Tools and Techniques to Aid in the Marking and Categorisation of Digital Images

By Colin Callanan

Institute of Technology, Sligo

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

A thesis submitted in fulfilment of the degree of Master of Science

2006

Supervisor: Mrs. Dana Vasiloaica

Submitted to the Higher Education Training and Awards Council, August 2006

ABSTRACT

Image processing technology is used in everyday applications to do things such as correct red-eye in digital cameras, and to detect terrorists at airports. Recently it has become more widespread and ubiquitous in society. With the surge in use of image processing algorithms and technology, the need for reliable evaluation for those technologies has increased, and research on objective and quantitative performance estimation methods is actively investigated. This thesis starts out by giving a comprehensive overview of image processing algorithms and performance evaluation techniques, ultimately drawing the focus to the area of testing algorithms using ground truth measurements which allow accurate reports on algorithm success and accuracy to be carried out. After understanding this area, it was revealed that no comprehensive tools exist in the industry for the gathering of ground truth data. Thus, the research moved towards finding out what kind of working methodology would best tackle the important area of ground truth gathering through image marking. After this thorough study, a promising framework was proposed, a framework that would certainly fulfil all the requirements of the difficult ground truth gathering task. The thesis then details how the software for ground truth gathering was designed and implemented based on the established framework. The research concludes that the tool developed certainly helped aid the task of algorithm testing, as proven by the testing and use of the application at the partner company.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dana Vasiloaica, who gave some much needed help and guidance throughout the composition of this thesis. I would also like to thank Michael Barrett at the IT Sligo for steering me in the direction of this MSc. after I had finished my degree. I send my gratitude to all of the staff at FotoNation Ireland in Galway, who helped us get the project started and gave plenty of feedback throughout the various stages of the project. Additionally, I must thank my co-researcher John Brady, who offered much advice and help along the way, and my family who gave much support throughout the duration of the project.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Declaration

TO WHOM IT MAY CONCERN

The work in this thesis: "Tools and Techniques to Aid in the Marking and Categorisation of Digital Images" represents the research carried out by Colin Callanan under the supervision of Dana Vasiloaica, and does not include work by any other party, with acknowledged exception.

Signed: Colin Callanan

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

List of Figures 8
Chapter 1: Introduction 10
Chapter 2: Literature Review 13
2.1 Introduction 13
2.1.1 Digital Images and People 13
2.1.2 Image Processing Advancements 14
2.2 Algorithm Development and Testing 20
2.2.1 Exploration of current development and testing tools 22
2.2.2 Case Study: FERET 28
2.2.3 Factors Hindering Algorithm Improvement 30
2.3 Marking and Categorisation 32
2.3.1 Performance Evaluation and Ground Truth 34
2.3.2 The Categorisation Process and Use of Markings 37
2.3.3 Importance of a Good Marking Tool 37
2.5 The Proposed Solution – An Overview 39
Chapter 3: Methodology and Requirements 42
3.1 Introduction 42
3.2 Evaluating the Algorithms 42
3.2.1 The Testing and Reporting Methodology 43
3.2.2 Discovering the Requirements for the Framework 45
3.3.1 Overview of Framework Components 48
3.4 Discovering The Requirements For Image Marking 51
3.4.1 Use Cases 51
3.4.2 A Framework for Marking 55
3.5 Overview of the Framework and Marking Tool 58
3.5.2 Goals of the Marking System 59
Chapter 4: Software Design 61
4.1 Design at the Architectural Level 61
4.1.1 Conceptual Class Diagram 62
4.1.2 Class Diagram 63

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.1.3 Sequence Diagram 66
4.1.4 Overall System Architecture 68
4.2 Technical Design 69
4.2.1 Storage Mechanism – XML Database 69
4.2.2 Revised Class Diagram 75
4.2.3 Graphical User Interface Design for the Image Marking Tool 77
4.3 Design Conclusion 93
Chapter 5: Implementation **95**
5.1 Approaching Implementation and Testing 95
5.1.1 Exploration of Programming Environments and Other Resources 96
5.1.2 Selection of Programming Environment and Development Technique 99
5.2 Database Interaction and Interfacing with the Client 100
5.2.1 Client Side Requirements 100
5.2.2 Database Specification 100
5.2.3 PFML Server Side Implementation 103
5.2.4 Building the Data Model on the Client 110
5.2.5 Overview of the Database and Data Model 125
5.3 Implementation of the Client Interface 126
5.3.1 Interface Construction and Controls 127
5.3.2 Integrating the Data Model with the View 133
5.4 Integration with the Image Testing and Reporting Framework 153
5.4.1 Java Perspectives 153
5.4.2 Sharing Data Throughout The Application 156
5.4.3 Goals Fulfilled 157
Chapter 6: Testing **158**
6.1 Testing Strategy 158
6.2 Functional Testing 160
6.3 Usability Testing 175
6.3.1 A look at usability 175
6.3.2 Improving Usability 176
6.3.3 Usability Test Results 176

IT Sligo An Institiúid Teicneolaíochta, Sligeach

6.3.4 Usability Evaluation 178
6.4 Conclusions of Test 179
Chapter 7: Conclusion and Recommendations **180**
7.1 The Research Journey 180
7.2 The Resultant Tool 181
7.3 Recommendations for Future Work in the Field 182
7.4 Conclusion 182
APPENDIX A: XML Attributes **183**
APPENDIX B: Geometry Types Defined in the Schema **184**
Point 184
Circle 184
Ellipse 184
Rectangle 185
LineString 185
LinearRing (aka Polygon) 185
APPENDIX C: Usability Questionnaire **187**
References **190**

IT Sligo An Institiúid Teicneolaíochta, Sligeach

List of Figures

Figure 2.1: Acquisition of a Digital Photo........14
Figure 2.2: Algorithm Execution........17
Figure 2.3 Image Processing Applications........18
Figure 2.4 Algorithm Development Cycle........20
Figure 2.5 Algorithm Result Analysis........27
Figure 2.6: Image Marking........32
Figure 2.7: Ground Truth Comparison........34
Figure 3.1: System Inputs and Outputs........43
Figure 3.2: Image Marking Inputs and Outputs........44
Figure 3.3: Overall Framework Use Cases........46
Figure 3.4: Framework Overview........48
Figure 3.5: Image Marking Use Case........52
Figure 3.6: Image Marking Application Overview........55
Figure 3.7: Image Marking Tool........57
Figure 4.1: Tree Hierarchy........61
Figure 4.2: Image Marker Tool Conceptual Class Diagram........62
Figure 4.3: Model-View-Controller........64
Figure 4.4: Sequence Diagram........66
Figure 4.5: Conceptual Model........68
Figure 4.6: Getting an Image from the Image Database........70
Figure 4.7: Sending an Image Back To The Image Database........71
Figure 4.8: Send Markings Data To The Client........72
Figure 4.9: Revised Class Diagram........75
Figure 4.10: The Old Image Marking Tool........79
Figure 4.11: Photoshop........81
Figure 4.12: Interface Prototype........82
Figure 4.13: New Design Sketch........85
Figure 4.14: Views in the Image Marking Tool........89
Figure 5.1: Extracting Data from the XML Document........101
Figure 5.3: PFML API Packages Used........123
Figure 5.4: Interface Components........130

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Figure 5.5: Data Model and View Integration 134
Figure 5.6: Exchanges Between the Views 149

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Chapter 1: Introduction

The Image Processing Industry

In the last ten years there has been a digital image revolution, with soaring interest in image processing technology across the consumer and business landscape. Algorithms – advanced mathematical formulae that carry out a specific task – are used in many of the new digital cameras for jobs like red-eye detection, red-eye correction and image enhancement. Digital devices with such technology have become a ubiquitous and requisite commodity for the recording, displaying and communication of visual representations. IDC's European Consumer Digital Imaging Survey claims that "there has been a meteoric adoption of digital photography in Europe" (IDC Press Release, 2005:1). The technology has become less expensive, and consumers have a desire to use Image Processing technology in their day-to-day lives for tasks as simple as capturing, editing and archiving photos taken on family occasions, sporting events and holidays with digital cameras.

