`; their roles are to preserve *binding*'s semantics. The properties of the `Binding` class are used to provide the `Input` and `Output` classes with the ability to establish one-to-one relationships between themselves (i.e. by binding to each other). `Binding` passes on its capabilities to both `Input` and `Output` by superclassing them (as shown above). Figures 3.35a and 3.35b compare and contrast connections and bindings.

In both of these diagrams, the conceptual flow of data is from left to right; hence, a binding allows data to flow *from* an input *to* an output. This 'contradictory' behaviour is achieved by the framework's *feedthru* mechanism, which utilises the binding relationship to *propagate* the programmer's Data API calls. This mechanism allows connections between `Primitives` (i.e. between `InputManagers` and `OutputManagers`) to be established that go via one or more intermediate `FeedThru` junctions. In this way, operations that are connected to each other across a `Meta`'s boundary need not be aware of the fact they are actually communicating via an intermediate `FeedThru`. Figure 3.36 illustrates a

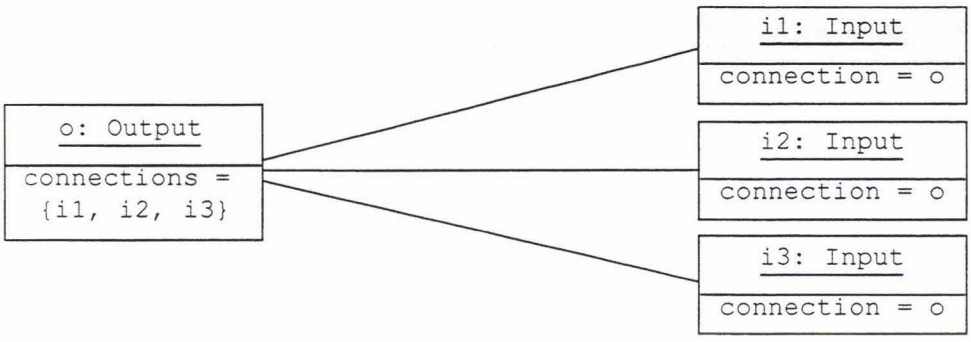

Fig 3.35a: *Connections from an output to multiple inputs*

Fig 3.35b: *A binding from an input to an output*

simple situation where two primitives, `op1` and `op2`, are connected to each other via a `FeedThru` (this is necessary because `op2` is contained within a `Meta` to which `op1` is external).

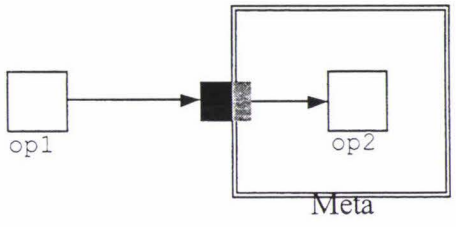

Fig 3.36: *A connection made via a* `FeedThru`

When data is passed between `op1` and `op2`, the programmer's Data API calls are propagated in their respective directions as necessary; examples for both directions are given in figures 3.37a and 3.37b. The scenario in figure 3.37a begins with a call to `OutputManager`'s `validate()` method, (or equivalently with any of `OutputManager`'s Data API methods). As before, the instance of `OutputManager` responds by calling all of the `recv()` methods of its connected `Inputs`. When it calls `recv()` on the `FeedThru`'s `Input`, this propagates the original call to `send()` on to the `Output` to which it is bound, which in turn calls all of the `Inputs` it knows about. In this way, the progress information being broadcast ultimately reaches all interested `InputManagers`, regardless of whether they are connected directly to the originating `OutputManager` or indirectly via `FeedThrus`. The `InputManager` in this scenario cannot continue to propagate the calls because it is not bound to anything.

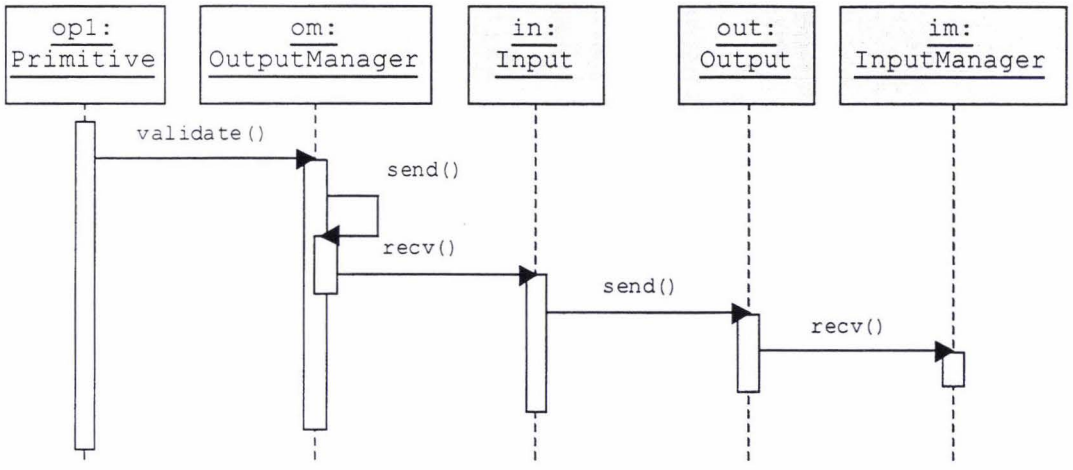

Fig 3.37a: Propogating calls from an output to an input

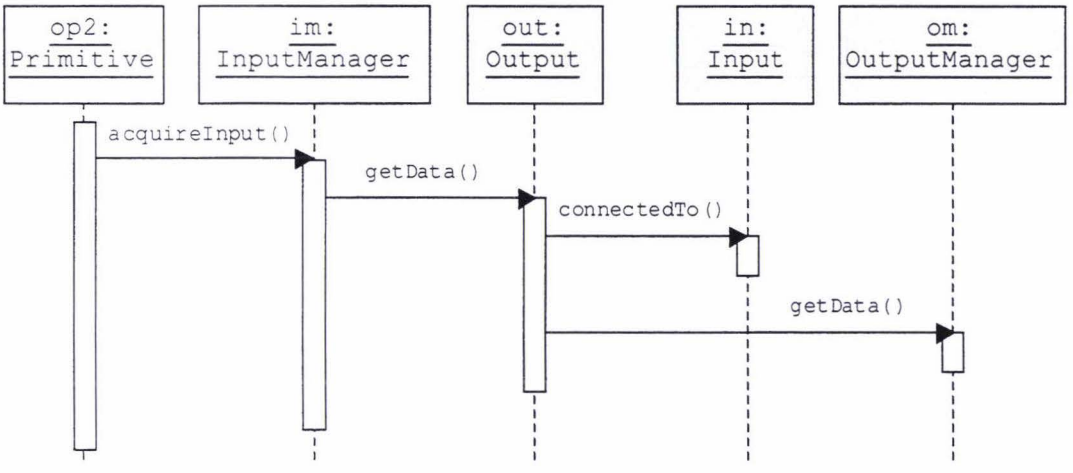

Fig 3.37b: Propogating calls from input to output

This second scenario (see figure 3.37b) shows how calls to `InputManager`'s Data API are propagated. Consider a call to `acquireInput()` (or any `InputManager`'s Data API call). `InputManager` responds by calling `getData()` on the `Output` to which it is connected. This in turn invokes the `connectedTo()` method of the `Input` it is bound to. The `connectedTo()` method returns the instance of `OutputManager` that this bound `Input` is itself connected to. The original `getData()` call is then propagated on to this `OutputManager`. All three of `DataManager`'s methods may be propagated in this way, so that any of `InputManager`'s Data API calls will work over an indirect connection. If an `Output` finds that it is not bound to any `Input`, then instead of propagating an incoming `DataManager` call, it executes it. In this example, for instance, the `OutputManager`'s `getData()` method would return with a copy of this output's `Data` instance.

The strength of the framework's *FeedThru* mechanism is that it is completely transparent from the perspective of the `InputManager`s and `OutputManager`s that must ultimately communicate with each other. Further, there is no limit to number of intermediate `FeedThru`s a connection may traverse (multiple `FeedThru`s would be necessary where a `Meta` operation encapsulates other `Meta`s). Note that the `FeedThru` class itself has no direct role in the operation of the framework's data passing mechanisms, as demonstrated by the previous two sequence diagrams. Rather, it simply aggregates instances of `Input` and `Output`, which themselves propagate the necessary Data API calls.

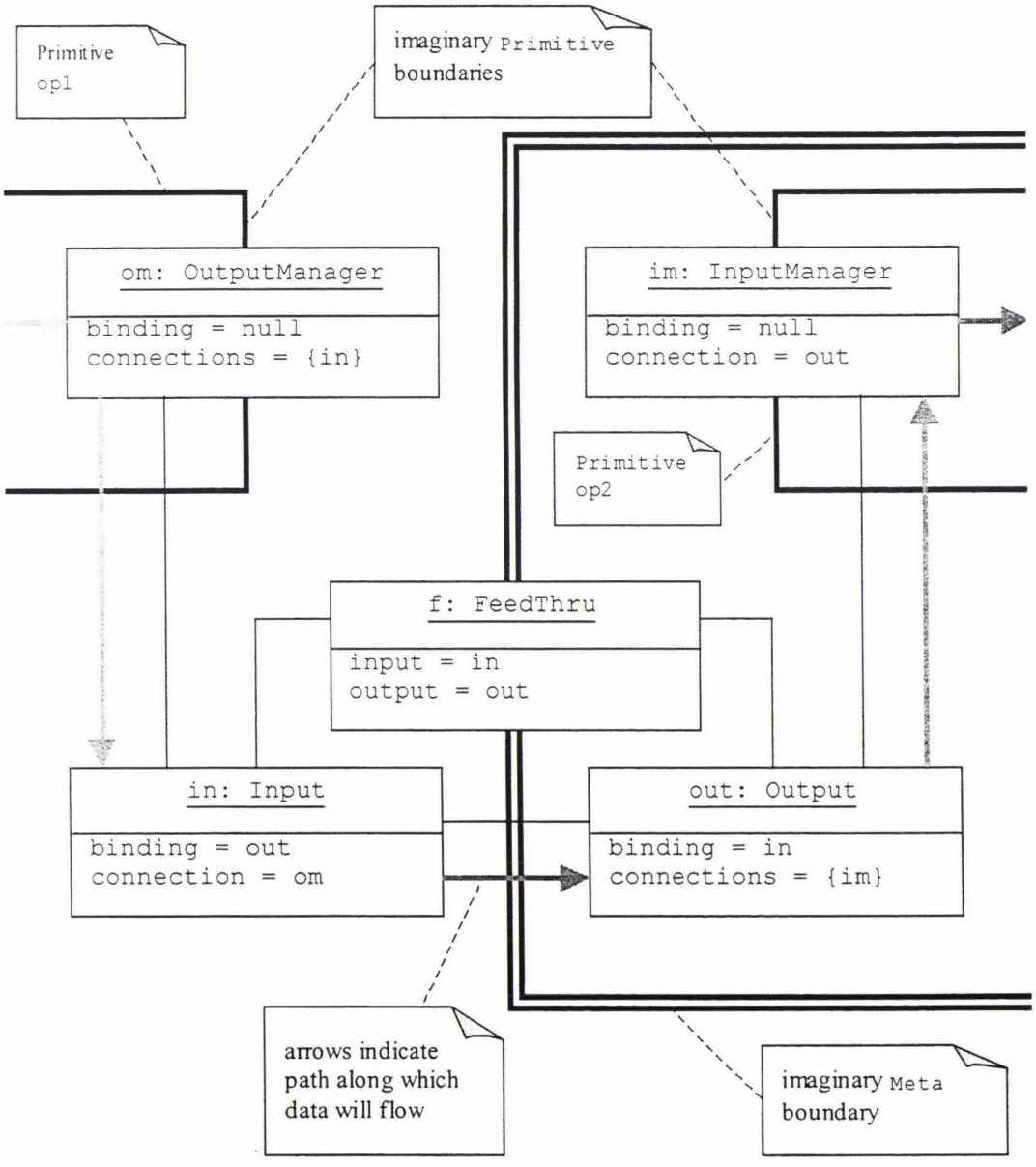

Fig 3.38: *A `FeedThru` in use*

Figure 3.38 illustrates a 'zoomed in' view of the object structure that would exist for the simple example that was given in figure 3.36 above. The object diagram in this figure has been augmented with some additional information; it shows where the various I/O objects belong in terms of their owner operations, and the relationships between them which allow data to flow along the intended connection path.

As the `Primitive op1` produces its data, the `OutputManager` at its output supplies this data to the `Meta`. This data then flows, via the `Meta`'s `FeedThru`, to the encapsulated `Primitive`'s `InputManager` (belonging to `op2`); in this way, `op2` is able to acquire and process its input data (on behalf of its encapsulating `Meta`).

3.4.8 Inputs, Outputs, Sources and Sinks

3.4.8.1 Connection Points

To account for the distinction between the internal and external 'sides' of a `Meta` (see figure 3.32 above) the framework employs the terms *input*, *output*, *source*, and *sink* to refer to the various points on any operation where a connection may be physically established (as illustrated in figure 3.39). While a source is really an output, and a sink is really an input, source and sinks are distinguished by being the internal connections of the `Meta`.

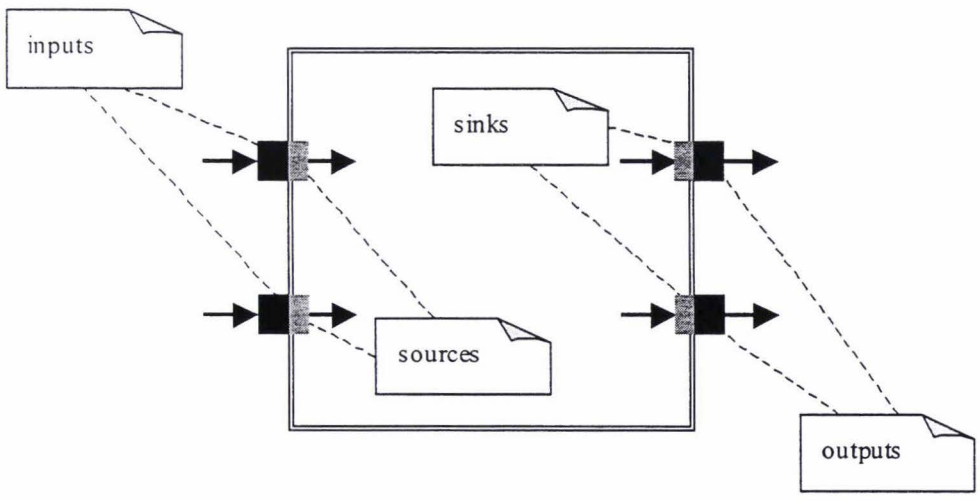

Fig 3.39: *Inputs, outputs, sources and sinks*

There are four distinct connection scenarios that are considered legal by the framework. The diagram in figure 3.40 labels these scenarios as **a**, **b**, **c**, and **d**. These are briefly described here:

(a) This is the standard case, where two different `Operator`s are encapsulated by the same `Meta`, and a connection is established between them. The output of `op1` is connected to the input of `op2`.

(b) This is where a `Meta` is connected to itself. In this case its sink is connected to its source, effectively by passing any intermediate operations it may contain.

(c) In this case an `Operator`'s input is being supplied by whatever is being input to its encapsulating `Meta`. The input of `op1` is connected to the source of the `Meta`.

(d) The contrary case to **c** above. The sink of the `Meta` is connected to the output of `op2`.