Purpose of Research

Digital imaging, a multibillion dollar industry incorporating areas such as digital cameras, airport security and medical screening, is ever in need of more efficient, innovative algorithms to perform image processing tasks such as face detection, red-eye detection and correction, retinal scanning, retinal identification, and so on. While a vast amount of effort goes into the development of advanced algorithms, little work has been done towards refining and automating their testing process. The accuracy of an algorithm can only be proven by extensive testing and it is this very process that can speed up, or slow down the algorithm development process. Also, the actual accuracy and quality of an algorithm can be improved through rigorous testing. Thus, this thesis intends to study the current approach to algorithm testing, and to identify ways in which testing process can be improved. Based on the research, it is hoped that a new framework for testing can be formulated; a framework that will help to rectify any of the current problems that exist within the image processing algorithm testing field.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Research Approach

In order to gain an understanding of the image processing field, time was initially spent examining emergent literature in areas such as face detection, biometrics, algorithm development, and algorithm testing procedures. After analysing writings and applications from the image processing field, weaknesses were identified within current practices. A study of the testing methodology was conducted, and thus a framework that would cover all areas of the image testing and marking process was uncovered. After coming up with a framework for testing, it was necessary to turn this concept into a tangible, practical software project. Thus, an application to fulfil the needs of advanced algorithm testing was developed and tested.

Developments in the Field

Image processing algorithms are executed on the image-test-set. For this thesis, the algorithms used are those that must detect a specific object within an image. In reality, the object may be partially visible, may be at different angles, may be various distances from the camera or may be out of focus. Thus, an image-test-set consists of a set of images along with markings which state the locations of the objects that must be detected. A well varied database of images is required. Regarding image sets where the object to be detected is a face, a number of standardized databases are already available from various universities and research institutes, such as the MIT Test Set (Sung and Poggio, 1998) and the Kodak data set (A. C. Loui, C. N. Judice, S. Liu, 1998) and the UCD Colour Face Image Database for Face Detection (Sharma and Reilly, 2003). However, a lot of the image test sets found are limited to simplified backgrounds where the item for detection is relatively easy to detect. Therefore, one of the requirements of this thesis is the creation of a suitable image database to be used for algorithm tests. The image set will vary depending on what objects must be detected. For instance, for red-eye detection, a good selection of images demonstrating red-eye instances must be available. The following broad requirements have been identified for the image database to be used in this application:

(i) Objects at various distances from the camera

IT Sligo An Institiúid Teicneolaíochta, Sligeach

(ii) Objects in Complex Backgrounds

(iii) Partially obscured objects

Structure of Thesis

The thesis is divided into seven chapters. Chapter 2 surveys the literature on image processing and identifies various characteristics of the algorithm testing field that are of interest, as well as discovering the overall lack of research and development in the area of algorithm development. Chapter 3 examines the methodology used for the testing of algorithms and proposes a framework that will solve many of the problems that exist in the algorithm testing field. A set of requirements are also drawn up at this stage, and these requirements comprise the key goals of the research. Chapter 4 aims to evolve the conceptual framework into a practical software application. Classes are identified and some technical design issues are tackled. Chapter 5 details the actual implementation of the solution, showing in detail the technical accomplishments of this research, and explaining the inner working of selected key areas of the final application. Chapter 6 looks at a testing strategy for the application, and gives an account of usability tests that were carried out. Finally, Chapter 7 gives a short conclusion, and explores some recommendations for possible future work.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Chapter 2: Literature Review

2.1 Introduction

2.1.1 Digital Images and People

For centuries people have had an unwavering interest in taking photographs. Traditionally, photographs have been used for capturing portraits of people, places and objects; people are interested in recording portraits of themselves and places visited. People want to retain a physical artefact of important events that have taken place, of a holiday or journey, or of friends made along the way. People want to record photographic evidence of their children growing up, as well as the household pets and animals.

It can be seen that people are using digital cameras more often and in more ways and situations than they used film cameras. Some of the new and interesting uses are highlighted by Narayanaswami, Raghunath; "Because digital images require little physical space, some users have converted their children's space consuming art projects into compact yet accessible digital albums" (Narayanaswami, Raghunath, 2004:65). A photograph class known as ephemeral images has emerged out of digital camera use, helping to serve as a memory aid or for transcription. Such images are usually deleted after use; It can be said that the apparent increased usage is down to the nature of digital photography; "Consumers' desire to capture images using digital cameras has been shown to be remarkably high, which reflects the ability to take, re-take, and delete digital photos as desired" (IDC Press Release, 2005:1). The user may archive, digitally manipulate, email, and print or upload captured images to the Internet with ease. The low cost of capturing, storing and viewing digital images on small devices such as mobile phones have prompted this sort of usage. In the coming years, it is likely that people will find ever broader uses for their imaging devices.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

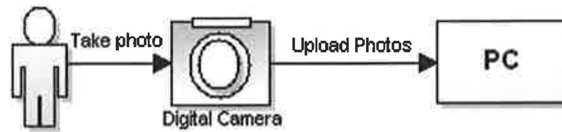


Figure 2.1: Acquisition of a Digital Photo

Narayanaswami and Raghunath state; "We expect digital cameras to be used symbiotically in several environments, including homes, offices, shops, cars, airplanes, and trains", one of the trends being that "We can expect camera storage capacities to continue doubling almost every year" (Narayanaswami, Raghunath, 2004:67). The same article concludes with; "Digital cameras with short and long-range wireless communications capabilities and large storage capacities are poised to change the way we capture, view, and use digital images" (Narayanaswami, Raghunath, 2004:67).

2.1.2 Image Processing Advancements

Digital Image Processing is said to be the analysis and manipulation of images with a computer, and involves:

a) The acquisition of an image through a digital source like a scanner or a digital camera
b) The manipulation of the image in some way. This could include compression, enhancement, and analysis of the image.
c) The presentation of the result in some way, for instance, if the result is an altered digital photo, then it could be printed on photographic paper or published to a website. Alternatively it may be the results of image analysis that took place in (b), e.g. "Two eyes have been detected in the image".

According to Hongjun Xu, "Digital Image Processing is the study of representation and manipulation of pictorial information by a computer" in order to "Improve pictorial information for better clarity (human interpretation)" and "Automatic machine processing of scene data (interpretation by a machine/non-human, storage, transmission)" (Xu,

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2003:4). In short, Image Processing technology enables the manipulation and analysis of data and information in the form of images.

Due to increased consumer demands, the technology behind digital cameras is evolving rapidly. For instance, instead of just capturing an image based on reflected light, the digital cameras of today have a very complex CCD (charge coupled device) which captures an image by storing light as individual pixels which make up a total image – often consisting of tens of thousands of pixels. The stored image is then processed by the camera and saved to memory.

Once an image has been stored in the memory of the digital camera, it can be processed further. Here are some examples of the image processing tasks that may take place:

a) The image may be digitally transferred from the camera to a PC. The user may choose an editing tool such as PhotoShop to enhance or edit the image.
b) The image may be processed and altered by the actual digital camera or capturing device itself. For instance, the camera may contain technology to eliminate red-eye from photos. If the user has "Red-Eye Correct" switched on, then the camera will process the image after it has been taken. In the processing stage, the camera will detect the red-eye occurrences in photos and will alter the image to fix the red-eye.

Regarding the red-eye example given in B) above, a program built into the camera might use some sort of advanced problem-solving formulae to automatically detect and correct the occurrence of red-eye in images – this logical mathematical formula is referred to as an algorithm.

There are many algorithm types that may be used to deduce information from visual images; face-recognition, red-eye detection, red-eye correction, product inspection, biometrics, fingerprint analysis, disease detection are just a few examples. This thesis is principally concerned with algorithms that detect and correct everyday faults in consumer digital images, such as face, red-eye and dust detection and correction algorithms, and that is why such examples are frequently used within this thesis. When conducting the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

literature review, much research material on other techniques was also uncovered, but they are too numerous to mention, and do not hold relevance to the work being carried out.

2.1.2.1 Common problems solved by algorithms

As advanced imaging technology is finding use in everyday tasks, advanced detection algorithms help solve conventional problems that users may encounter with digital imaging media. As this thesis is predominantly concerned with photographs taken using digital cameras, it is useful to explore some of the algorithms regularly encountered within this field, namely:

- **Red-Eye Detection** – finds the occurrences of red-eye in an image. Red-eye is a condition caused by the camera flash reflecting off the back of a person's eye, resulting in a photo where an individual appears to have red/gold eyes
- **Red-Eye Correction** – manipulates the actual image to eliminate the appearance of red-eye
- **Dust Tracking** - finds the occurrences of dust in an image. Dust or spots of dirt may appear on the lens, and may affect the quality of the end photograph.
- **Dust Correction** – manipulates the actual image to reduce/eliminate the effect of dust on the overall quality of the image
- **Face Detection** – finds the location and extent of an anonymous face based on it's visual appearance
- **Face Recognition** – recognises an individual within a personal photograph collection
- **Blur Detection and Correction-** locates an area containing blur; correction algorithms will attempt to sharpen the problem area.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Taking **face recognition** as an example; it involves the detection of faces in an image, and the comparison of these detected faces with a database of known faces to determine if there is a match. Similarly, **face detection** algorithms will "determine whether or not there are any faces in the image and, if present, return the image location and extent of each face" (Yang et al, 2002:34). Margaret L. Johnson gives a brief outline of what happens within such recognition algorithms; "in a face recognition system, facial geometry algorithms work by defining a reference line – for example, the line joining the pupils of the eyes – and using it to measure the distance and angle of various facial features relative to this reference"(Johnson, 2004:92).

In Fig.2.2 below an image is fed into the computer system. In this instance, a face detection algorithm is run, and it finds all faces in the image. Secondly, the faces are extracted from the image. And thirdly, the system determines if the faces are recognised by its database of known faces or not. Identification is when the system matches a detected face with a particular known individual, and verification is when the system verifies that a person is who they say they are.

Figure 2.2: Algorithm Execution

All methods of image recognition involve the use of advanced algorithms, a small number of which are being studied in this research. In the above case, the algorithms must have the ability to detect the presence of blur, dust and red-eye. Yang et al. states, "To build fully automated systems that analyze the information contained in face images, robust and efficient face detection algorithms are required." (Yang et al, 2003:34). It is thus vital that algorithms have been constructed and developed meticulously, with a string testing and reporting framework to facilitate continual improvement and performance measurement.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.1.2.2 Real World Problems Solved

The results of the development and evolution in image processing, and the widespread use of digital images, have yielded many advances in the technology. Many scientific practices have benefited from such advancements. In particular, based on aforementioned evidence, it is clear that the area of image recognition has flourished. Ming-Hsuan Yang et al. (2003) claims that research efforts in face processing have helped to advance computer vision techniques such as face recognition, face tracking, pose estimation, and expression recognition. In face detection and recognition over two-hundred different approaches to algorithm development have been reported. Each approach comprises highly evolved mathematical calculations and analysis of images using techniques that have been evolving since the inception of digital imaging ("Detecting Faces in Images: A Survey", Yang et al.).

Digital Image Processing is used in a wide range of business and industrial applications, as shown by Hongjun Xu (2003, page 19) in Fig 2.3:

BIOLOGICAL:	Automated systems for analysis of samples.
DEFENSE/INTELLIGENCE	Enhancement and interpretation of images to find and track targets.
DOCUMENT PROCESSING	Scanning, archiving, transmission.
FACTORY AUTOMATION	Visual inspection of products
LAW ENFORCEMENT/FORENSICS	Face recognition, fingerprint analysis.
MATERIALS TESTING	Detection and quantification of cracks, impurities, etc.
MEDICAL	Disease detection and monitoring, therapy/surgery planning

Figure 2.3 Image Processing Applications

IT Sligo An Institiúid Teicneolaíochta, Sligeach

In most cases, image processing algorithms do something useful after extracting information from a digital image; "The rapidly expanding research in face processing is based on the premise that information about a user's identity, state, and intent can be extracted from images, and that computers can then react accordingly, e.g., by observing a person's facial expression" (Yang et al., 2003:34). With the continual research and development in imaging technology, the next natural step for its evolution is the realization of the technology to solve more pressing and critical problems. The security industry in particular has embraced image processing technology; incorporating algorithms that deduce information from visual images, including face recognition and fingerprint analysis technologies. Furthermore, there has been the development of more advanced image recognition systems that can detect human faces in photographs and video. With the growing security concerns after the New York bomb attacks on September the 11th 2001, there is a remarkable need for such technology.

Face recognition is now being used in many worldwide security systems, where it can be implemented in systems that detect known individuals – terrorists can now be recognised on video camera.

There is also evidence that the technology is evolving and moving into the 3D realm. A good example of such development is given by the Israelis Michael and Alex Bronstein (2003), who have worked on image recognition algorithms that may contribute to the advancement of international security. Such technology scans and maps the human face as an actual three-dimensional surface, providing a far more accurate reference for identifying a person than aforementioned image recognition systems, most of which rely on two-dimensional images. Such a product can potentially meet a wide range of security needs in a world shaken by the September 11 attacks and various bombings thereafter. "The system could be employed at airports or border crossings where a 3-D security camera could scan passengers' faces and compare them with a database of three-dimensional pictures of suspected criminals or terrorists, Michael and Alex Bronstein said." (CNN, 2003:1).

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.2 Algorithm Development and Testing

In order to build reliable and efficient systems that are able to deduce information from consumer digital images, consistent, strong and efficient recognition algorithms are necessary. While features such as face detection in surveillance systems or automatic red-eye removal in digital cameras have greatly improved in the past few years, they have still not reached desired standards; there are still many challenges in this arena; "Despite their success, many of the appearance-based methods suffer from an important drawback: recognition of a face under a particular lighting condition, pose and expression can be performed reliably provided the face has been previously seen under similar circumstances" (Belhumeur, 2005:2). And this is just one example. Later on in this chapter, current development and testing techniques are explored with a view to finding out what work can be done to aid in the evolution of digital image algorithm development through improved testing procedures.

Image processing algorithms require an involved development process, with many incremental improvements to fine-tune performance across a broad range of images, verifying performance at all stages of the testing process using testing and reporting techniques. "Algorithm development is a dynamic process" and "evaluations let researchers know the strengths of their algorithms and where improvements could be made. By knowing their weaknesses, researchers know where to concentrate their efforts" (Rizvi, 1998:14).