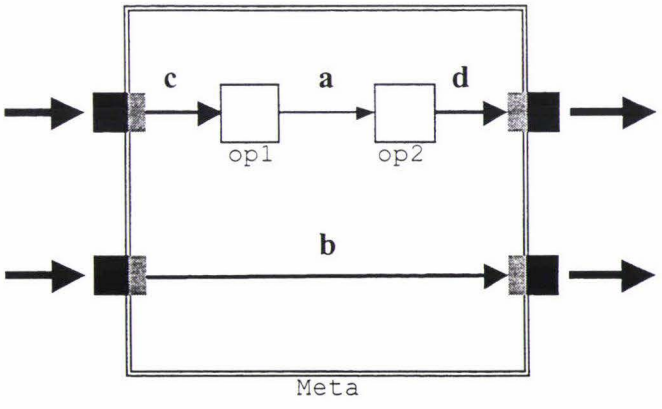

Fig 3.40: *The four possible connection scenarios*

Any other connection scenario would cross the boundary of the `Meta`, and so is considered illegal by the framework. This is because it would violate the encapsulation role of the `Meta` operation; if such a connection is attempted then the it fails and no connection is established. Figure 3.41 illustrates such an illegal scenario.

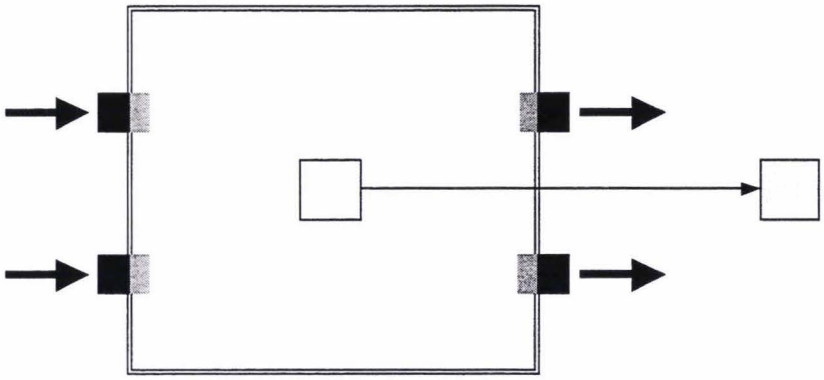

Fig 3.41: An illegal connection attempt

3.4.8.2 PortManager

`Operator`'s `connect()` method is called to establish connections between operations. This method relies on the framework's `PortManager` class, which provides named access to an `Operator`'s `ports`. Figure 3.42 illustrates this class in the context of the classes and methods that make use of its services.

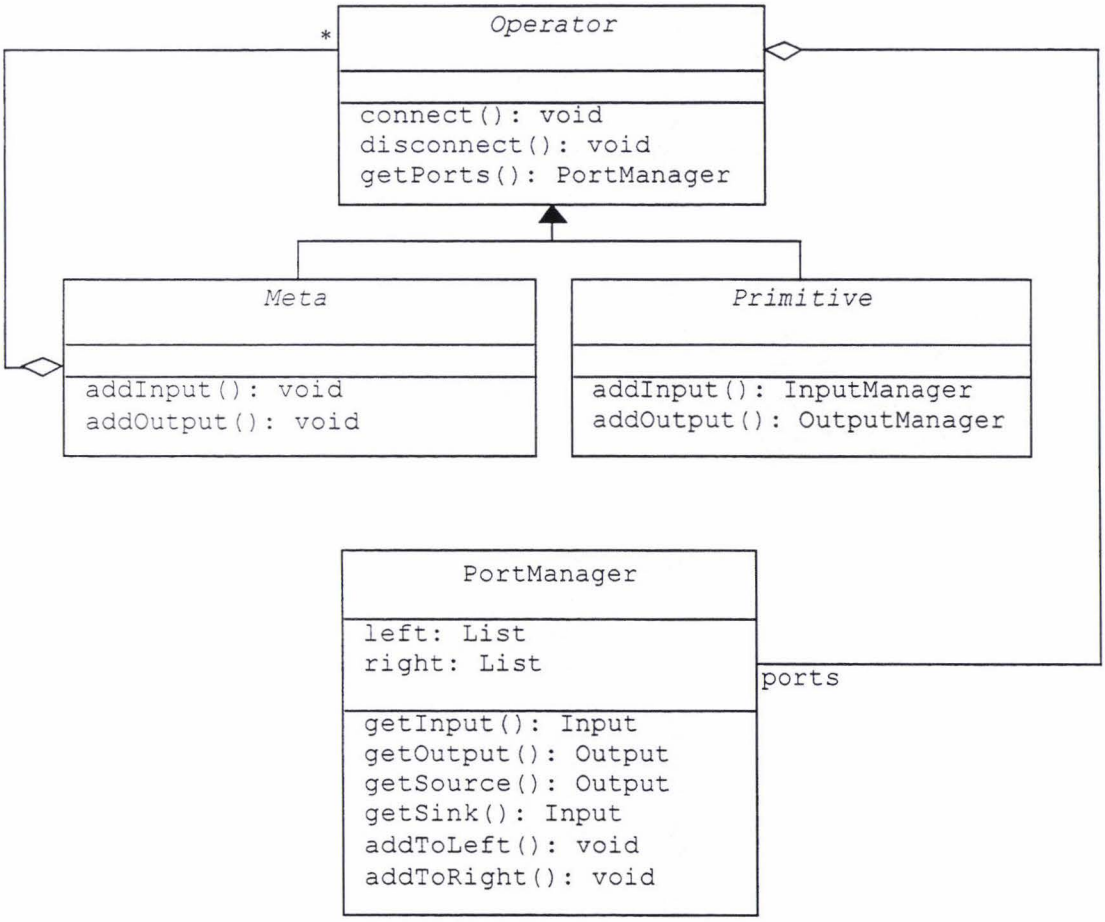

Fig 3.42: The `PortManager` class

The `PortManager` class maintains two lists; one contains all of an `Operator`'s `left` hand ports (those that take data in), and the other all of its `right` hand ports (where data is output). It's `addToLeft()` and `addToRight()` methods add a new `Port` to the appropriate list; this new `Port` is passed in as a parameter. These methods are used by the `addInput()` and `addOutput()` methods of both the `Meta` and `Primitive` classes to create new inputs and outputs. When applied to a `Meta`, a new `FeedThru` is created and added. For a `Primitive`, an `InputManager` or an `OutputManager` is created as appropriate, and added. In this latter case the new `InputManager` or `OutputManager` is also returned to the programmer, who may use it to access that `Port` from the `Primitive`'s `implementation()` method.

`PortManager` provides access to the `Port`s that it is managing through its *get* methods, which return the specified port by name. When establishing a connection, calls to `Operator`'s `connect()` method rely on `PortManager`'s *get* methods to access the appropriate connection points on the operations being connected.

Note that calling `getSource()` or `getSink()` on a `Primitive`'s `PortManager` will return with `null`. This is because `InputManager`s and `OutputManager`s are one-sided `Port`s, and have no internal connection points as such. This prevents internal connections being established between a `Primitive` and itself (the way that a `Meta` is able to in scenario **b** in figure 3.40 above).

3.4.9 Summary

This section has covered the classes and APIs that participate in the framework's data passing mechanisms. The *Observer* design pattern (Gamma, *et al.*, 1995) provided the basis for enabling the flow of data between operations. This pattern is realised by the framework's `Input` and `Output` classes, which together implement a one-to-many relationship, allowing multiple `Input`s to observe a single `Output`. In order to simplify the process of connecting `Input`s and `Output`s together, connections are always made and broken from their input ends, via calls to `Input`'s `connect()` and `disconnect()` methods.

To ensure programs appear responsive to users, the framework's *progress* mechanism enables data to be passed between operations at an arbitrary level of

granularity. The advantage of this from the user's perspective is that inputs can be changed and results recalculated without the need to wait for any given operation to complete its processing. The implication for the programmer is that, for a given data type, the measure of data chosen to represent a single unit cannot be separated from that type itself. Hence, operations that use a particular data type must be implemented to follow the *protocol* that such a type defines.

Within a `Primitive`, the programmer gains access to its inputs and outputs through the framework's Data API, which is provided by the `InputManager` and `OutputManager` classes. `OutputManager` provides methods that control the initialisation and finalisation of a `Primitive`'s outputs, and the availability of the data at a given output during processing. `InputManager`'s methods may be used to first 'activate' a connection by acquiring the data at a given input, and then to regulate access to this data so that it may be processed as it becomes available progressively. The aim of the Data API is to provide the programmer with almost transparent access to an operation's inputs and outputs, so that the necessary processing code can be implemented with a minimum of concern for the actual 'flow' of data into and out of the various operations in a program.

The `InputManager` class relies on the framework's `DataManager` interface to provide access to the data. Under normal circumstances this is from an `OutputManager`, but could also be a `DefaultDataManager` or `ConversionManager`. `DefaultDataManager` provides an input with a default value in the event that it has no physical connection, and `ConversionManager` handles automatic type conversions. The `DataManager` interface provides a convenient and effective solution in both of these cases because it frees the `InputManager` class from having to know about where its data is coming from, thereby simplifying its implementation.

The framework's `Port` interface allows instances of `Input` and `Output` to be aggregated (in different combinations) to form the 'points of connection' through which operations may be interconnected. The `Input` and `Output` classes implement this interface themselves, thereby providing the basis for the `InputManager` and `OutputManager` classes. These two classes are in effect 'one-sided' ports that interface between a `Primitive`'s inputs and outputs and the programmer. The framework's `FeedThru` class, on the other hand, is a 'two-

sided' port that is used by `Meta` to connect between operations that are external and those that internal to a `Meta`. The `FeedThru` class aggregates an `Input` and an `Output`, and utilises the framework's *binding* relationship to implement the framework's *FeedThru* mechanism. This mechanism propagates Data API calls in their respective directions so that `Primitive` operations may continue to pass data between themselves, despite being separated by one or more intermediate `FeedThru` junctions. The strength of this mechanism is that it is transparent to the `InputManager`s and `OutputManager`s that are passing data to each other.

An operation's points of connection may be thought of as falling into four categories: *inputs*, *outputs*, *sources*, and *sinks*. The former pair are external to an operation. They share their meaning for both `Primitive` and `Meta` operations; these are the points where a given operation connects externally to other operations in a program. The latter pair have alternate interpretations for `Primitive`s and `Meta`s. For a `Meta`, this pair represent the internal connection points that allow a `Meta`'s member operations to connect (indirectly) to their external counterparts. A `Primitive`, on the other hand, has no sources or sinks as such; rather, its internal 'connection points' are provided through the framework's Data API, which interfaces with the programmer.

An operation's `Port`s, which represent its points of connection collectively, are managed through an instance of the framework's `PortManager` class. This class provides named access to an operation's `Port`s, and facilitates `Operator`'s `connect()` method in establishing new connections. It achieves this by providing methods that make it convenient to access an operation's `Port`s in terms of its inputs, outputs, sources and sinks.

3.5 Scheduling

The framework's scheduling mechanisms control when and how an operation is invoked, how its processing may be temporarily suspended if it runs out of input data, and how it may be woken up again when more data becomes available. An operation may also need to completely restart if one or more of its inputs become invalid during processing.

In Java, control of a program can be spread across multiple threads through its language-level support for multi-threading. The framework's scheduling mechanisms are designed on the basis that each `Primitive` operation in a program carries out its processing within its own thread of control. Figure 3.43 illustrates this concept.

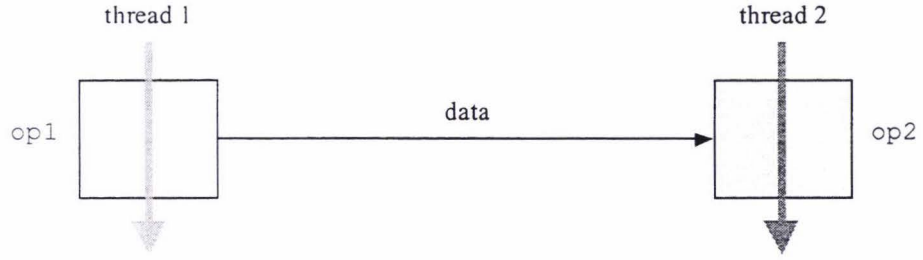

Fig 3.43: *Each* `Primitive` *runs in its own thread of control*

In section 3.4, it was shown how `InputManager` and `OutputManager` communicate with each other, thereby allowing data to be passed progressively between `Primitive`s. This section will describe how these communication mechanisms facilitate the control of a given `Primitive`'s thread. It also discusses sequencing of operations, which involves ensuring that `Meta` operations are never self-nested, and ensuring that co-dependencies between interconnected operations are avoided.

3.5.1 Blocking and Unblocking

A `Primitive` must be able to wait (if necessary) for new data to become available, and then resume processing when it does. This is achieved through the framework's *blocking* mechanism, which relies on `Primitive`'s `block()` and `unblock()` methods (see figure 3.44). The `block()` method puts a `Primitive`'s thread to sleep; this method is called by a `Primitive`'s `InputManager` when it

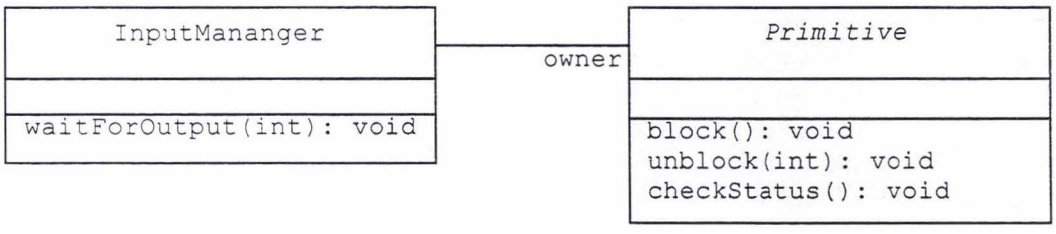

Fig 3.44: *The framework's blocking mechanism*

needs to wait for its corresponding output to reach the requested progress level. `InputManager` performs this wait through its `waitForOutput()` method, which continually calls `block()` on its `owner` until its output's progress reaches the required level (see the sequence diagram in figure 3.45).

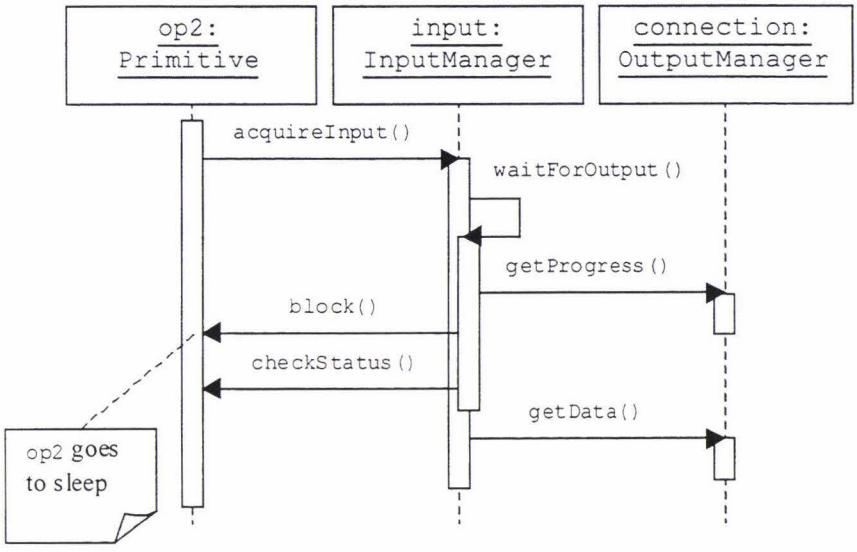

Fig 3.45: *Waiting to acquire an input*

A `Primitive`'s thread may be woken up (i.e. its `block()` method can be made to return) by a thread from another `Primitive` calling its `unblock()` method. This occurs whenever an output's progress level changes (see figure 3.46). Hence, the `block()` and `unblock()` calls work together. The two figures below indicate the points where the threads belonging to `op1` and `op2` synchronise with each other.