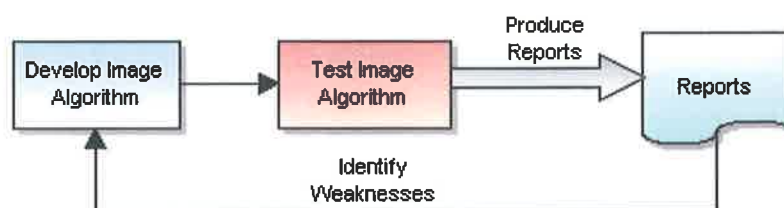


Figure 2.4 Algorithm Development Cycle

An Institiúid Teicneolaíochta, Sligeach
IT Sligo

Getting an algorithm to perform correctly is not a simplistic task. There are many unexpected and unpredictable scenarios within images which may throw a well developed and moderately tested algorithm off track. In the examples given in 1.2.1 – Red-Eye detection, Red-Eye correction and so on. many difficulties can arise. For example:

- **Red-Eye reflections** may be present, and may confuse the algorithms as to where the red-eye is located, as it is spread around the outside of the eye
- **Red-Eye false alarm** may occur, where the algorithm may detect a non-eye-related red shape in the photo which is the same size and color as a red-eye occurrence, in which case it is incorrect to alter the image
- **Blur false alarms** may occur if the algorithm decides an area is blurry, but the area is not actually blurred, e.g. a reflection in a pond.
- **Dust** may be falsely detected if dust-like dots make up part of the actual image, for example a design on a t-shirt.
- **Non-Detection** may occur if the algorithm fails to find either blur, red-eye or dust when occurrences of such are present.

Complications such as these mean that algorithm development may become quite difficult, as algorithms need to be altered time and time again to reduce error rates and to fine tune performance. "As recognition technology has matured and is being considered for more applications, the demand for evaluations is increasing. At the same time, the complexity and sophistication of the evaluations is increasing" (Grother, 2003:5).

Tools and techniques that aid in the job of inspecting and testing algorithms will certainly help to speed up the process of algorithm development and testing. This thesis is concerned with the job of inspecting and testing image processing algorithms, so by conducting some research in this area, knowledge is gained of the current tools and techniques developers have been using in this field of algorithm inspection and testing.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Thus it may be possible to find out how helpful such tools and techniques have been, and whether there are weaknesses in their application.

2.2.1 Exploration of current development and testing tools

"The result of evaluation leads the developer to develop better technology by analyzing the weakness" (Hong, 2004:8). Major evaluations like the FERET (Face Recognition Technology) tests, which evaluated emerging approaches to face detection, have helped to measure the power of face recognition technology and have also served to drive its evolution. The primary motive of FERET is "to assess the state of the art, identify future areas of research, and measure algorithm performance" (Phillips et al., 2000:1090 a – The Feret Evaluation Methodology for Face-Recognition Algorithms). According to Phillips, "Progress has advanced to the point that face-recognition systems are being demonstrated in real world settings. The rapid development of face recognition is due to a combination of factors: active development of algorithms, the availability of a large database of facial images, and a method for evaluating the performance of face-recognition algorithms." (Phillips et al., 2000:1090 a). If anything, FERET has provided a snapshot of how effective Image Processing algorithms can become when under constant scrutiny, evaluation and improvement.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

In this section, some of the evaluation tools and techniques currently being used in the imaging industry will be examined to help uncover the strengths and weaknesses of current practices. It will also become evident what work needs to be done if an improved testing and reporting framework is to be developed.

2.2.1.1 Testing and Improving Algorithm Performance Through Evaluation

Testing is required when developing any kind of computational algorithm, and in the field of image processing, testing is seen as a vital and crucial stage worthy of much time and effort. Whether the algorithms being tested are for face detection, red-eye detection, or fingerprint analysis, much importance is given to the testing and reporting stage of development.

This thesis is only concerned with more complex algorithms that require ground truth measurement, i.e. features that must be detected within an image are previously marked to measure if they have been detected. These are the kinds of algorithms where more than one object may be detected in an image, and where there is no simple true or false – detected or undetected – result that evaluations can be based upon. For instance, an algorithm that must recognise a fingerprint either finds a match, or does not. Whereas an algorithm that must detect the occurrence of red-eye in a photo may not find an instance of red-eye that has been marked (false negative), or may locate red-eye at a location that is not marked (a false positive). For this thesis, the researcher is interested in developing testing tools and techniques for such algorithms that require the use of markings – ground truth – to measure the efficacy of algorithms.

Algorithm developers use evaluation metrics and techniques to measure algorithm performance and evaluate. Here are some examples of some of the techniques that may be used when evaluating an algorithm:

A) Image Marking

As described by Sharma and Reilly in "A colour face image database for benchmarking of automatic face detection algorithms", "to evaluate the algorithm, a person must first go through all images in the database, marking the faces that he/she finds in each image" (Sharma, Reilly, 2003:3). This is the *image marking* stage of algorithm testing. In this particular example, the aim of the algorithm is to detect faces; therefore the face locations are stored for each image along with other image meta-data. Then automatic tests are run

IT Sligo An Institiúid Teicneolaíochta, Sligeach

which will reveal the false positives and negatives by comparing the markings done by a human with the actual detected regions. According to Prag Sharma and Richard B. Reilly (2003), it is then necessary to use an "accuracy measure to confirm a 'correctly detected' face. A spreadsheet with details of the faces present in each image of the database is also provided" (Sharma, Reilly, 2003:5). An example of such a spreadsheet is given in section 2.2.1.3. The actual process of markings images is of much importance. When images are well marked they can be easily categorised in the database, making it easy to run tests on specific categories of images, e.g. all faces with eyes. This is a key area in this thesis so it will be tackled in more detail in Section 3 – "Marking and Categorisation".

B) Metrics

For evaluation purposes, it is important to examine some of metrics developers employ when running tests. *Detection rate* is defined as the number of items correctly detected by the algorithm vs. the number of items determined by a human (as defined in the image marking stage in A above). A *false positive* is where an image region is declared to be a face but it is not and a *false negative* is where a face is not detected at all. "Thus, false positives and negatives can be automatically evaluated by comparing the location of the detected regions to the hand segmented results" (Sharma, Reilly, 2003:5).

C) Further Comparisons

Many developers use advanced mathematical formulae to calculate the performance of the current algorithm against the previous version of the algorithm, or against an algorithm developed separately, or by another vendor. This allows developers to find out whether the algorithm measures up against other developments in the field.

Regarding testing and evaluation techniques for other types of imaging algorithms, the field of biometrics holds some quite interesting applications of the technology, of which face detection is only one aspect; "Biometrics is a technology that automatically identifies or verifies an individual based on one's physiological and behavioral characteristics, and usually fingerprint, face, iris, voice and signature recognition" (Hong, 2004:2). By and large, many of the developments in biometrics are based on

IT Sligo An Institiúid Teicneolaíochta, Sligeach

identification and verification; "In identification applications, a system identifies an unknown face in an image; i.e., searching an electronic mug book for the identity of a suspect. In verification applications, a system confirms the claimed identity of a face presented to it. In recent times there has been much interest in biometrics for the purpose of security, thus augmenting the need for testing; "With the great interest in biometrics, it is necessary to evaluate the biometric systems accurately. For most biometrics products, the evaluation was not good enough to estimate the performance" (Hong, 2004:1). Again, the problem of insufficient testing and evaluation techniques crops up. In biometrics, a lot of testing is centered around verification and identification. Regarding technology evaluations; some examples are The Fingerprint Verification Competition (FVC) 2000 and 2002, which are technology evaluations that measure performance of single-finger fingerprint algorithms. Also the FERET (Face Recognition Tests) technology evaluations measure the performance of face recognition algorithms across multiple vendors. Evaluations have served to give industry benchmarks, allowing for vendor to vendor comparison, and improvement. FERET will be tackled in more depth in Section 2.2.2.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.2.1.2 Use of a Large Test Image Database

All imaging algorithms require a large database of images for training and testing purposes. *Training* refers to the process of getting an algorithm to work successfully with a finite set of visual images which may contain a limited number of people and scenarios. This set of images is known as the *training set*, and the algorithm is developed using this training set for testing. Taking face detection as an example, the algorithm will be able to detect all faces in the training set. *Testing* refers to the process of evaluating the algorithm using a different set of images – the *test set*. The algorithm will encounter images it has not worked with before and as a result, the real strengths and weaknesses of an algorithm will be revealed.