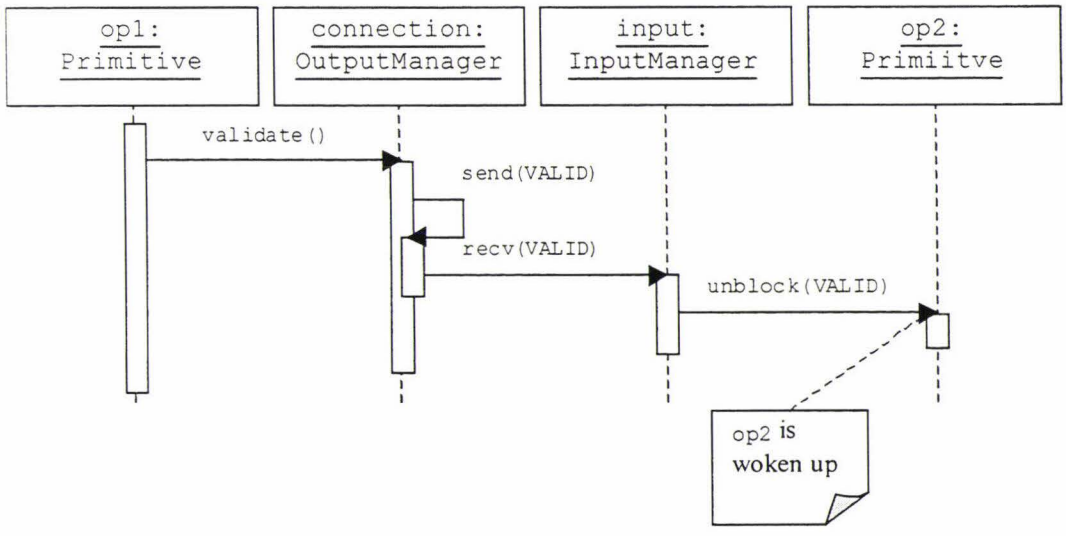

Fig 3.46: *Unblocking a* `Primitive`

3.5.2 Primitive's Main Loop

Each `Primitive` operation has a *main loop*, which executes continuously independent of all other `Primitives`. The main function of this loop is to call `Primitive`'s `implementation()` method, wherein its processing is carried out. Figure 3.47 illustrates the class structure, methods and attributes that facilitate `Primitive`'s main loop.

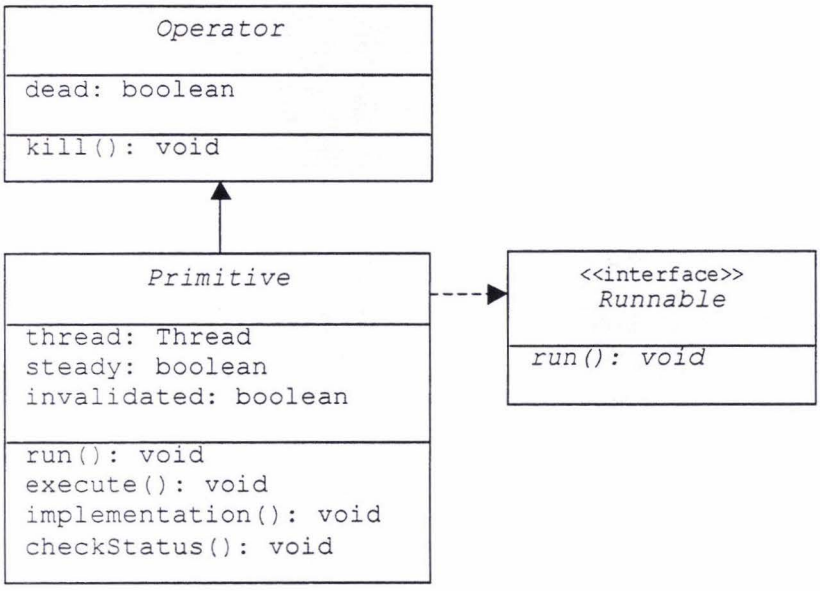

***Fig 3.47:** Implementing `Primitive`'s main loop*

`Primitive` realises Java's `Runnable` interface, meaning that this class is capable of being the target of a `Thread`'s `run()` method. `Primitive`'s `thread` attribute references an instance of Java's `Thread` class (which represents Java's threading abstraction); this is used to provide each instance of `Primitive` with its own thread of control. Every `Primitive` executes the following algorithm for its main loop:

```
while (!dead)
  if (!steady) {
   // method to carry out processing
   implementation();
   steady = true;
  }
  else {
    // sleep until one or more inputs become invalidated
    block();
    if (invalidated) steady = false;
  }
```

`Primitive`'s `run()` method implements the enclosing while-loop, which continually calls its `execute()` method. The `execute()` method implements the body of the loop. `Primitive`'s main loop is separated over two methods to facilitate scheduling multiple primitives within a shared thread, which is discussed further in section 3.5.3 below. This loop relies on three flags:

`dead`: Every `Operator` maintains a `dead` flag which indicates when an operation is no longer needed. This allows the developer to tell any operation to die, which in `Primitive`'s case means exiting its main loop, disconnecting all of its inputs and outputs, and allowing itself to be garbage collected. The developer may kill any operation at any time by calling its `kill()` method. At construction time, `dead` is set to false, and is subsequently set to true via a call to `kill()`. `Primitive` uses its `dead` flag to determine when to terminate its main loop. Note that programmers may override an operation's `kill()` method when subclassing either `Primitive` or `Meta` in order to handle a particular operation's own 'clean up' requirements. In these cases, the superclass' `kill()` method must be called so that the operation is properly cleaned up.

`steady`; `Primitive`'s `steady` flag indicates when all of a `Primitive`'s input data has been processed and its outputs are complete (i.e. the operation is in a *steady state*). This flag is initially set to false, and is set to true on the successful completion of the `Primitive`'s `implementation()` method. Once this flag has been set to true, the `implementation()` method is no longer invoked; instead, the `Primitive`'s thread simply goes to sleep by calling its `block()` method. The `steady` flag is reset to false when a `Primitive` is woken up and finds that its `invalidated` flag has been set to true. At this time, `Primitive`'s main loop re-iterates and its `implementation()` method is re-invoked.

`invalidated`; `Primitive`'s `invalidated` flag is set to true whenever *any one* of its `InputManager`s calls its `unblock()` method and passes in the progress level INVALID. Hence, this flag indicates when *at least one* of a `Primitive`'s inputs has recently been invalidated. The `invalidated` flag is checked by `Primitive`'s `checkStatus()` method, which has the responsibility for initiating the necessary restart. For the case where a `Primitive` is currently processing data, this flag causes an exception to be thrown, in turn causing its `implementation()` method

to abort and restart. The framework's use of exceptions to correctly control `Primitive`'s main loop is covered in the following section.

3.5.3 ControlFlowException

A `Primitive`'s thread may restart its main loop at any time by throwing an instance of `ControlFlowException` (see figure 3.48). `Primitive`'s `checkStatus()` method is called to determine whether or not an exception needs to be thrown. It is automatically called whenever a `Primitive` calls one of its `InputManager` Data API methods (as shown in figure 3.45 above), but it may also be called directly by the programmer from `Primitive`'s `implementation()` method. A programmer may wish to do this if a `Primitive` is carrying out a long calculation and is not regularly calling `InputManager`'s Data API methods (for instance, because all of the required data is already available). In that case, making periodic calls to `checkStatus()` will ensure that processing will be aborted in a timely manner should an input become invalidated.

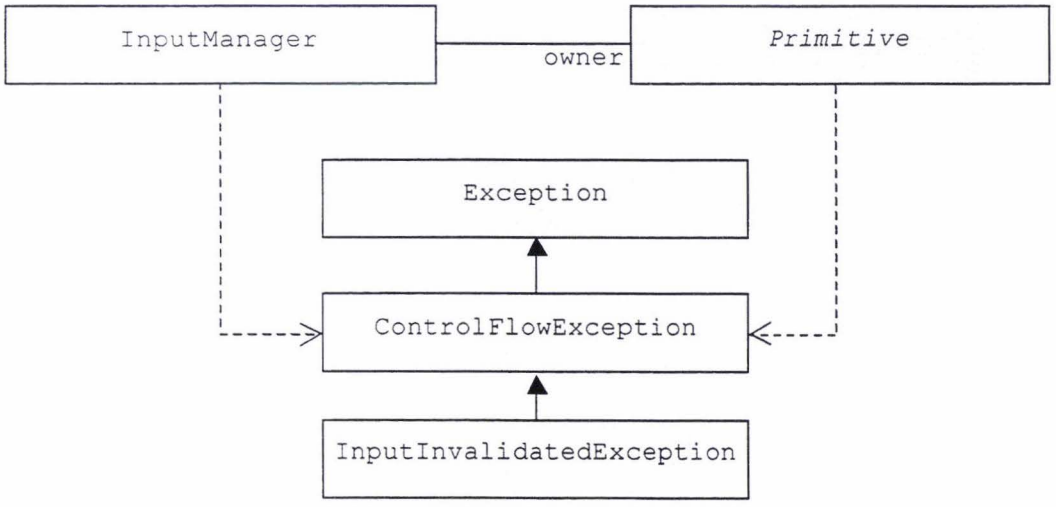

***Fig 3.48:** Implementing `Primitive`'s main loop*

The `checkStatus()` method will do nothing if all of a `Primitive`'s inputs have remained valid. If `Primitive`'s invalidated flag is set, then this flag is reset and `InputInvalidatedException` is thrown. This exception will cause a restart in all cases *except* when the programmer has called `acquireInput()` (this method catches this exception). Because `acquireInput()` is not meant to return until the input being acquired becomes valid, a restart is not necessary in this circumstance. A restart is *always* required, for instance, when it is detected that a `Primitive`'s

`dead` flag has been set, so in this case a `ControlFlowException` would be thrown. This exception is guaranteed to cause a restart.

3.5.4 Shared Threads

Although conceptually, every `Primitive` is considered to run within its own thread, this is not ideal from a resource allocation perspective, especially where a program utilises many `Primitive` operations. Hence, the framework provides an alternative for `Primitive`s that do not need or want their own separate thread. For example, the programmer may decide that the processing performed by a particular `Primitive` is too simple to warrant its own thread, or the developer may decide to limit the number of threads used by his or her program. The solution is to provide a pool of shared threads that can be drawn upon by `Primitive`s that do not have their own thread. Using the framework's *shared thread* feature, any `Primitive` may be configured with or without its own thread.

The framework is designed so that from the programmer's perspective there is no difference between implementing a `Primitive` that is intended to run in its own thread, and one that is not. Any `Primitive` may operate in either mode. The framework distinguishes between `Primitive`s in each mode using the concept of a `Primitive`'s *weight*: a *heavyweight* `Primitive` has its own thread, whereas a *lightweight* `Primitive` has no thread of its own and therefore must utilise a shared thread.

From a scheduling perspective, there is a difference between light and heavy `Primitive`s. A heavyweight `Primitive`'s thread is exclusively dedicated to carrying out that `Primitive`'s processing; therefore, it may be put to sleep for as long as necessary while it waits on an input's data. However, because one or more other `Primitive`s may share a lightweight `Primitive`'s thread, it is never allowed to block. Rather, a lightweight `Primitive` is only allocated a shared thread when all of its inputs are in the COMPLETE state, that is, all of the data it requires is completely available. When this occurs, it's `execute()` method is invoked to initiate a single iteration of the `Primitive`'s main loop (hence the reason for separating a `Primitive`'s main loop over a pair of methods). The `execute()` method invokes the `Primitive`'s `implementation()` method, giving it a single chance to process its data and reach its steady state. If at any time

before its processing completes the `Primitive` tries to block for any reason, or any of its inputs are invalidated, an exception is thrown and the shared thread is returned to it's pool. Figure 3.49 illustrates the framework's `ThreadPool` class together with `Primitive`'s support for thread sharing.

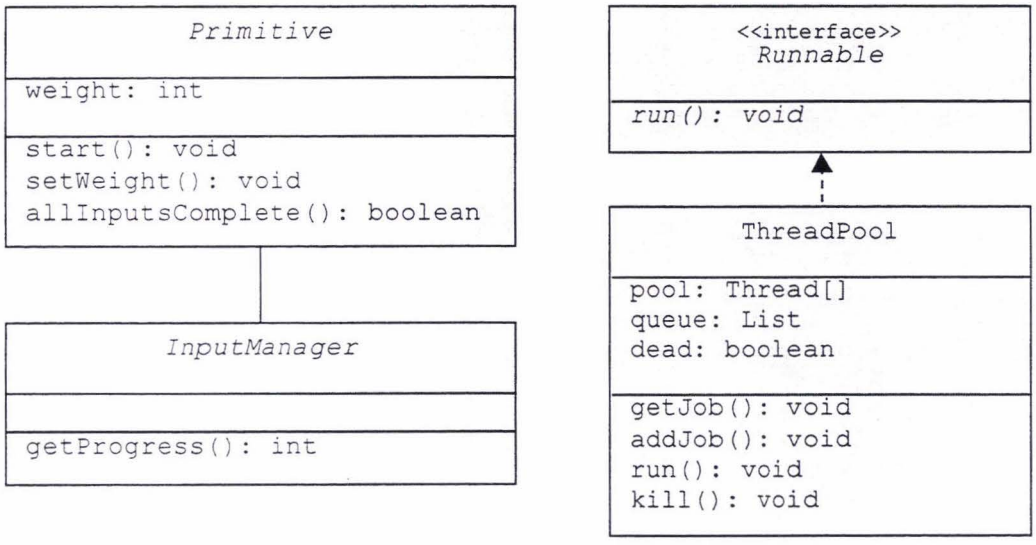

Fig 3.49: *The `ThreadPool` class*

Every program created using the framework has access to one `ThreadPool`, which provides shared threads for all `Primitive`s in that program. The thread sharing services are accessed through a `Primitive`'s `unblock()` method. For a heavyweight `Primitive`, the `unblock()` method wakes up its thread. For a lightweight `Primitive`, the framework calls its `allInputsComplete()` method. This method iterates through all of the `Primitive`'s `InputManager`s, calling `getProgress()` on each. Such a `Primitive` may only execute when every one of its `InputManager`s reports that its `connection`'s progress is in the COMPLETE state. If this is the case, the `Primitive` is added to `ThreadPool`'s job `queue` by passing it as a parameter to `ThreadPool`'s `addJob()` method. Once at the front of this `queue`, the next shared thread to become available will remove the `Primitive` from the `queue` via a call to `getJob()` and invoke its `execute()` method.

The number of shared threads in a `ThreadPool` is determined at its construction time. `ThreadPool`'s `run()` method implements the shared thread version of `Primitive`'s main loop:

```
while (!dead)
  if (!queue.isEmpty()) {
    Primitive job = getJob();  // remove first job in queue,
    job.execute();             // and then execute it
  }
  else
    try { wait(); }
    catch(InterruptedException ex) {}
```

Whenever a job is added to `ThreadPool`'s queue via a call to `addJob()`, all sleeping threads in the thread pool are woken up via a call to `notifyAll()`[9]. Only one will get the new job; if this was the only job then the rest simply go back to sleep. `ThreadPool` also maintains a `dead` flag so that it too can be killed when necessary.

A `Primitive` may be configured as heavyweight or lightweight at the time it is constructed via a call to its `start()` method. This method takes a single integer parameter, which may be either HEAVY or LIGHT (these values are constants defined by the framework). This should be the last call that is made before the constructor exits. The programmer may allow the developer to specify an operation's initial weight by providing a parameter in the operation's constructor for this purpose. Note that the `weight` of a `Primitive` can be changed at any later time via a call to `Primitive`'s `setWeight()` method. The `Primitive` will then adopt its new scheduling behaviour at the earliest convenient time after this call is made.

3.5.5 *Workspace*

The framework requires that every `Operator` has a `parent`, which is the `Meta` that encapsulates it (see figure 3.50). This makes it easier for the framework to 'supervise' the operations in a program, since every operation will be listed as a member of its `parent Meta`. For instance, when a Meta is killed each of its member operations can be killed by iterating through its member list. The role of the framework's `Workspace` class (see figure 3.51) is as a root `Meta`. Every program created using the framework must utilise a `Workspace`, which will directly or indirectly encapsulate all of the program's operations; `Workspace` is therefore the only `Operator` that has no `parent`.

[9] Java's `wait()`, `notify()` and `notifyAll()` methods may be called on any object. They are used to halt and then re-start the thread that is currently active in an object.

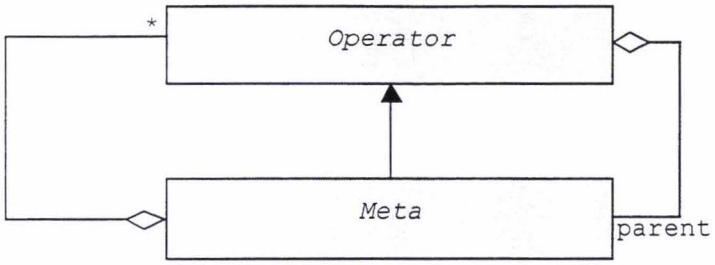

Fig 3.50: An `Operator`'s parent

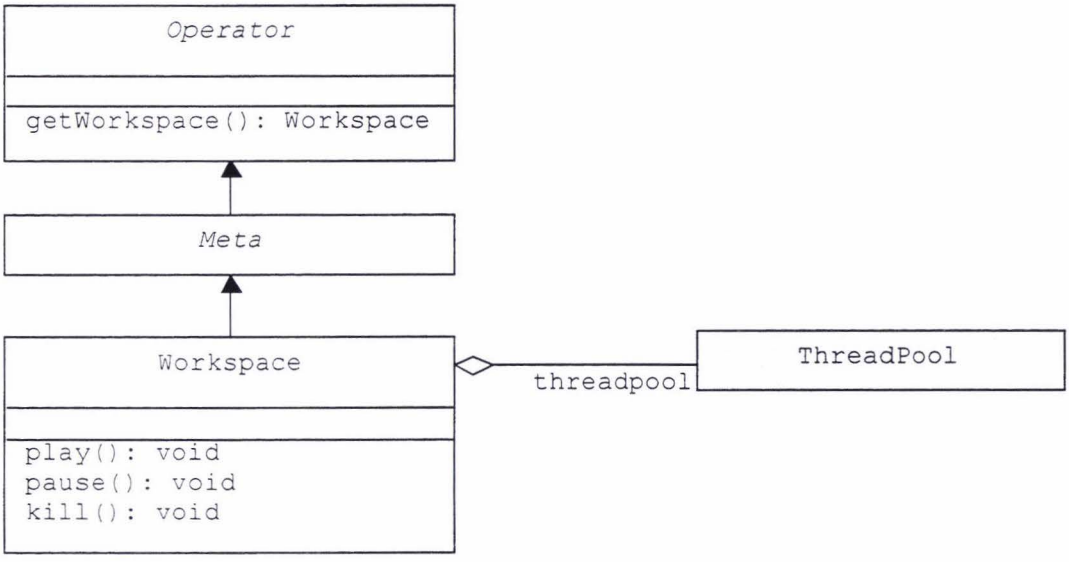

Fig 3.51: The `Workspace` class

Every `Operator` has access to its encapsulating `Workspace` instance via its `getWorkspace()` method. Through this method, any `Primitive` may access its `ThreadPool`, which is attributed to `Workspace`. `Workspace`'s `pause()` and `play()` methods suspend or resume all of the threads of all of the operations in a program, including shared threads. These methods allow a program to be temporarily halted and then restarted from where it was stopped. This functionality is implemented using Java's `ThreadGroup` class, which is used to manage multiple threads.

3.5.6 Sequencing

To simplify the implementation of the framework, and to prevent possible scheduling lockouts, mutual dependencies are not allowed. Two specific situations that the framework detects and prevents are: 1) where a `Meta` is self-nested; and 2) where operations are interconnected such that they are co-dependent.

3.5.6.1 Self-nested Metas

A self-nested `Meta` is one that is contained (either directly or indirectly) within a second `Meta` that already contains it; this situation must be prevented because it would lead to an infinite loop when attempting to traverse every operation in a program. This is equivalent to saying, *a* cannot contain *b* if *b* already contains *a*. In order to avoid `Meta`s being self-nested, `Meta`'s `addOp()` method conducts a search up through its `parent` hierarchy looking for the operation being added. If this search ends with a `null` reference, then the program's root-level `Workspace` has been found and `addOp()` is successful; if the search instead leads to the candidate member, then there is a loop and the `addOp()` method fails.

3.5.6.2 Co-dependent Operators

Before a connection is established, it must be determined that such a connection will not lead to a co-dependency. This equivalent to saying if there is a connection from the output of *a* to the input of *b*, then no connection can be established from the output of *b* to the input of *a*. Co-dependencies result in an operation depending on its own outputs. Such an operation is not well defined under the scheduling mechanisms described earlier in this chapter.

Detecting co-dependencies is complicated by the fact that each operation may have several inputs. To solve this problem, the framework employs an ordering algorithm that detects potentially illegal connection attempts. This algorithm relies on the fact that for any group of interconnected operations that contain no co-dependencies, it is possible to find an ordering such that any given operation can carry out its processing in full using only the outputs of those operations that precede it by order. Conversely, if there are co-dependencies then no such ordering will exist. When a new connection is made, if the output end of the connection precedes the input, the connection succeeds immediately. Otherwise The framework attempts to re-order the operations to make the new connection legal. If the re-ordering cannot be done, then the new connection fails.

Because no connection may cross the boundary of an enclosing `Meta`, only the member `Operator`s of the parent `Meta` need to be re-ordered. Further, a connection only needs to be checked if it is between two member `Operator`s, not if it involves the parent `Meta` itself (refer to the connection scenario labelled **a** in figure 3.40 in section 3.4.8.1).

3.5.7 Summary

This section has discussed the framework's use of threads for scheduling control of a program among its various `Primitive` operations. The basis of the framework's scheduling approach is to allocate a thread to each `Primitive` operation. In this way, each `Primitive` may run independently of all the others. Synchronisation between dependent `Primitive`s is achieved through its `block()` and `unblock()` methods, which put a `Primitive`'s thread to sleep and wake it up again respectively.

A `Primitive`'s thread iterates its main loop continuously until it is killed. This loop's primary purpose is to call the `Primitive`'s `implementation()` method; upon successful completion of this method, the `Primitive` achieves its *steady state*. In this state, a `Primitive` simply sleeps until its processing needs to be restarted due to one of its inputs being invalidated. Through the use of exceptions, a `Primitive` may also restart its main loop part way through its processing; this is necessary if one of its inputs is invalidated while it is processing that input's data.

To avoid over-allocation of Java's thread resources, a `Primitive` may be configured without its own thread, in which case it relies on a thread from a shared thread pool instead. The framework distinguishes between `Primitive`s that have their own thread and those that don't by referring to them as *heavyweight* and *lightweight* respectively. Every program created using the framework has access to an instance of the framework's `ThreadPool` class, which allocates a thread to a lightweight `Primitive` whenever the data at each of its inputs becomes completely available. A `Primitive` running in a shared thread may not block; if it tries to, or if one of its inputs is invalidated during processing, it aborts and must be reallocated another shared thread at a later time.

The framework's `Workspace` class subclasses `Meta` and provides a root parent for all `Operator`s in a program. `Workspace` also aggregates a program's `ThreadPool`, and provides methods to suspend and resume a program. Finally, by detecting and avoiding situations where `Meta`s are nested, or where the operations in a program are co-dependent, the framework avoids scheduling lockouts.

3.6 Applet Support

The framework's capabilities are made more accessible for applet development by specialising and customising `Applet`, Java's base applet class, and providing the means to display a program's results to the user within an applet's user-interface.

3.6.1 Hiperlet

The basis of the framework's support for applet development is its `Hiperlet` class (see figure 3.52). `Hiperlet`'s main purpose is to aggregate an instance of `Workspace`, to which all of the operations needed by the applet will be added. This attribute may be accessed by its `root()` method. The `connect()` and `disconnect()` methods are intended to result in a more readable programming style when establishing the required connections between the applet's operations.

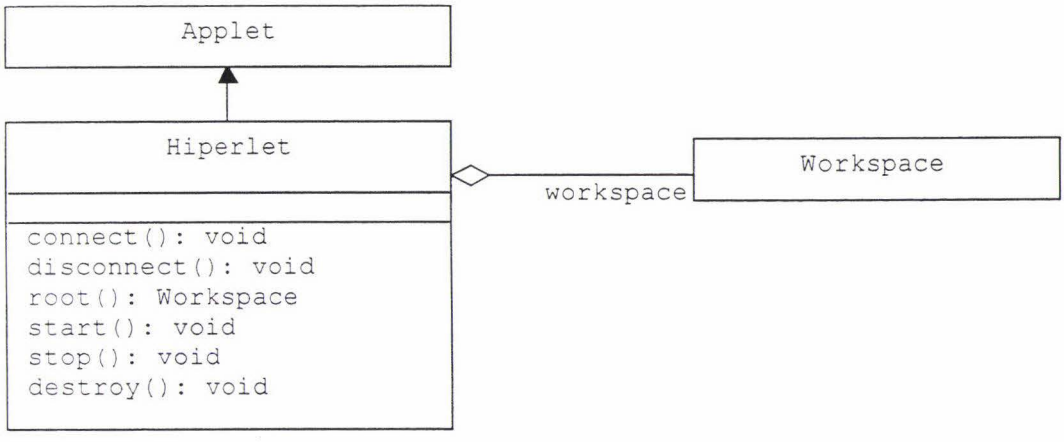

Fig 3.52: *The `Hiperlet` class*

The methods `start()`, `stop()` and `destroy()` are inherited from `Applet`. These are `Applet`'s life-cycle methods, which are invoked by the Web-browser that is running the applet at the appropriate times. If the user leaves the Web-page containing the applet (causing `stop()` to be invoked), the browser may decide to keep it in memory rather than destroying it, just in case the user intends to return to that page in the near future (at which time `start()` is invoked). `Hiperlet`'s `stop()` and `start()` methods call its `Workspace`'s `pause()` and `play()` methods respectively. In this way, the browser will not waste processing resources on the applet when the web-page that contains it is not being viewed. In the event that the browser does `destroy()` the applet, `Hiperlet` invokes its `Workspace`'s

`kill()` method, which in turn kills all of its member operations. The use of the `Hiperlet` class allows the developer to safely ignore these details.

Developers subclassing `Hiperlet` to create a new applet should:

- Design and implement the applet's user-interface. The developer may rely on a visual editing tool (such as Borland JBuilder) to assist in laying out this user-interface.
- Implement the new applet's constructor. This should first call `Hiperlet`'s constructor and pass in the program name and the number of shared threads to be used by its thread pool. The `hiperInit()` method should then be called.
- Implement `hiperInit()`. This method should instantiate the required operations and establish the necessary connections.

The framework ensures each operation will start running as soon as it is created. Hence, once `hiperInit()` returns, the applet will be fully operational.

3.6.2 *Display*

The framework provides a `Display` class that the programmer can subclass to implement operations for displaying different types of data. This class provides a blank 'canvas' on which anything can be drawn. For instance, an operation for displaying an image would subclass `Display` and add an input through which its image data could be acquired. It would then render this image to its canvas. Figure 3.53 below illustrates the structure of the `Display` class.

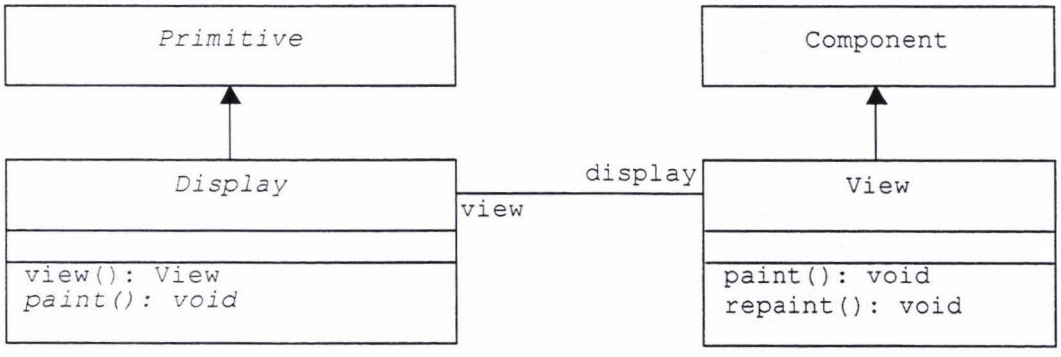

Fig 3.53: *The `Display` class*

The framework's `View` class, which is subclassed from Java's `Component` class, provides `Display` with its canvas. Because it is a `Component`, an instance of `View` may form a visible part of an applet's user-interface. `View` may also be manipulated by visual editing tools such as Borland JBuilder because it qualifies as a Java Bean. This greatly assists the developer in laying out an applet's user-interface in terms of where a program's results will be displayed on the screen.

The association between an instance of `View` and an instance of `Display` may be set up at `Display`'s construction time. This is achieved by passing an instance of `View` (which has been previously added to an applet's user-interface) to `Display`'s constructor, as a parameter. Once this is done, `View` will pass on all of its paint messages (which it receives from Java's runtime whenever it needs to be re-painted) to its associated `Display` instance by calling `Display`'s own `paint()` method. Hence, the programmer may implement `Display`'s `paint()` method to define how its associated `View` should render its viewable contents.

Subclasses of `Display` should also implement `Primitive`'s `implementation()` method. This method should acquire the operation's input data and prepare it for rendering. For instance, if an image is being displayed, then a renderable image should be created from this image's pixel array (this is done using Java's `Image` class). The `repaint()` method on this `Display`'s associated `View` should then be called, which tells Java's runtime to invoke the `View`'s `paint()` method as soon as possible.

3.7 Additional Features

This section covers a number of additional features provided by the framework that are better described in terms of the design decomposition provided by the previous sections, rather than within it.

3.7.1 IllegalActionException

When using the framework to develop a program, the developer may do something that the framework considers *illegal*. Examples are: establishing a connection that would cause a co-dependency; or, instantiating an operation without specifying its parent `Meta`. Such actions cannot be detected at compile time, and so result in an `IllegalActionException` being thrown at run-time.

`Operator`'s `connect()` and `disconnect()` methods both throw this exception if they are called upon to perform an illegal action. By catching such exceptions, developers can easily determine where the problems are with their programs, and the nature of those problems.

3.7.2 Looping Constructs

Many image processing tasks rely on looping constructs (for-loops, while-loops, etc.) to perform iterative operations. Unfortunately, these are not easily represented using a connection-oriented, data-flow style of programming. Further, the framework does not allow explicit loops to be created, as these would involve co-dependent operations. The framework's solution to allowing data to be passed 'upstream' is its `Construct` class, which subclasses `Meta`. The member operations of a `Construct` become the operations that are iterated. Figure 3.54 illustrates the concept underlying `Construct`.