Based on evidence gained while investigating the specific area of face detection, it is evident that algorithm developers have conducted a lot of interesting research in testing and evaluation procedures. One of the problems identified in current testing and

evaluation practice has been the lack of a comprehensive image database. As Sharma and Reilly report in “A colour face image database for benchmarking of automatic face detection algorithms”, “Although several face image databases exist, most of them are geared towards the evaluation of face recognition algorithms” (Sharma, Reilly, 2003:2). A related problem has been the use of the same image data sets for both training and testing purposes; this means an algorithm is developed to detect faces in a certain set of images, and then the algorithm is evaluated based on its performance in detecting faces in this small range of images; performance in images outside that range is not evaluated. “An algorithm that is not designed properly will not generalize to another data set. To obtain an objective measure of performance requires that results are computed on a separate test data set.”(Phillips, 2005:2). Lei Zhang et al. summed up the main problems in face detection in the following piece; “Though efficient and robust face detection algorithms have become available, the effectiveness of available face recognition algorithms is still limited to images of mug shots in which faces are mostly in frontal and with reasonably homogenous lighting conditions and small variations in facial expressions” (Lei Zhang, 2004:1).

Consequently, many of the face detection algorithms developed have not been tested on an image database containing test sets which possess a high degree of variability in terms of scale, location, orientation, pose, facial expression lighting conditions, etc, according to Grother (2005). For instance the MIT database (MIT Database, 2006) consists of frontal and near frontal view images on a cluttered background. Such databases do not provide the challenges that face detection algorithms can encounter in real applications: such as poor image quality, presence of multiple faces and faces with different orientations (up-right and rotated). This is a universal problem which applies across the board, to all detection algorithms.

More recently, awareness of this problem has grown, and more researches have attempted to address this problem. Namely, image sets in databases set up by Sung and Poggioas, Schneiderman, Kanade and Rowley possess a higher degree of variability, often with more than one faces in images.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.2.1.3 Statistically based evaluation speeds the improvement and evolution of algorithms

P. Courtney and N.A. Thacker (2001) propose that the lack of algorithmic reliability has been due to the neglect of the important role that statistics must play in algorithm development. They report that one obvious problem is the testing of newly developed algorithms on a small number of carefully chosen test images, wherein fact the new algorithms effectiveness should be tested against those of existing algorithms. In the chapter "Performance characterisation in computer vision: The role of statistics in testing and design" (Courtney and Thacker, 2001), it is revealed that performance evaluation is not just finding out whether algorithms perform as expected; according to Courtney and Thacker, it involves the use of objective, usually statistical, measures for comparing the performance of vision algorithms (Courtney and Thacker, 2001). Thacker also makes the point that rigorous approaches to testing do exist such as test metrics like Feature Detection Reliability metrics like the ROC (Receiver Operating characteristic) curve, which is a graphical plot of the fraction of true positives versus the fraction of false positives. They propose a framework on how to compare algorithms to draw defensible, accurate and detailed conclusions about their performance. Such development work has helped to advance algorithm development practices in many ways; one thing is that developers can clearly see where problems lie. The data gathered in systems like ROC provides developers with very useful data quickly, and this in turn speeds the evolution and improvement of detections algorithms.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

	A	B	C	D	E	F	G	H	I
1	File Name (*.jpg)	Total Faces	Frontal	Intermediate	Profile	Upright	Rotated	Occluded	Structural Comp
2	Person0001.jpg	1	1	0	0	1	0	0	0
3	Person0002.jpg	1	1	0	0	1	0	0	0
4	Person0003.jpg	1	1	0	0	1	0	0	0
5	Person0004.jpg	1	0	1	0	0	1	0	0
6	Person0005.jpg	1	0	1	0	1	0	0	0

Figure 2.5 Algorithm Result Analysis

2.2.2 Case Study: FERET

In FERET tests, algorithms were evaluated against different categories of images. The categories were broken out by a lighting change, people wearing glasses, and the time between acquisition of the database image and the image being presented to the algorithm. Looking at algorithm performance in these areas provides an insight into the face recognition field, as well as the actual strengths and weaknesses of individual algorithms.

It is obvious that meticulous testing of algorithms is necessary in order to ensure accuracy and consistency across results, especially in systems where security is vital, e.g. a biometric system that verifies a human retina, and allows the user to operate military equipment. Algorithms must be tested to ensure every possibility and scenario for the use of the algorithm has been visited more than once. By exploring the FERET testing procedures, it can be seen what challenges exist in the testing of imaging algorithms.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Large Numbers of Tests and Large Databases of Images

The nature of algorithm testing is that it is quite time consuming and requires large numbers of tests to be run on images. For instance, in the paper "Detecting Faces in Images: A Survey" by Ming-Hsuan Yang (2002, P44), the distribution based methods mentioned used 4,150 positive examples of facial images, and 43,166 more images for the sample of non-face patterns to test an algorithm. The Inductive Learning Approach uses 2340 frontal face images in a FERET dataset in order to test its algorithms. The Hidden Markov Model was tested on an MIT database of 432 images, each with a single face, in order to determine the success rate of the technique, while the Information Theoretical Approach uses a face training database consisting of nine views of 100 individuals.

In the face detection field, it appears that developers formulate numerous algorithms but many of them are not tested on data sets where images possess a sufficient degree of variability in pose, location, facial occlusions (such as glasses, beards, etc). This problem

could be solved by integrating a vast image database into the testing architecture that is being proposed for this research. Databases will be used to empirically evaluate recognition algorithms in a specific domain. Thus the database used in this research will need to be able to accommodate a particular dataset of images and will need to provide an efficient interface to the retrieval and storage of such images. Within this set of images, there must be a wide variety of image content.

According to Yang et.al (2002:49), a database where "each image consists of an individual on a uniform and uncluttered background" is "not suitable for face detection". Rather, a large database of images with photos of individuals in various situations, poses and angles are necessary. For example, the Purdue AR database contains over 3,276 colour images of 126 people (70 males, 56 females) in frontal view with several varying factors, such as facial expression, illumination conditions and occlusions. One of the problems identified with FERET tests was that the majority of images consisted of individuals in full frontal poses, and images were non-colour.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

2.2.3 Factors Hindering Algorithm Improvement

One of the factors hindering algorithm development has been the lack of truly effective testing and reporting tools that allow the efficacy of algorithms to be measured and compared. This is known as performance characterization, performance estimation or benchmarking. "One of the major criticisms of computer vision over the last few years has been due to a general lack of algorithmic reliability. In our opinion, this has largely been due to the neglect of the important role that statistics must play in algorithm development." (Courtney, Thacker, 2001:3).

Algorithm testing in image processing has been carried out in a somewhat cumbersome and unsystematic fashion, without any standardised approach; While there are certain areas in computer vision where much literature is readily available and practices are generally standardised; the 9th Workshop "Theoretical Foundations of Computer Vision" conference reported that the research community faces a distinct lack of standardised, well grounded and industry-wide accepted methodology in the area of imaging algorithm testing (9th Workshop "Theoretical Foundations of Computer Vision", 1998). Now experts are beginning to research more into this area. "It is now accepted by many in the machine vision community that a more rigorous approach to studying the performance characteristics of vision algorithms is required." (Courtney, Thacker, 2001:3).