Conceptually, the framework's approach is to allow data to flow upstream by passing it back within the 'walls' of a `Construct`. This is accomplished by modelling a `Construct`'s inputs and outputs as *switches*. In figure 3.54, the top pair of switches are in a *looping* state; in this state, data is fed internally back around in order to begin the next iteration of the loop. Since the data will be invalidated for each iteration, the `Construct` must buffer any data being fed back.

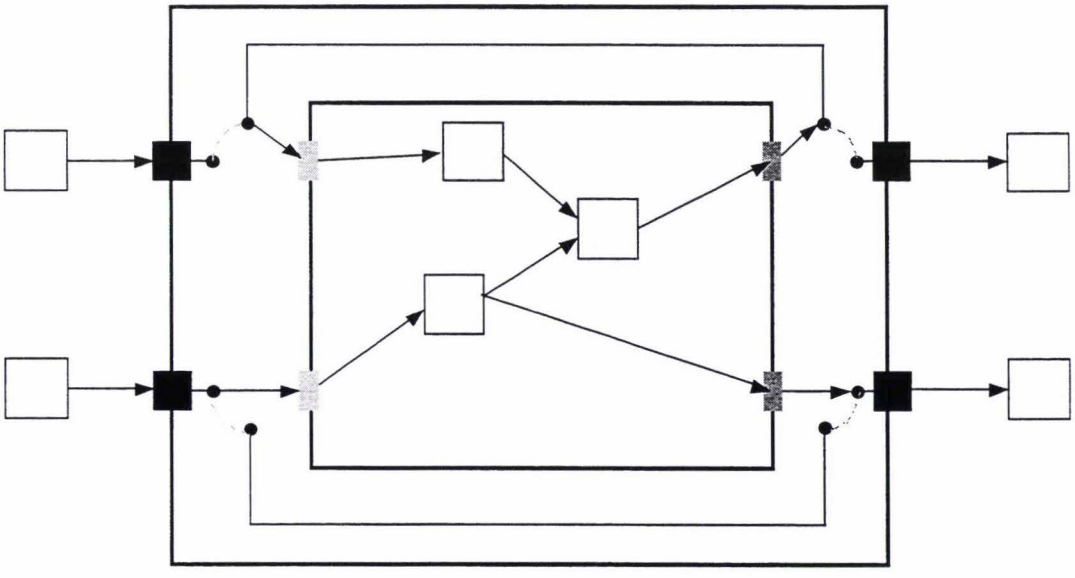

***Fig 3.54:** A Construct*

Conversely, the bottom pair are in a *feedthru* state; in this state, the switches are equivalent to a pair of `FeedThru`s. This state is used to represent either the initial (input) or final (output) iteration of a loop. Theses switches are *switched* to their appropriate states depending on the loop condition. This may be an internal counter or it may be a boolean condition supplied by an input which the `Construct` supplies for that purpose.

The framework's binding relationship provides a convenient switching mechanism. Figures 3.55a and 3.55b below demonstrate how an `Input` and two `Output`s are used together to provide both *looping* and *feedthru* behaviour for a `Construct`'s input. The framework's `SwitchIn` class aggregates such an arrangement of objects and implements its switching behaviour by simply re-binding its `Input`s and `Output`s as appropriate. Similarly, a `SwitchOut` class aggregates two `Output`s and an `Input` in order to provide the equivalent functionality for a `Construct`'s output. Both switch classes implement the framework's `Port` interface since they represent connection points on the `Construct`.

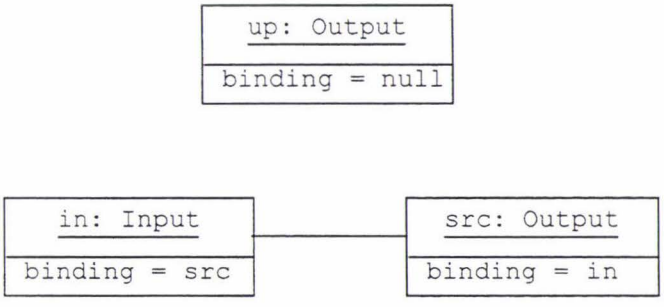

Fig 3.55a: *Providing feedthru behaviour at a `Constuct`'s input*

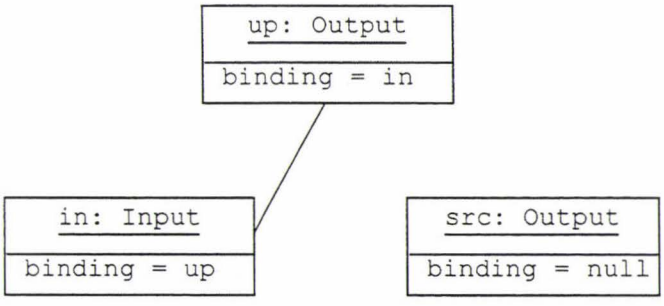

Fig 3.55b: *Providing looping behaviour at a `Constuct`'s input*

3.7.3 Macro

Occasionally, the developer will want to specify some processing that is better suited to a process-oriented representation. One common example of this is evaluating a complex function that depends on one or more `Scalar` input variables. This could be achieved by implementing a new `Primitive` operation to carry out the required calculation, however one of the purposes of the framework was to remove this level of complexity from the algorithm developer. An alternative approach is to make available the many elementary `Primitive`s that would be required to build up the function (see figure 3.56). However this solution becomes very cumbersome for functions that are even moderately complex, both in terms of development effort and the framework's own scheduling overhead.

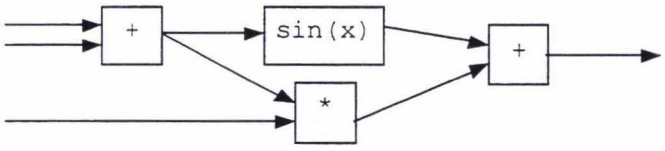

Fig 3.56: *Building up a complex function from elementary* `Primitive` *operations*

The framework overcomes this problem through its `Macro` operation. A `Macro` is a `Primitive` that can effectively be customised to perform almost any kind of processing using a custom *macro language*. This macro language can be used to write scripts that process scalars, vectors, or lists of scalars and vectors. It is not intended to manipulate large data structures such as images, but provides a means to quickly and efficiently define a processing task that would otherwise be cumbersome in data-oriented form.

To use a `Macro` operation, the developer creates a new `Macro` instance and passes in a text string containing the desired script. The `Macro` operation then compiles this script, thereby determining what inputs and outputs are required. The necessary inputs and outputs are then created. When writing a new script, the developer may specify an input variable by preceding its name with a '`@`' character, and an output variable by preceding its name with a '`#`' character. Thus, a new `Macro` operation created using the string:

```
#out = @in * 2;
```

would cause a `Primitive` operation to created with one input and one output. Passing a `Scalar` value into the input of this operation would result in its output providing that value multiplied by two. In addition to the standard arithmetic operators, the developer may use a range of standard functions (sin, cos, etc.) and may use parentheses to override operator precedence. Also, the programmer may implement custom functions for use in developers' scripts, thereby making the language extensible.

A particularly useful aspect of the framework's macro language is its support for conditional operators. These are provided in the form of short-circuit boolean operators, and a ternary operator. These operators allow alternative portions of a script to be evaluated depending on its inputs. This provides a convenient solution for developing programs that require conditional behaviour. In general, this is also something that is not easy to represent efficiently within a data-oriented programming environment.

These conditional processing capabilities can be used to implement conditional processing of images (or other larger data types). The simple example script given below demonstrates this principle:

```
#im_out = (@cond > 0) ? @im_in1 : @im_in2;
```

Here, two images are being input, `@im_in1` and `@im_in2`. There is also another input, `@cond`, which is intended to accept a scalar type. The single output, `@im_out`, then depends on the value that is input to `@cond`. If this value is positive, `@im_in1` is output, otherwise `im_in2` is output. More complex cases are possible, for instance providing a list of images at an input and selecting an appropriate image from this list to output.

4. Application Example

The purpose of the framework developed in the previous chapter was to facilitate the creation of Java applets for interactively demonstrating image processing concepts. The efficacy of the framework for this purpose is illustrated in this chapter through the design and development an applet for image thresholding.

4.1 Concept

A global threshold operation is a point operation that produces a binary output image from an input greyscale image. The second input (or parameter) is a threshold level that specifies the pixel intensity at which to threshold the input image. Each pixel in the input image that has a pixel value greater than or equal to the threshold level results in the corresponding pixel in the output image being set to white. Similarly, each pixel less than the threshold level produces a corresponding black pixel in the output image.

Thresholding is commonly used to separate objects in an image foreground from the background. Given an input image that has been carefully acquired (i.e. proper lighting, focus) it is often possible to use a threshold operation directly to produce an image with a black background, and with all of its foreground objects white. This facilitates many subsequent processing algorithms, such as counting the number of foreground objects in an image or performing shape analysis on those objects.

The success of a threshold operation on any image is largely dependent on the threshold level that is chosen. In turn, the best choice for this level is dependent on the image itself. While there are systematic methods for selecting a threshold level (Russ, 1994), frequently a threshold level will be found manually by adjusting the threshold level until the desired outcome is achieved. For each threshold level that is tried, the result is inspected 'by eye' until the chosen level produces a result that looks best to a human observer.

A practical understanding of the threshold operation can also be conveyed in a similar manner by presenting the user with an image, and inviting them to select a good threshold. By presenting a selection of images the power and also limitations of the threshold operation may be demonstrated. In section 2.3 a data flow representation for the demonstration of a threshold operation was presented. This will be used here as the basis of a design that will be implemented using the framework.

4.2 Design

Figures 4.1 and 4.2 below reproduce (in slightly altered form) the diagrams used earlier to represent the design of a possible demonstration applet. Figure 4.1 illustrates how the applet developer plans to connect together a number of operations to provide the functionality for the demonstration. The input image being threshold is provided by a load image operation, and the threshold level comes from a slider. At the right, two display image operations display the original and processed images, allowing a comparison to be made between the two. This figure also shows the names that have been assigned to each of the inputs and outputs of the operations that will be used to construct this demonstration.

Figure 4.2 illustrates how the applet's user-interface is intended to appear. Diagrams such as the two above can be used by applet developers to specify the designs for any image processing concept they wish to demonstrate. The two components to the design are the dataflow between the image processing operations, and the view seen by the user of the user interface components.

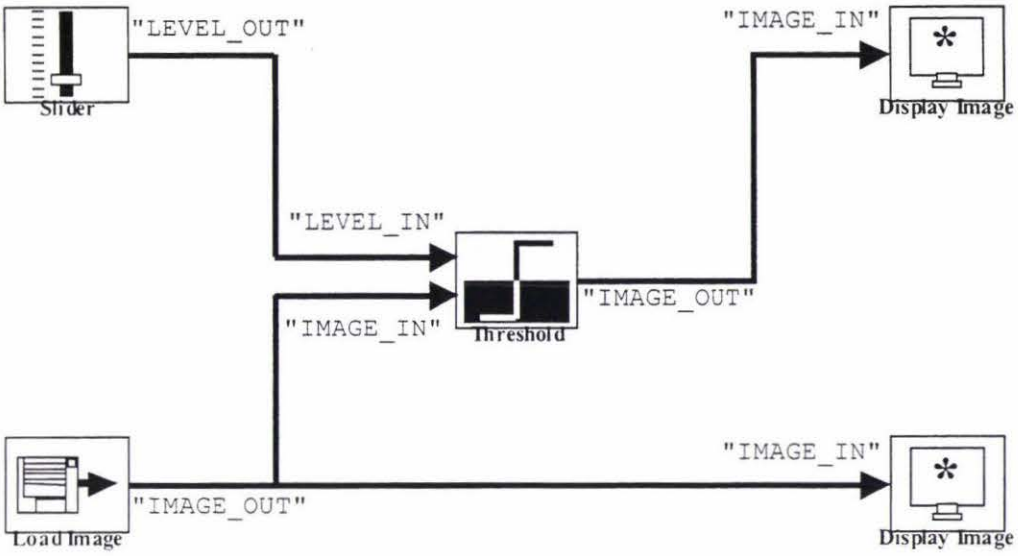

Fig 4.1: Interactively thresholding an image

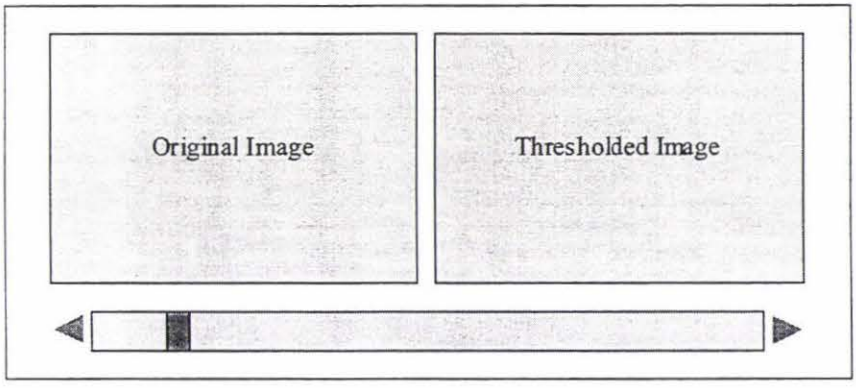

Fig 4.2: User-interface design for demonstrating a threshold operation

4.3 Implementation

This section describes how, using the services provided by the framework, an applet for interactively demonstrating a threshold operation can be developed. In this section, the roles of both the programmer and the developer are addressed. From the programmer's perspective, the examples show how the framework's Data and Operation APIs are used to implement the required operations. The developer, on the other hand, assumes these operations are available and ready for use. He or she then constructs the desired functionality for the applet using the framework's Developer API.

4.3.1 Threshold

The threshold operation is an example of a `Primitive` operation. The programmer must subclass `Primitive` and implement its constructor (where its inputs and outputs are created) and also its `implementation()` method (where the thresholding algorithm itself may be implemented). The following code segment creates the required subclass, naming it `Threshold`. It also declares the variables that will be used to create `Threshold`'s inputs and outputs. Note that there is one set of declarations for each input or output required.

```
// Threshold operation
public class Threshold extends Primitive {

  // Input LEVEL_IN
  private InputManager  LEVEL_IN;
  private String        LEVEL_INn = "LEVEL_IN";
  private int[]         LEVEL_INc = { SCALAR };
  private Scalar        LEVEL_INd = new Scalar(128);

  // Input IMAGE_IN
  private InputManager  IMAGE_IN;
  private String        IMAGE_INn = "IMAGE_IN";
  private int[]         IMAGE_INc = { BYTEIMAGE };

  // Output IMAGE_OUT
  private OutputManager IMAGE_OUT;
  private String        IMAGE_OUTn = "IMAGE_OUT";
  private ByteImage     IMAGE_OUTd = new ByteImage();

  ...
```

This code illustrates a convenient and useful idiom for specifying information about the inputs and outputs that are to be created for an operation. The name of the input or output is used to reference the corresponding `InputManager` or `OutputManager`. This name is then suffixed with different characters in order to declare various additional pieces of information the pertain to it. For example, its name as a string (suffix 'n'), an input's compatibles types (suffix 'c'), an input's default value (suffix 'd'), or the instance of `Data` that an output is to manage (also suffix 'd'). In this case, `Threshold`'s level input is given a default value of 128, which represents 'mid-grey'. The following gives `Threshold`'s constructor implementation, which uses these declarations to create its inputs and outputs.

```
  ...

  // construct Threshold operation
  public Threshold(String name,   // developer can specify name
                   Meta parent)   // and parent
```

```
                    throws UserActionException {

    // call Primitive's constructor
    super(name, parent);

    // create inputs
    LEVEL_IN = addInput(LEVEL_INn, LEVEL_INc, LEVEL_INd);
    IMAGE_IN = addInput(IMAGE_INn, IMAGE_INc);

    // create outputs
    IMAGE_OUT = addOutput(IMAGE_OUTn, IMAGE_OUTd);

    // start operation in its own thread
    start(HEAVY);
  }

  ...
```

`Threshold`'s constructor creates its inputs and outputs via calls to `addInput()` and `addOutput()`, passing in the information that was declared above. In this way, a `Threshold` operation's ports will automatically be constructed whenever it is instantiated. `Threshold`'s constructor also assigns this operation its own thread, so that it will be able to process its inputs as soon as they become available.

The following lists `Threshold`'s `implementation()` method in full. Before any processing is performed, its inputs are acquired and its outputs are initialised. Note that the size of `Threshold`'s output image is set to the size of its input image.

```
  ...

  public void implementation() throws ControlFlowException {

    // acquire threshold level
    Scalar level = (Scalar)LEVEL_IN.acquireInput();
    LEVEL_IN.waitForComplete();
    int v = level.asInteger();

    // acquire input image to threshold
    ByteImage imgIn = (ByteImage)IMAGE_IN.acquireInput();
    int width = imgIn.getWidth();
    int height = imgIn.getHeight();
    byte[] pixIn = imgIn.getPixels();

    // initialise output
    IMAGE_OUT.invalidate();
    IMAGE_OUTd.setDimensions(width, height);
    IMAGE_OUT.validate();
    byte[] pixOut = IMAGE_OUTd.getPixels();

    // do threshold
    for (int y=0, index=0; y < height; y++) {
```

```
        // get next row of input image
        IMAGE_IN.getAvailable(y);

        // process row
        for (int x=0; x < width; x++, index++)
          if (pixIn[index] >= (v-128))
            pixOut[index] = 127;
          else
            pixOut[index] = -128;

        // make processed row available at output
        IMAGE_OUT.newData(1);
      }

      // finalise output
      IMAGE_OUT.dataComplete();
    }
  }
```

Once the I/O setup steps are complete, this method executes a double for-loop that iterates through every pixel, comparing each with the threshold level and producing the appropriate output. At the start of each row, a call to `getAvailable()` causes processing to pause until the next row of the input image becomes available. Once each row has been processed, a call to `newData()` makes the processed row available at the output. Finally, a call to `dataComplete()` finalises the output image.

In this implementation, pixels are accessed directly from their arrays for both the input and output images. Because it is assumed that the input threshold level is in the range 0 to 255, this creates additional complexity in that an offset (subtracting 128) must be applied for each comparison operation. This complexity could be hidden by implementing *get* and *set* pixel methods in the `ByteImage` class (an appendix to this thesis discusses the pros and cons of this and other alternatives with respect to pixel access).

This is all that is required to implement the `Threshold` operation. The `ByteImage` and `Scalar` data types are assumed to have already been implemented; these are as they were described in the previous chapter.

4.3.2 Control

In order to implement the slider control, a `Control` class will be created. This class implements Java's `AdjustmentListener` interface, making it capable of responding to any user-interface element that generates adjustment events. This

could be a slider, or it could be a dial or any other user-interface element capable of being 'adjusted'. Hence this class has greater potential for re-use than if it was purely a 'slider control' class.

The `Control` class is subclassed from `Primitive`, and begins as follows, using the same style as before.

```
public class Control extends Primitive
  implements
    AdjustmentListener {

  // Output VALUE_OUT
  private OutputManager VALUE_OUT;
  private String        VALUE_OUTn = "VALUE_OUT";
  private Scalar        VALUE_OUTd = new Scalar();


  public Control(String name,
                 Meta parent,
                 int initial)
                 throws UserActionException {

    super(name, parent);

    VALUE_OUT = addOutput(VALUE_OUTn, VALUE_OUTd);

    fire(initial);
  }

  ...
```

Note that this constructor does not call `start()`. This is because it is not driven by its own thread, nor is it driven by its inputs (because it has none). Rather, it is driven by Java's event handler, which calls `AdjustmentListener`'s `adjustmentValueChanged()` method whenever the user-interface element associated with it is adjusted by the user. This in turn updates its output, as demonstrated by the rest of `Control`'s implementation below. Instead, `Control`'s constructor has an additional parameter named `initial`. This integer value is passed to `fire()` just before the constructor exits. This provides a way for the developer to specify a `Control`'s initial value at construction time.

```
  ...

  public void implementation() throws ControlFlowException {}

  private synchronized void fire(int v) {
    VALUE_OUT.invalidate();
    VALUE_OUTd.setValue(transform(v));
    VALUE_OUT.validate();
    VALUE_OUT.dataComplete();
```

```
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
      fire(e.getValue());
    }

    protected double transform(int v) { return v; }
  }
```

`Control`'s `implementation()` method does nothing because it never needs to be scheduled in the usual sense. When `adjustmentValueChanged()` is invoked, the new value from its the user-interface element is passed to `fire()`. This then notifies `Control`'s output of its new value using the standard Data API calls.

Note that this value is passed via a call to `transform()`. Programmers or developers can override this method (by subclassing `Control`) to perform some transformation on the value before it is output. For example, integer values from a slider in the range 0 to 100 could be linearly transformed to the real-valued range -0.5 to +0.5. This feature provides an additional level of customisation to this class, enhancing reuse.

4.3.3 *DisplayImage*

To display an image, the programmer must subclass the framework's `Display` class. For the purposes of this chapter, this class will only handle input data of type `ByteImage`. To display other image types, the programmer may modify this class to detect the incoming type and act accordingly, or may create multiple `Display` subclasses, each of which handles a different image type.

```
public class DisplayImage extends Display {

  // Input IMAGE_IN
  private InputManager IMAGE_IN;
  private String       IMAGE_INn = "IMAGE_IN";
  private int[]        IMAGE_INc = { BYTEIMAGE };

  // Java AWT image display stuff
  private IndexColorModel cm;
  private MemoryImageSource mis;
  private Image img;

  ...
```

The additional declarations are Java Abstract Windowing Toolkit (AWT) classes needed to actually draw an image on the screen; they are used in `DisplayImage`'s

`implementation()` method. `DisplayImage`'s constructor is implemented as follows.

```
...

public DisplayImage(String name,
                    Meta parent,
                    View view)
                    throws UserActionException {

  super(name, parent, view);

  IMAGE_IN = addInput(IMAGE_INn, IMAGE_INc);

  cm = new IndexColorModel(8, 256,
                           LUT.ramp(),   // red component
                           LUT.ramp(),   // green component
                           LUT.ramp());  // blue component

  start(HEAVY);
}

...
```

Here, `DisplayImage`'s input is created, its `IndexColorModel` attribute is instantiated, and it is started with its own thread. `LUT.ramp()` creates a linear ramp from black to white so that the image appears in grey scale to the user. The remainder of `DisplayImage`'s implementation consists of its `paint()` and `implementation()` methods.

```
...

public void paint(Graphics g) {
  if (img != null) g.drawImage(img, 0, 0, null);
}

public void implementation() throws ControlFlowException {

  // acquire input image
  ByteImage imgIn = (ByteImage)IMAGE_IN.acquireInput();
  int width = imgIn.getWidth();
  int height = imgIn.getHeight();
  byte[] pixels = imgIn.getPixels();

  // allocate new MemoryImageSource
  mis = new MemoryImageSource(width, height,
                              cm, pixels,
                              0, width);
  // create image
  img = view().createImage(mis);

  // draw image
  int y=0, h=0;

  while (!IMAGE_IN.dataConsumed()) {
```

```
        y = IMAGE_IN.getAvailable();

        mis.newPixels(0, h, width, y-h);
        view().repaint(0, h, width, y-h);

        h = y;
      }
    }
  }
```

`DisplayImage`'s `paint()` method simply draws the image that has been created using the graphics context that is passed to it. The `implementation()` method creates this image using an instance of `MemoryImageSource`, which it updates each time a new set of rows becomes available from its input. Each update is accomplished by calling `newPixels()` on the `MemoryImageSource` instance, and then repainting `DisplayImage`'s `View` via a call to its `repaint()` method.

4.3.4 ThreshDemo

To create the demonstration applet, the developer must subclass `Hiperlet`, instantiate the required operations, and then connect them together. For the purposes of this chapter, it is assumed that a visual editor (for example, Borland JBuilder) has been used to layout the applet's user-interface. In this case, the developer would lay out two `View` beans and a single `Scrollbar`, following the plan in figure 4.2 above. `Scrollbar` is a Java AWT widget that is used in this example as a slider. Using a visual editor, almost all of the code below may be automatically generated. Only two lines need to be added manually by the developer: the call to `Hiperlet`'s constructor, and the call to `hiperInit();`

```
public class ThreshDemo extends Hiperlet {

  // user interface elements
  Scrollbar scrollbar = new Scrollbar();
  View left = new View();
  View right = new View();
  XYLayout xYLayout1 = new XYLayout();

  // construct ThreshDemo
  public ThreshDemo() throws UserActionException {

    // call Hiperlet's constructor, use zero shared threads
    super("THRESHDEMO", 0);

    // call initialisation methods
    try {
      jbInit();       // initialise user-interface
      hiperInit();    // initialise framework components
    }
```

```
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    scrollbar.setMaximum(255);
    scrollbar.setMinimum(0);
    scrollbar.setOrientation(0);
    scrollbar.setBlockIncrement(10);
    scrollbar.setValue(128);
    scrollbar.setVisibleAmount(1);
    this.setLayout(xYLayout1);
    xYLayout1.setHeight(600);
    xYLayout1.setWidth(650);
    this.add(right, new XYConstraints(337, 17, -1, -1));
    this.add(left, new XYConstraints(13, 17, -1, -1));
    this.add(scrollbar, new XYConstraints(21, 461, 604, 21));
  }

  ...
```

Once this has been done, the developer need only implement `hiperInit()` (it is assumed here that a load image operation has been implemented and is available). The code for `ThreshDemo`'s `hiperInit()` method is given below.

```
  ...

  private void hiperInit() throws Exception {

    // instantiate operations
    Threshold    thresh = new Threshold("THRESH", root());
    LoadImage    load   = new LoadImage("LOAD", root(),
                                        "testimage.jpg");
    Control      slider = new Control("SLIDER", root(), 128);
    DisplayImage disp1  = new DisplayImage("DISP_1", root(),
                                           left);
    DisplayImage disp2  = new DisplayImage("DISP_2", root(),
                                           right);

    // associate scrollbar with Control
    scrollbar.addAdjustmentListener(slider);

    // establish connections
    connect(slider, "VALUE_OUT", thresh, "LEVEL_IN");
    connect(load,   "IMAGE_OUT", thresh, "IMAGE_IN");
    connect(load,   "IMAGE_OUT", disp1,  "IMAGE_IN");
    connect(thresh, "IMAGE_OUT", disp2,  "IMAGE_IN");
  }
}
```

These ten lines of code represent the bulk of the effort required by the developer to actually create this demonstration applet. Figures 4.3a and 4.3b provide screenshots of the applet taken while running it under Microsoft Internet Explorer 5 for Windows.

Fig 4.3a: *Screenshot of threshold demonstration applet (threshold = 75)*

Fig 4.3b: *Screenshot of threshold demonstration applet (threshold = 155)*

This example is a minimal implementation only. With a little further effort, the user-interface could be embellished by labelling where appropriate, for instance with a scale for the slider control. This demonstration might also benefit by displaying a histogram of the original image below the slider. This would reveal the locations of the image's various intensity peaks, thereby further assisting the user's understanding of the choice of threshold level.

4.4 Conclusions

This chapter has made an example of the use of the framework's APIs to implement a simple demonstration applet. The programmer has the responsibility for implementing the various operations that are required by the developer. Once this is done, they may be documented and distributed to other developers using the framework.

The applet developer has very little work to do once the necessary operations are made available. Using a visual development tool such as Borland JBuilder, an applet's user-interface can be constructed very easily. The developer then need only instantiate the required operations and establish the necessary connections between them to create the finished applet.

5. Summary and Conclusions

This thesis has dealt with the design and development of a Java framework for creating image processing programs. The initial motivation for this research was to provide interactive elements for teaching image processing electronically. Chapter one identified the World Wide Web as a popular and attractive delivery medium for such electronic instruction, and discussed the potential of Java applets to provide interactive demonstrations of image processing concepts via Web-browsing software. In this way, practical interactive and theoretical textual instructional content could be made accessible via an integrated user-interface.

At the close of the first chapter, the use of a *framework* was proposed in order to significantly reduce the development burden associated with creating such applets. This framework would provide a layer of abstraction, thereby allowing the developer to draw upon its services rather than implement every new applet from scratch. The framework would embody much of the functionality that is common to each applet. As well as reduced development effort, the advantages of using a framework are simpler designs, more maintainable code, and fewer bugs.

Chapter two expanded on the requirements of the proposed framework. This chapter examined traditional approaches to providing interactive content for electronic image processing instruction. Interactive algorithm development software was identified as being well suited to this purpose. However, it was found that traditionally, instructional systems utilised such software as a separate

component for providing their interactive content, and a number of disadvantages of this approach were identified:

- The resulting learning environment is not seamless.
- There is a need to acquire and install the necessary software.
- Users may not follow instructions properly for using the software.
- User-interfaces for this category of software are quite general.

Despite these disadvantages, it was recognised that certain design aspects of algorithm development software could be drawn upon in the design of the framework. A subsequent analysis identified four essential components that need to be provided by the framework, along with several application programming interfaces (APIs) that allow these components to work together. These APIs are also the means by which programmers and developers gain access to the framework's services. The components identified were:

- An extensible operation library. New operations may be defined in terms of the framework's Operation API.
- Data types. The framework's Data API allows commonly used data types in image processing to be defined and utilised by the framework's operations. These also need to be extensible.
- A supervisory system. This was identified as being responsible for passing data between operations, and scheduling them for execution. The Data and Operation APIs provide this functionality.
- Applet support. This component facilitates the development of applets that incorporate the framework's capabilities. The Developer API is used to provide support for applet development using the framework.

Figure 2.2 gave an idealised representation of the framework that placed these components and APIs in the context of the framework as a whole, and illustrated their relationships to one another.

The final contribution of this second chapter was a dataflow representation for programs built using the framework. A dataflow model was chosen to give a connection paradigm that allowed programs to be constructed purely in terms of interconnected networks of image processing operations. By considering any unit of functionality, including user-interface elements and file I/O operations, to fall within this paradigm, a uniform scheme was arrived at for representing image processing programs using the framework.

A flow based design also capitalises on the data flow model's naturally visual characteristics. These allow data flow oriented development environments to provide intuitive and user-oriented tools for specifying and manipulating image processing algorithms. It was felt that the adoption of a data oriented model would translate into similar benefits for the user of the framework, through the framework's Developer API.

The design and development of the framework was broken in to five separate areas: *Operations*, *Data Types*, *Data Passing*, *Scheduling* and *Applet Support*. These provide a high-level decomposition for the framework's design. Chapter three covers this design by describing and explaining the framework's various class hierarchies, APIs and mechanisms.

The framework's design may be summarised as a combination of the *Composite* and *Observer* design patterns (Gamma, *et al.*, 1995). The *Composite* pattern allows groups of operations to be treated as though they are a single operations. This provides a convenient mechanism for managing the operations in a program that does not violate the connection paradigm on which the design is based. In particular, the framework's `Workspace` class provides a root-level repository for keeping track of a program's operations.