Without effective testing, the performance of algorithms is questionable, and it can be hard to pinpoint problem areas in algorithms or to compare algorithm performance. Thus, the key aim of this thesis was the development of a framework for testing and reporting to systematically evaluate algorithm performance.

Some of the limiting factors identified thus far include:

IT Sligo An Institiúid Teicneolaíochta, Sligeach

I. Statistical Tools

No comprehensive statistical tools are available to aid in face recognition algorithm comparison. While FERET proposed a standard testing protocol and standard dataset for evaluation, it was discovered by Yambor et al. that more involved and powerful algorithm evaluation tools are required. The article "Face Recognition: Hypothesis Testing Across All Ranks" (Mislav Grgic et al, 2005), reports that serious statistical tools have not been used in face recognition algorithm comparisons. Essentially, based on evidence seen in this paper, stats tools are available, but are not being deployed as widely as is desirable within the testing and reporting part of the development process.

II. Framework and Working Environment

Based on user profile studies at the partner company, it is evident that there is overall dissatisfaction with current testing and reporting practices, with some major shortcomings in the current testing and reporting system in use. After interviewing the user base, it has been found that:

- Users who worked on the marking of image sets for algorithm testing and reporting purposes found the task slow, difficult and tedious.
- The tools for marking images are not as accurate as desired, meaning that markings may not be sound. This means ground truth data will suffer, and the entire image algorithm testing process is made less effective.
- Algorithm developers saw a fragmentation in the different development stages for an algorithms lifecycle. Stages were not coherent, lacking a streamlined process where one stage feeds seamlessly into the next. The disparity between stages may contribute to a loss in development time and decreased developer focus. The different stages include; importing images to the system, marking images, adding metadata, image categorisation, and also the process of actually running tests on algorithms.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

As reported, testing and standardized reporting on image processing algorithms up to now is relatively underdeveloped. Based on studies carried out at the partner company, evaluation techniques currently in use do not live up to expectations, and different applic ations need to be used to carry out one task. Many incremental algorithm changes and improvements will be necessary, and each time the algorithm has to be retested using the same tools. Reporting tools have to be used after testing has taken place to evaluate algorithm performance.

2.3 Marking and Categorisation

As found out in Section 2.2.1.1.A "Image Marking", image marking and categorisation are important and worthwhile stages of the testing process. In image marking a person must first go through all images in the database, manually marking the faces found in each image. The face locations are stored (this is known as the ground truth data), along with other image data. Then automatic tests are run revealing the false positives and negatives by comparing the markings done by a human with the actual detected regions.

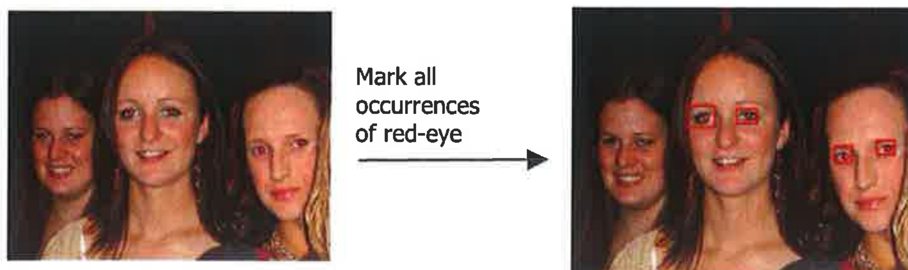


Figure 2.6: Image Marking

There are some interesting examples of the usage of such marking systems in the imaging industry at the moment. For instance, when comparing face verification algorithms using appearance models, Kang et.al. built the appearance models by using a set of labeled images, within which the areas of interest are marked and stored as co-ordinates, "All the images were marked up manually with 68 key points on each face. We match the models to these known points and extract the model parameters. This allows us to test the classification performance alone" (Kang et.al, 2002:480). Similarly, in research conducted by Feris, Gemmell, Toyama and Kruger during their project "Facial Feature

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Detection Using A Hierarchical Wavelet Face Database" (2002), the project involved the application of an affine transformation to feature positions taking into consideration the whole face. In order to evaluate the effectiveness if the algorithms, each set of automated feature localisations was compared with the hand marked locations of each feature. "An 'accurate' localization is characterized as one in which the feature was localized to within 3 pixels of the hand-marked position." (Feris et. al, 2002:10). Based on results shown in this paper, it is apparent that this technique of measuring algorithm accuracy worked well, and gave researchers solid, clear results. Another use of image marking was for the "Fast Face Detection and Pose Estimation" project (Fleuret and Geman, 2002) where poses were marked by going through images individually and manually marking the real pose on every face. By using this technique of evaluation, the distance between the real location of the pose and the one found by the algorithm could be accurately measured. Only one instance was discovered where hand marking systems were not adequate, and this was by Jaynes et.al, 2005, in a study where video cameras were used and more than 10-minutes footage had to be manually marked for every 500th frame– in this instance manual marking proved to be very slow; "the effort required over 100 human-hours of effort. This number includes only the hand-segmentation and labeling of subjects and objects in the video sequences. Clearly a more efficient approach is needed." (Jaynes et.al, 2005:7).

Because this thesis is concerned mainly with the task of image marking and categorization for algorithm testing and performance evaluation, it is worthwhile to explore the fundamental testing goals that are motivating development and research in this area of the imaging industry, and also to investigate the research and development work that has already been carried out.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.3.1 Performance Evaluation and Ground Truth

The key purpose of using a marking system to aid in performance evaluation is the acquisition of ground truth data. Ground truth data could be simply defined as the expected output; in the image marking systems already mentioned, the ground truth data is the marked data, e.g. a photo has 1 face with 2 eyes, so it is marked as such. If an algorithm finds 1 face and 1 eye in the same photo, then – according to our ground truth data – the algorithm has missed an eye, and is flawed. "Ground truth acquisition is the process that generates both the actual input for the evaluated system and the expected output (ground truth) for comparison with the actual output." (Dori and Liu, 1999:3). In Fig 2.7, the ground truth (shown as "Instances of Red-Eye") data are compared with the number of instances of red-eye detected by the algorithm through a range of images. As can be seen, in image 1, the ground-truth seems to match the detected instances, but on through image 2 and image 3, the detection rate has fallen, indicating that perhaps the algorithm is not performing as well under particular conditions that exist in image 2 and image 3.