The *Observer* pattern allows an operation's outputs to connect to another operation's inputs on a one-to-many basis. This allows a single output to supply multiple inputs, if necessary. The framework's data passing mechanisms augment the Observer pattern to allow information on a data's availability to be passed *progressively* between an output and the inputs it is connected to. This allows an operation to begin processing before all of the input data is available, and to abort

its processing in a timely manner if one or more of its inputs become invalid part way through processing.

The basis of the framework's scheduling approach is to allocate every operation its own thread. The framework also allows a variation on this that conserves thread resources by providing a shared thread pool for lightweight operations. The framework's blocking mechanism automatically synchronises dependent operations. The scheduling mechanisms are simplified by avoiding loops that cause the outputs of any operation to eventually depend upon themselves. This is achieved by enforcing a strict ordering relation among interconnected operations, thereby preventing mutual dependencies.

To assist the development of applets using the framework, a `Hiperlet` class is provided that subclasses Java's `Applet` class and incorporates a `Workspace`. This provides a clean slate, or platform, on which applets with the desired functionality can be built. Chapter four of this thesis demonstrates the development of one such applet, wherein the roles of both the programmer and the developer are considered.

The application example that is discussed shows how an applet to demonstrate the concept of a threshold operation can be conceptualised, designed, and implemented using the framework. This example demonstrates that, given a well populated operation library, the effort required to implement a new applet on the part of the developer is relatively trivial. Hence, the design achieves its goal of providing an easy-to-use framework for significantly reducing the development effort involved in creating applets for demonstrating image processing concepts.

5.1 Future Work

This final section discusses several areas in which the framework may be developed further. The first area is aimed at further reducing the effort required by the developer to develop applets using the framework. Currently, the developer must still write a small amount of Java code in order to implement a new applet. This code declares the operations that are used by the applet, and connects them together to give the desired dataflow. By providing a visual editing tool for specifying the operations to be used by an applet and the connections

between them, this code could be generated automatically, thereby freeing the developer from having to write any code at all. This would enable a developer to construct applets without having to learn Java.

There are at least two potential ways that such a tool could be provided. One alternative is a stand-alone tool (possibly developed in Java). However this approach has a drawback if the developer also uses a visual app-builder tool (such as Borland JBuilder) to develop the applet's user-interface. This would involve exporting the code generated by the tool into JBuilder (or vice versa), leading to a cumbersome overall development environment. A better alternative would therefore be to use JBuilder (or similar) as the visual editing tool for building the applet's user-interface *and* for specifying the applet's functionality. This could be achieved using Java Beans to represent an applet's operations in the form of non-visual components (i.e. those for which code needs to be generated but do not appear on screen, as do normal user-interface elements). Such an approach would require some additional research into the best way of mapping the framework's connection paradigm to JBuilder's Java Bean and code generation support.

It may be that removing the need for the developer to write Java code altogether is unrealistic. If this were the case, it might be sufficient to supply the developer with the support tools necessary rather than to automate the process totally. This could be done, for instance, by auto-generating just the skeleton of an applet (or as much as possible) given a visual specification provided by the developer. This would reduce the coding effort at least in part.

The second area of potential future work is the use of the framework as a starting point for the design of a fully-fledged interactive image processing algorithm development environment. In chapter two, it was noted that software in this category invariably relies on an implicit framework to underpin its design. The use of a framework in this capacity helps ensure a modular and well-structured design is achieved; most importantly, it ensures that this design is extensible.

Because the framework is data-oriented, it would be well suited to providing the basis for an algorithm development environment that employed a visual, data flow style of user-interface. This would involve creating a visual language editor that interfaced with the framework's APIs to provide the user with a number of tools

for specifying and manipulating algorithms. These tools would allow the user to manipulate (i.e. create, delete, connect, group, disconnect, etc.) the operations in an algorithm. In addition, this editor would need to display the current state of the user's work-in-progress in a graphical form.

In its present form, the framework is very close to providing all that is needed for this potential application. One feature that it would benefit from in this respect is the ability to group a number of operations together as a single `Meta` operation at runtime. Such an 'encapsulation' feature would be useful because it would allow users to partition their algorithms part way through development. Figures 5.1a and 5.1b illustrate how such a feature would work.

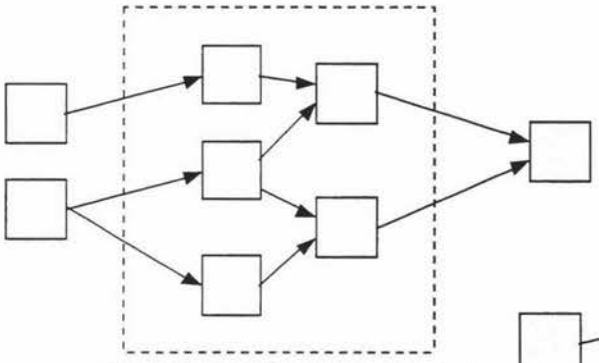

Fig 5.1a: *Before encapsulation*

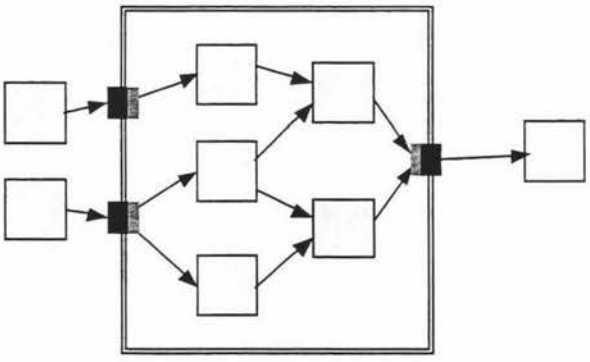

Fig 5.1b: *After encapsulation*

Note that this feature should preserve an algorithm's existing connections so that the algorithm remains functionally equivalent after the encapsulation has been performed. The reverse of this feature should also be provided so that `Meta` operations can be removed while leaving their members. One further feature required would be the ability to load and save a developed algorithm as a data file.

Thirdly, although the framework was initially developed with an image processing application in mind, it could in principle be used for developing *any* kind of data flow program. This is a positive corollary of the framework's general purpose connection paradigm and the general nature of its data passing and scheduling mechanisms. Effectively, this represents a third prospect for future development

using the framework. By implementing operations and data types for a particular application area using the appropriate framework APIs (for instance some alternative form of signal processing, or any algorithm that can be represented as a data flow) the framework could then be used to develop programs in that area.

A fourth potential area deals with making the framework more efficient in terms of its use of resources, in particular with respect to memory allocation. Dataflow languages do not lend themselves to in-place processing, so that even algorithms that follow a sequential flow will tend to allocate new memory for each operation in the sequence. The framework could be improved by enabling it to determine when an operation can safely throw away the memory it has allocated. The `Data` class' `allocate()` and `deallocate()` methods provide the hooks that allow the framework to manage the memory used by operations. These methods would have to be utilised through some scheme that determined the best time to allocate and deallocate memory, which would be dependent upon the algorithm implemented by a particular program.

A fifth area of future work is an improvement to the framework's method for determining whether or not a suitable `ConversionManager` exists in the event that an input is incompatible with the type of data is receiving. Currently, the information required to search through the available `ConversionManager`s is held statically by the `ConversionManager` superclass. While this is a convenient solution because it simplifies the search code, it is not ideal because this information needs to be updated whenever a new conversion manager is defined. This makes extending the framework in this respect an awkward task if the full source code is not on hand. A better solution would be to incorporate this information into subclasses of `Data`. Then, when a new data type was defined, the programmer would have the opportunity to specify the types it could be converted into. This would involve extending the framework's Data API for defining new types, and re-implementing `ConversionManager`'s `checkConvertible()` and `create()` methods. Fortunately, such a solution could be implemented without altering the way the rest of the framework interacts with this mechanism.

A final area of development is to build upon the error detection mechanism provided by the framework's `IllegalActionException` class. Currently, it is

the developer's responsibility to detect when this exception has been thrown. However, this basic mechanism could be extended to provide a more advanced, automated mechanism for informing developers of illegal actions performed by their programs. For example, the framework's applet support features could use `IllegalActionException` to generate pop-up dialog boxes when illegal actions are encountered. A feature such as this would further improve the framework's developer-friendliness.

References

Bailey, D. G. & Hodgson, R. M., *"VIPS – A Digital Image Processing Algorithm Development Environment"*, Image and Vision Computing, vol. 6 no. 3, p.176-84, August 1988.

Barker, P., *"Design Guidelines for Electronic Book Production"*, In: Nato ASI Series, vol. F76, Multimedia Interface Design in Education, Edwards, A. D. N. & Holland, S., (Eds.), ch.6, p.83-96, Springer-Verlag, Berlin, Heidelberg, 1992.

Booch, G., *"Object-Oriented Analysis and Design with Applications"*, 2nd Ed., Addison-Wesley, Menlo Park, California, USA, 1994.

Booch, G., Rumbaugh, J. & Jacobson, I., *"The Unified Modeling Language User Guide"*, Addison-Wesley, Massachusetts, USA, 1999.

Brusilovsky, P., Schwarz, E. & Weber, G., *"Electronic Textbooks on the World Wide Web: From Static Hypertext to Interactivity and Adaptivity"*, In: Web-Based Instruction, Khan, B. H., (Ed.), ch.30, p.255-62, Educational Technology Publications, Englewood Cliffs, New Jersey, USA, 1997.

Cady, F. M., Hodgson, R. M., Pairman, D., Rodgers, M. A. & Atkinson, G. J., *"Interactive Image-Processing Software for a Microcomputer"*, IEE Proceedings, vol. 128 part. E no. 4, p.165-70, July 1981.

Castleman, K. R., *"Digital Image Processing"*, Prentice-Hall, New Jersey, 1979.

Fisher, R. B. & Koryllos, K., *"Interactive Textbooks: Embedding Image Processing Operator Demonstrations in Text"*, International Journal of Pattern Recognition and Artificial Intelligence, vol. 12 no. 8, December 1998.

Fisher, R. B., Perkins, S., Walker, A. & Wolfart, E., *"HIPR: Hypermedia Image Processing Reference"*, CDROM published by John Wiley and Sons, Chichester, 1996.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison Wesley, Inc., USA, 1995.

Gonzalez, R. C. & Wintz, P., *"Digital Image Processing"*, 2nd Ed., Addison-Wesley, Massachusetts, USA, 1987.

Gosling, J., Joy, B. & Steele, G., *"The Java Language Specification"*, Addison-Wesley, Massachusetts, USA, 1996

Hanna, H., *"Development of Computer Assisted Learning to Assist in the Teaching of Image Processing and Image Coding"*, Proceedings of the IEEE 1st International Conference on Multimedia Engineering Education, p.387-91, July 6-8, 1994.

Harger, R. O., *"Teaching in a Computer Classroom with a Hyperlinked, Interactive Book"*, IEEE Transactions on Education, vol. 39 no. 3, p.327-35, August 1996.

Jain, A. K., *"Fundamentals of Digital Image Processing"*, Prentice Hall, Englewood Cliffs, New Jersey, 1989

Jensen, J. R., *"Educational Image Processing: An Overview"*, Photogrammetric Engineering and Remote Sensing, vol. 49 no. 8, p.1151-7, August 1983.

Jordán, R. & Lotufo, R., *"Interactive Digital Image Processing Course on the World Wide Web"*, Proceedings of the 1996 IEEE International Conference on Image Processing in Education (ICIP-96), vol. II, p.433-6, September 16-19, 1996.

Khan, B. H., *"Web-Based Instruction (WBI): What Is It and Why Is It?"*, In: Web-Based Instruction, Khan, B. H., (Ed.), ch.1, p.5-18, Educational Technology Publications, Englewood Cliffs, New Jersey, USA, 1997.

Lyon, D. A., *"Image Processing in Java"*, Prentice Hall, Inc., New Jersey, USA, p.39-47, 1999.

Mukundan, R., *"Binary Vision Algorithms in Java"*, Proceedings of the 1999 International Conference on Image and Vision Computing, p.145-50, August 30-31, 1999.

Ngan, P. M., *"The Development of a Visual Language for Image Processing Applications"*, Ph.D. Thesis, Computer Science, Massey University, Palmerston North, New Zealand, 1992.

Oliver, C. E., Strayer, M. R. & Umar, V. M., *"Building an Electronic Book on the Internet: CSEP – an Interdisciplinary Syllabus for Teaching Computational Science at the Graduate Level"*, Proceedings of the 1994 IEEE Frontiers in Education Conference (FIE '94), p.430-3, November 2-6, 1994.

Quatrani, T., *"Visual Modeling with Rational Rose and UML"*, Addison-Wesley, Massachusetts, USA, 1998.

Rasband, W., *"Image User's Manual"*, National Institute of Health, Maryland, 1992.

Rasure, J. R. & Williams, C. S., *"An Integrated Data Flow Visual Language and Software Development Environment for Image Processing"*, International Journal of Imaging Systems and Technology, vol. 2, p.183-99, 1990.

Roman, D., Fisher, M. & Cubillo, J., *"Digital Image Processing – An Object-Oriented Approach"*, IEEE Transactions on Education, vol. 41 no. 4, p.331-3, November 1998.

Russ, J. C., *"The Image Processing Handbook"*, 2nd Ed., CRC Press, Inc., Boca Raton, Florida, USA, 1994.

Sonka, M., Dove, E. L. & Collins, S. M., *"Image Systems Engineering Education in an Electronic Classroom"*, IEEE Transactions on Education, vol. 41 no. 4, p.263-72, November 1998.

Zukowski, J., *"Mastering Java 1.2"*, Sybex, San Francisco, USA, 1998.

Appendix 1: Optimally Accessing Image Data in Java

Certain aspects of the Java programming language are found to lead to a variety of approaches, or 'patterns', for accessing image data. This appendix investigates the pros and cons of the various options from a pragmatic perspective, and evaluates their performance characteristics by experiment. To help focus this investigation and simplify the experiments, only 8-bit grey scale image data is considered. From the results of these experiments, an attempt is made to reconcile the performance costs of adopting various patterns with other more pragmatic considerations such as source readability, design flexibility and code maintainability. The conclusions that are drawn relate to the 'optimal' patterns given different processing requirements.

A1.1 Introduction

Writing image processing software in any language requires some scheme for accessing the image data. The approach chosen will impact on the resulting programs for the whole of their lifetimes (Lyon, 1999). Hence, it is important to understand the various options and their advantages and disadvantages.

This appendix investigates the problem of accessing 8-bit grey-scale images using the Java programming language (Gosling, *et al.*, 1996). For these purposes, Java imposes certain constraints on the programmer that lead to a variety of approaches for dealing with this common pixel format. These constraints stem mainly from:

- **Java's lack of pointers**. This limitation means that, given an array of numbers representing an image, the only way to access those numbers is via the array reference itself.

- **Java's lack of an *unsigned byte* type.** This rather troublesome limitation means that it is not possible to optimally store and manipulate images whose data values fall naturally into the range 0 to 255 (i.e. an 8-bit range). Rather, the programmer is forced to use other primitive types to do an equivalent job.

The full implications of these constraints will be discussed in the following section where several different 'storage/access patterns' arise. Each pattern sets out: a) how the image will be represented (stored); and b) how the pixels in the image should be read and/or modified (accessed). The purpose of this investigation is to determine the pros and cons of the various options available and to quantify the performance differences between them.

A1.2 Storage/Access Patterns

There are a number of factors that can be varied to give different patterns. This section discusses these contributing factors along with the impacts each is likely to have on the resulting patterns.

A1.2.1 Array Configuration

Though arrays must be used, there is still the choice as to whether to specify these as having either one or two dimensions. One of the drawbacks of one-dimensional arrays is that they make it necessary to calculate a pixel's array index from its x and y coordinates before accessing it. This has potential consequences for source code readability, especially if the code must be littered with multiply/add combinations wherever a pixel is being referenced.

One-dimensional arrays also lack 'automatic' bounds checking along both image axes. Java verifies that all array accesses are within bounds, and throws an exception if this is not the case. This is not guaranteed to happen in the one-dimensional case if, for example, an image stored by row is accessed at a location whose x coordinate is greater than the image's width. As a result, bugs in the code may be harder to detect and fix. The work-around is to manually check bounds, but this only burdens the programmer with unwanted detail and hence threatens readability even further.

One up-side of one-dimensional array usage is that the programmer can have much greater confidence as to the order in which main memory is being accessed during a scan through the image (Lyon, 1999). This is because the programmer can control the order in which index values are calculated. Conversely, Java makes no guarantee as to whether two-dimensional arrays are stored in row- or column-major order. Hence it is possible, using two-dimensional arrays, to

unwittingly cause a program to jump around wildly in main memory as it processes an image. For large images and relatively small amounts of main memory, this can cause 'thrashing' as virtual memory is swapped in and then back out to disk in order to accommodate repeated non-localised memory accesses. Fortunately, owing to the increasing availability of inexpensive RAM, it is safe to ignore this consideration in the majority of cases.

In the two-dimensional array case, bounds checking is automatic on both axes, and readability does not suffer since pixels can be referenced via their x and y coordinates directly. Thus, it only remains to determine the performance cost in choosing two-dimensional over one-dimensional arrays (or vice versa). This will be determined by experiment.

A1.2.2 Primitive Type Selection

Given that the requirement is to store images whose pixels values lie in the range 0 to 255, a primitive type must be selected to hold these values. Again, there are two obvious options: use *unsigned bytes* or use *shorts* (which in Java are 16-bit signed quantities). Using *shorts* means the required range is fully represented. However, because they take up 16 bits instead of 8, memory requirements are effectively doubled. Using *bytes* gets around the memory problem, but now all values above 127 become negative.

This problem can be addressed by 'shifting' or offsetting the values to the required range. Whereas normally a pixel value of 0 represents black and 255 represents white, instead –128 can be treated as black and +127 as white. This yields equivalent 'less than/greater than' relationships between the grey levels. Using this scheme, many image processing operations function equivalently, for example thresholding an image (as long as the threshold value also follows the new convention).

However, things are not always this simple. For example, care must be taken when adding two images together. Take two images, both of which have pixel values that are all zeros, and hence appear mid-grey. The expected result upon adding these two images is an image that is substantially whiter than the originals. Instead, the same mid-grey image is produced, since zero plus zero equals zero.

Of course, the true result can be obtained by applying a correction, that is, by adding 128 to each pixel value. However, this raises two additional problems in itself. First, adding a correction is an additional operation that has to be performed for each output pixel, and hence performance will be degraded. Second, the correction itself is dependent on the type of image processing operation being implemented. In the case of a non-linear convolution, for example, the proper correction actually depends on the kernel coefficients themselves. All of this means additional code, which affects both performance and source readability.

The correction calculation can be avoided in all cases by always adding 128 to a pixel value before it is used, and subtracting 128 before it is written back to the image. This works in Java since all types smaller than 32 bits are promoted to *integer* type before any arithmetic is carried out. However, this does not solve the problem so much as shift it around somewhat.

A1.2.3 Access Strategy

So far, it has been assumed the programmer has access to the array containing the image. Thus, indexes directly into that array are needed in order to read or modify the pixels. Here again, another option is available, which involves providing method calls to access the image data. This is a convenient way of hiding implementation detail. For example, the code to add and subtract 128 (referred to in the previous sub-section) can be put within a pair of get- and set-pixel methods, as can code to check bounds or calculate one-dimensional array index values. The programmer can now use *byte* types or one-dimensional arrays without concern for correction factors, out-of-bounds errors or index calculations.

Unfortunately, if the access methods provided attempt to do *too* much, they will be relatively slow to execute. As a result, many simple image processing operations will be hindered unnecessarily by code they do not need. In addition, there is a performance overhead associated with method invocation itself that needs to be taken into account. The former can be addressed by providing multiple access methods, allowing the programmer to trade-off performance with functionality. The latter can be ameliorated by using non-virtual methods. In Java, all methods are virtual unless they are declared *final*. Declaring a method to

be *final* by-passes the virtual look-up overhead, resulting in increased performance. The down-side is that *final* methods cannot be overridden, thus sacrificing some design flexibility.

A1.3 Experimental Design

A1.3.1 Evaluation Criteria

From the previous section, the trade-offs among the various patterns are mainly in terms of:

- **performance** – how fast the code runs;
- **space efficiency** – how much memory is used;
- **source readability** – how easily the source code is able to be understood, debugged and maintained;
- **flexibility** – the extent to which the programmer is able to take advantage of particular language features Java may support.

A1.3.2 Patterns in the Experiment

Table A2.1 below summarises the patterns included in the experiment. The first column of the table gives a code name for each pattern, which will be referred to in the results and discussion section. The second column briefly describes the pattern itself. The third and final column gives a Java code example of how pixels would typically be accessed (read from memory) using each pattern.

Name	Description	Code Example
D_B_1	Direct Access using One-Dimensional Arrays of *Bytes*.	`p = pixels[x*width+y]+128;`
D_B_2	Direct Access using Two-Dimensional Arrays of *Bytes*.	`p = pixels[x][y]+128;`
D_S_1	Direct Access using One-Dimensional Arrays of *Shorts*.	`p = pixels[x*width+y];`
D_S_2	Direct Access using Two-Dimensional Arrays of *Shorts*	`p = pixels[x][y];`
MV_B	Virtual Method Access using *Bytes*.	`p = getPixel(x,y);`
MV_BU	Virtual Method Access (No Correction) using *Bytes*.	`p = getPixel(x,y)+128;`
MV_S	Virtual Method Access using *Shorts*.	`p = getPixel(x,y);`
MF_B	Non-Virtual Method Access using *Bytes*.	`p = getPixel(x,y);`
MF_BU	Non-Virtual Method Access (No Correction) using *Bytes*.	`p = getPixel(x,y)+128;`
MF_S	Non-Virtual Method Access using *Shorts*.	`p = getPixel(x,y);`

***Table A2.1:** Patterns compared in performance evaluation experiment*

The first four patterns use direct array access. Among these, using two-dimensional arrays of *shorts* is perhaps the most 'natural' pattern. This is because two-dimensional most naturally 'map' to the concept of an image as a surface having both width and height, and similarly, Java's *short* data type affords the most natural representation of the required 0 to 255 grey-level range. Using this pattern, pixel values can be accessed by specifying the x and y coordinates as the array indices directly, and those values can be used without correction.

Deviation from this natural choice to other combinations is now considered. Two-dimensional arrays of *bytes* sacrifice readability and incur some processing overhead since the pixel values must be corrected before use, but gain by halving memory requirements. One-dimensional arrays of shorts and one-dimensional arrays of bytes are also considered in order to compare the performance differences between indexing into one- and two-dimensional arrays.

The remainder of the patterns use method access. Although not made explicit in the table, all of these patterns utilise one-dimensional arrays. This is because it is no longer necessary to calculate array indices manually (the code for this is hidden inside the methods) so readability is not affected. Also, the comparison between the two types of arrays is already provided for by the direct access patterns. Note that none of the methods include bounds checking code.

Access patterns using both *shorts* and *bytes* are also considered. In the case of *bytes*, access methods with and without correction code are compared. In each case, the use of both virtual and non-virtual method types is considered.

A1.3.3 Experimental Conditions

In order to evaluate the performance of each pattern, the same image processing operation was performed multiple times, once for each of the chosen patterns. Timing measurements were then made for each implementation by measuring the time each took to complete some image processing task.

The image processing task chosen was a three-by-three (window) convolution. Care was taken to implement each pattern to produce identical results, with no 'tricks' or optimisations made.

For each implementation, an identical 256 by 256 pixel image was processed eight times over (as an alternative to processing a much larger image) to give each timing measurement. This helped to increase the measured times relative to the timing precision, thus reducing the effects of noise in the observed data. Each measurement was repeated fifty times per implementation to get an estimate of the spread of the data in each case.

A Pentium II 233Mhz system with 64Mb of RAM running Microsoft® Windows NT 4 and the Sun JDK 1.1.8 with Just-In-Time compiler enabled was used to run all of the trials.

A1.4 Results and Discussion

Figures A2.1a and A2.1b below contain the results obtained from the experiment. Figure A2.1a graphs the entire data set, with each bar corresponding to a different pattern. The vertical axis is the processing time, in milliseconds; each bar represent the average of the fifty measurements made (though not indicated on the graphs, the timing resolution of these measurements was found to be around 10 milliseconds). Most obvious from this graph is the fact that using virtual methods incurs a large performance penalty. Also, it is immediately obvious that using non-virtual methods may be a viable alternative to direct array access since the two have comparable performance measures.

The graph in figure A2.1b ignores the results from the virtual method patterns and focuses on the 'fast' group. It appears that two-dimensional array access is slower than the one-dimensional case. However it is to be remembered that in the former case automatic bounds checking is performed, which would contribute in part to the observed slow-down.

Looking at the lower trace of the second graph, it can be seen that direct array access (in the one-dimensional case at least) is the fastest option. Hence the overhead of method calls, even the non-virtual variety, is not completely unavoidable. However, the average differences involved are quite small, between 10 and 20 milliseconds. The corrected and uncorrected method calls have very similar results, indicating that the convenience of the former is almost certainly worth having.

Finally, the choice of primitive type does not appear to affect performance to any significant extent. This decision is more of a trade-off between memory requirements and convenience. If method calls are used (and it is clear that this is certainly an option) then the use of *bytes* becomes the best choice since the correction details can be hidden.

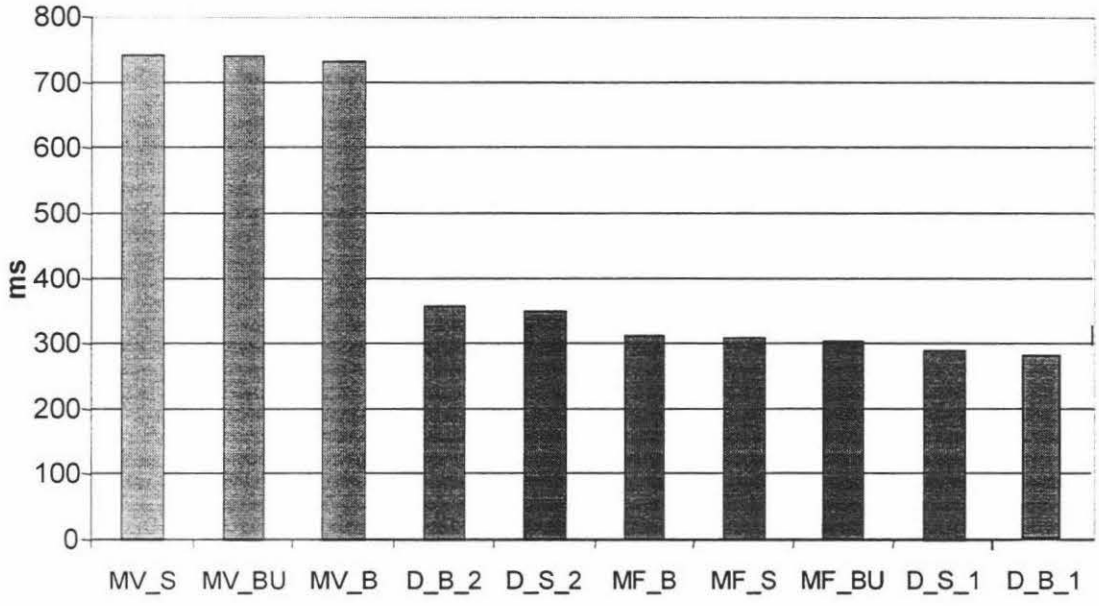

Fig A2.1a: *Graph of results from all trials*

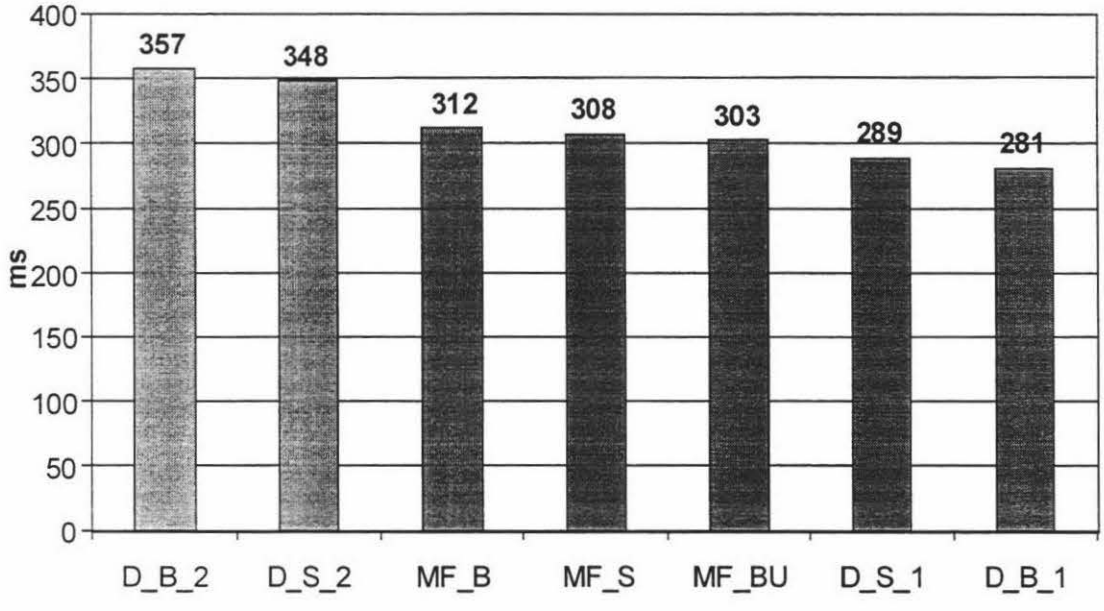

Fig A2.1b: *Graph of non-virtual method call and direct access results*

A1.5 Conclusions

Ideally, it is desirable to:

- keep memory requirements to a minimum;
- have fast access to the image data; and
- keep the source code uncluttered;

The data collected demonstrates that using non-virtual method access on an array of *bytes* is a viable approach to achieving this. In addition, this allows encapsulation to be preserved as the underlying array itself can be made inaccessible to the programmer.

In an image processing system environment in particular, memory requirements will be important since users are likely to want to work with multiple images simultaneously. Hence *bytes* will be preferred over *shorts*. Method calls can then be used to get around sign problems associated with *bytes*. Only if performance becomes ultra-critical, would it be necessary to do away with method calls and access the array directly.

The advantage of direct access is that it gives maximum power to the programmer to optimise the source code depending on the image processing operation being implemented.

At the other end of the spectrum, performance may be of little concern. For instance in software written purely for demonstration purposes, e.g. in teaching image processing. In this case, simplicity is most important. Again, method access can provide a clean interface to an image. If method access is undesirable, Another option in this case would be a two-dimensional array of *shorts*, as this provides a natural image representation.

Virtual methods incur a large performance penalty, but do add flexibility to programs. There may be some applications where certain object-oriented programming techniques are needed that require virtual methods. In this case, performance would not be a priority.

In conclusion, it has been found that using one-dimensional arrays of *bytes*, with non-virtual method access and with correction, is the best all-round approach to working with 8-bit grey-scale image data in Java.

A1.6 Future Work

The two main directions for future work that have been foreseen are:

- Repeating the experiment on multiple architectures, multiple operating systems, and multiple Java virtual machines (including within an applet context, i.e. in a browser);
- Extending the experiments to include colour image data.