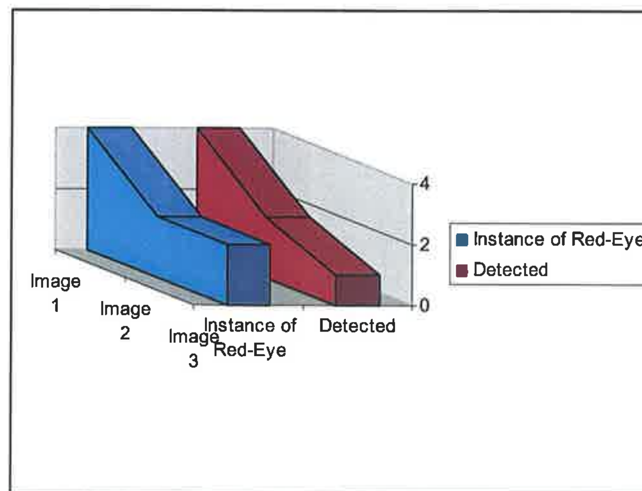


Figure 2.7: Ground Truth Comparison

IT Sligo An Institiúid Teicneolaíochta, Sligeach

In the publication "Principles of Constructing A Performance Evaluation Protocol for Graphics Recognition Algorithms" (Dori and Liu, 1999), a performance evaluation technique for use in graphics recognition algorithms is described. Graphics recognition algorithms are used in applications such as Optical Character Recognition software (software that recognises shapes of words in an image); " Graphics recognition is a process that takes as input a raster level image consisting of pixels or a vector level drawing consisting of symbolic primitives. The graphic objects that may be recognized from the input include text (character) regions, lines of various shapes (e.g., circular arcs and polylines) and styles (e.g., dashed lines and dash-dotted lines), special symbols, dimension sets, etc." (Dori and Liu, 1999:1). Graphics recognition algorithms are not completely dissimilar to those encountered in face detection and fingerprint detection, thus, the evaluation techniques used in this field are relevant and of interest. Also, the same problems are encountered in the graphics recognition systems as in those dealing with human features. One of the main challenges faced seems to be that of finding a system that allows for quantitative and objective evaluations and comparisons. There is a "lack of protocols that provide for quantitative measurements of metrics" and lack of "a sound methodology for acquiring appropriate ground truth data, and adequate methods for matching the ground truths with the recognized graphic objects" (Dori and Liu, 1999:1). In other words, the challenge in such systems is to measure whether the algorithm was detecting the "marked" data. Dori and Lio found that in order to advance the research in recognition algorithms and to reliably and successfully compare algorithms, there was a definite requirement for "the establishment of objective and comprehensive evaluation protocols and a resulting performance evaluation methodology" (Dori and Liu, 1999:1).

Dori views performance as a set of non-subjective metrics – or measurements – taken from the output data of an algorithm, against the actual expected output, or ground truth. The following three elements are stressed as being the real requirements to the successful evaluation of a recognition algorithm:

A. The expected output – or ground truth - needs to be known so it can be compared with the actual output, the results of the recognition algorithm. Importantly, it is

IT Sligo An Institiúid Teicneolaíochta, Sligeach

stressed that "a sound methodology of acquiring the appropriate ground truth data is required".

B. Each ground truth object – or in the case of human feature detection, each marked feature – must be matched up with, and compared with, the actual object recognized by the algorithms, and a sound matching method is needed

C. Representative metrics of interest must be used. These are the measurements that will be most helpful in evaluating the performance of an algorithm.

It is emphasised that real-life ground truth is highly desirable. However, the challenge expressed is that such "real-life" ground truth is difficult to acquire, as a large amount of intensive manual work is required to build databases of real life input, as well as their ground truths: "To comprehensively and thoroughly evaluate an algorithm on real-life drawings, real-life ground truth is highly desirable. However, this type of ground truth is hard to obtain, as it requires manual measurements, which are labor intensive and error-prone" (Dori, WenYin, 1999:3). Often, such data is error prone because of the method used to obtain ground truth. Another factor is that ground truth may be somewhat subjective; "Moreover, manual ground truth input is somewhat subjective and may vary from one human to another" (Dori, WenYin, 1999:3). Nevertheless, image marking is the only method to acquire ground truth for real-life drawings or images of people, so it needs to be employed as well as it can be.

When matching, Dori and Liu found that it was best to use a relative difference. In other words, the ground truth and the object recognised by the algorithm are not at identical locations, and the recognition result is to be matched with the ground truth – not vice versa – because the ground truth is what recognition is intended to reveal. "The ground truth is to be used as the basis for comparison" (Dori and Liu, 1999:4). "The performance of recognition algorithms is usually reflected by two rates: true positive and false positive" (Nalwa, 1993). According to Dori and Liu, it is important that the two rates are used together so that improvements can be seen by a definite increase in the true positive rate.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

To summarise; image marking allows a clear ground truth that all tests can be accurately measured by. Image markings an excellent way, and the only way, to work with large numbers of images and many test cases. If markings are accurate and precise, algorithm performance changes can be easily noted – improvements measured, etc.

2.3.2 The Categorisation Process and Use of Markings

When images are marked, the markings – or ground truth data – are stored as image metadata. Such metadata may not only be used for the purpose of algorithm evaluation, it may also be used as a means of categorising or annotating the images. Base on evidence gathered by Zhang et al. (2004), there seems to be an emerging trend in image annotation where keywords are associated with images or regions. According to Zhang et al., such categorisation tools have been helpful in solving image retrieval problems. However, the problem noted by Zhang et al. with some of the methods currently used to store images is that there are not any automated annotation functions. While the paper is centered more on categorisation tools and CBIR (content based image retrieval) for personal image collections, it is focused on categorising images by face identities so that a user may search for and retrieve all photos of a particular family member, or photos of all members of a particular group. Interestingly, this means that the categorisation tools given to the user will allow for the grouping of images based on a particular tag (e.g. ParisHoliday for all photos taken on a holiday in Paris), the annotation of individual images based on the person in the image, and the retrieval of images based on a query or tag.

Any image marking tool used in a testing/reporting environment will automatically generate image metadata as images are manually marked. For instance, such metadata may relate to the nose marked, the face marked, the person in the image, the date, a tag (showing what group the image belongs to), and so on. Such data could be used to help annotate and categorise images based on features marked, the date or the tag.

2.3.3 Importance of a Good Marking Tool

The ground truth – or in this case the image markings - are only as good as the tool used to mark them. A tool that provides an easy and fast way of markings images is not

IT Sligo An Institiúid Teicneolaíochta, Sligeach

available. If there are no good tools available for image marking, then the quality of the ground truth - the actual markings - may be questionable. This is especially true where a person must individually mark thousands of images – this is a manual and quite labour intensive job, requiring many hours of work. Based on what has been expressed by imaging researchers involved in the image marking process, it can be deduced that most tools are slow, hard to use, monotonous and non-intuitive. The process of marking images is a quite lacklustre affair, especially where a large number of images are used.

It is clear that there is a need for a tool that somehow helps to automate this marking process as much as possible, speeding it up and making it more user-friendly and intuitive. Such a marking tool would simultaneously and automatically categorises images based on marked content and other meta-data such as the name of the person or object in the image. A system combining such elements would be preferable to the current manual task in place.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

2.5 The Proposed Solution – An Overview

Throughout the exploration of literature, this writer has viewed many of the issues facing image processing algorithm testing in a modern development climate. This section attempts to relate some of the conclusions drawn post research by proposing an new innovative framework to encapsulate the entire testing and reporting process for algorithms.

2.5.1 The Proposed Framework

The framework proposed by this thesis is one that will provide an integrated and intuitive testing and reporting environment, consisting of the following tools:

- An Image Marking Tool for the establishment of ground truth data
- An Image Database Tool to store and manage collections of consumer digital images, and to interface with the two other tools in the framework
- A Testing Tool to run algorithm tests and to produce reports allowing for the measurement of algorithm efficacy.

The framework will streamline and speed up the testing phase, and thus development, of image processing algorithms.

Regarding the envisaged use of the proposed framework, it is likely that various image processing techniques will be tested on a set of images containing accurate ground truth data (images would have been pre-marked using the Marking Tool and saved into a database). Automatic and semi-automatic tools should be available in the Testing Tool to help with analysis of the results of image processing algorithm tests. Reports will be highly customable to help identify various problems occurring with image processing algorithms. As various versions of an image processing technique are tested on the same set of images, reports will clearly show which version is the most efficient.

Unlike previous algorithm testing techniques, algorithm evaluations can be conducted in an efficient and practical manner. This technology will improve the standard of algorithm performance in terms of speed and accuracy, and will also result in a shorter algorithm development lifecycle.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

2.5.2 Image Marking and Ground Truth Creation

More specifically, during this review of literature, the need has been identified for a semi-automatic marking tool to help facilitate in the testing of algorithms by providing reputable ground-truth data.

Some of the aims of the tool are as follows:

1. The tool must support the iterative nature of the image marking process by providing tools that help automate repetitive marking tasks, as well as moving from image to image
2. The tool must be intuitive. By this it is meant that the interface presented to the user is well conceived, ergonomic and visceral. It should support the natural inclinations of the user without the need for excessive learning and remembering of interface controls and concepts
3. The tool will allow accurate ground truth data to be collected by the provision of precision tools in the marking tool application. This will allow for the effortless creation of sound, pixel perfect, accurate markings
4. The tool will be efficient. Because speed is an issue, the system must make it possible to mar a large number of images in a minimal timeframe. Thus, measures must be taken to reduce the number of user interactions with controls such as buttons and menus to the least possible number. To do this, the application will contain intelligence to best predict what best tools are to be used for the marking, and what kind of menus to display, without asking the user beforehand.
5. The tool must be remotely accessible – once installed on a machine, the application should connect to a networked image database, thus providing location transparency. It does not matter if image are marked in Ireland or in the US, the same database tasks will be carried out and underlying networking information will be transparent to the user. When testing and reporting tasks take place, again it does not matter where this is carried out as all test scripts and results are saved in the same place.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

6. The tool must perform efficiently to help speed up the process of marking many images
7. The tool must be platform independent – the application will work efficiently on any operating system

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Chapter 3: Methodology and Requirements

3.1 Introduction

As identified in the previous chapter, image testing and reporting tools do not quite live up to the expectations of developers. By reviewing literature on image algorithm testing, performance measurement and image marking for testing, an understanding has been gained into what developers see as barriers to the use and development of such tools. Thus, in discovering the development requirements that will bring about an improved image algorithm testing system, it is possible to address the current industry needs. The resultant system should accelerate and improve the quality and efficiency of the algorithm testing process for imaging technologies. Focus is placed on image marking and categorization, as this thesis will present a solution to deal with this area in particular.

This methodology chapter will attempt to communicate what it is that needs to be achieved, and what the goal of the proposed system is. In order to elicit the goals of the project, the current testing/reporting methodology must be explored as it currently stands and the inputs and outputs of the procedure must be established. It will be seen how data and evidence gathered during research will influence the proposed requirements.

It should be noted that the early sections of this chapter give an overview of the proposed overall solution to algorithm testing as formulated by this researcher and the partner company. This overview shows the complete framework; henceforth the context of this researchers work is clear.

3.2 Evaluating the Algorithms

As imaging techniques become more widely applied, the need to critically evaluate new methods and algorithms – such as face detection - has been recognised by developers as an area of utmost importance. It is accepted by many in the image processing community that a more rigorous approach to studying the performance characteristics of imaging algorithms is required. As found out in literature review, one of the major criticisms of image processing over the last few years has been due to a general lack of algorithmic

IT Sligo An Institiúid Teicneolaíochta, Sligeach

reliability (Courtney, Thacker, 2001). This has largely been due to the neglect of the important role that statistics and ground truth must play in algorithm development, as well as a distinct lack of advanced, automated testing and reporting tools.

3.2.1 The Testing and Reporting Methodology

Based on evidence found during literature review (particularly in section 5 - "Recommended Solution"), an effective framework for testing and reporting is required. Thus, this work proposes to test algorithms by providing an integrated and intuitive testing and reporting framework.

The potential solution can be viewer as a black box (see Fig.3.1) with the following basic elements:

1. Images – Images with objects at various angles and different locations are collected using a digital camera and are stored in the system
2. Image Algorithm – An image algorithm – such as face detection, red-eye detection, dust detection – is written and is plugged into the system
3. Reports – The system produces a report showing how the algorithm performed. For example, if the object to be detected is a face, the algorithm should find faces. The report will show how many of the actual objects were found by the algorithm, and how many were not found

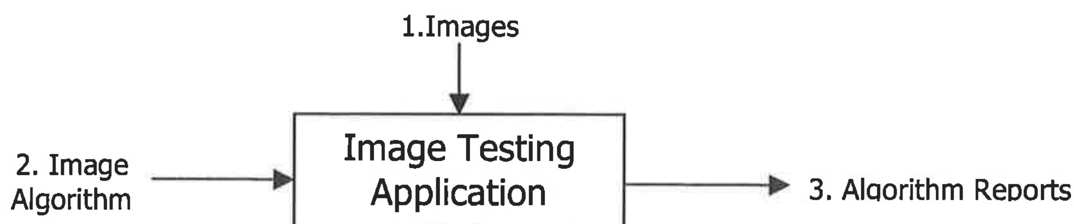


Figure 3.1: System Inputs and Outputs

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Inside the Image Testing Application black box lies the proposed framework, fulfilling all requirements of the testing/reporting process. In short, after images have been collected and stored, and algorithms have been developed, the proposed framework will work through a set of stages with detailed reports as the output.

This thesis is particularly concerned with the process of image marking and establishing ground truth data, without which it is not possible to measure algorithm effectiveness. Within the black box seen in Fig 3.1, another black box for the image marking process can be viewed as follows:

1. Images – Images of objects at various angles and in various locations are collected using a digital camera and are stored in the system
2. Marked Images – Marked images are retrieved. The images hold a set of meta-data specifying locations of all marked features. This data will be used for algorithm testing. Later on, the algorithm must find the marked objects.

Figure 3.2: Image Marking Inputs and Outputs

As mentioned in Chapter 1, "Introduction", the partner company already have an in-house tools for the task of image marking (Fig 3.2). However, the tool is quite limited in scope, so rather than reviewing it and attempting to re-engineer it; it will be left aside for now. Instead, research and requirements gathering will be carried out from the drawing board, in the knowledge that the original tool was put together quickly as a short term solution without sufficient research, rendering it an unsuitable reference point. The in-house marking tool may very well be re-examined later on when exploring interface requirements later on in this chapter, and also when designing the graphical user interface in Chapter 3, "Software Design".

IT Sligo An Institiúid Teicneolaíochta, Sligeach

3.2.2 Discovering the Requirements for the Framework

Taking a look into the Image Testing Application – seen as a black box system in Fig 3.1 - a set of requirements must be drawn up in each case. By understanding the initial requirements, it is possible to break the system up into separate components that will then work together to bring about the new framework – a framework that will carry out the necessary tasks to get our end result.

The need has thus been identified for a framework incorporating an integrated and user friendly environment which supports automated algorithm testing and reporting. To bring about such a framework, a divide-and-conquer stance is assumed, and the current view of the testing and reporting system must be broken down into the following requirement components:

- A facility to import images into an online image database, and to manage image-sets
- Ability to mark image-sets to establish a ground truth for testing purposes
- A facility to run algorithm tests on image sets based on test parameters, and generate reports on algorithm performance.

Support for the categorisation of images and the analysis and editing of image metadata should come about as a by product of the interactive framework. In order to validate and test algorithms comprehensively, a key element of the framework is the image database containing image test sets which possess a high degree of variability in terms of scale, location, orientation, pose, facial expression lighting conditions, etc. Additionally, the framework should bring about a platform independent system that is remotely accessible, in line with the proposed solution as defined in Chapter 2, Literature Review, section 2.5.

3.2.2.1 Use Cases

A basic use-case diagram (Fig 3.3) based on our initial requirements shows that a photographer will store and group images in the system, an image marker will use some sort of system tool to mark images, and an algorithm developer will plug his/her

IT Sligo An Institiúid Teicneolaíochta, Sligeach

algorithms into the system, will run tests, and will create and view reports based on algorithm performance.

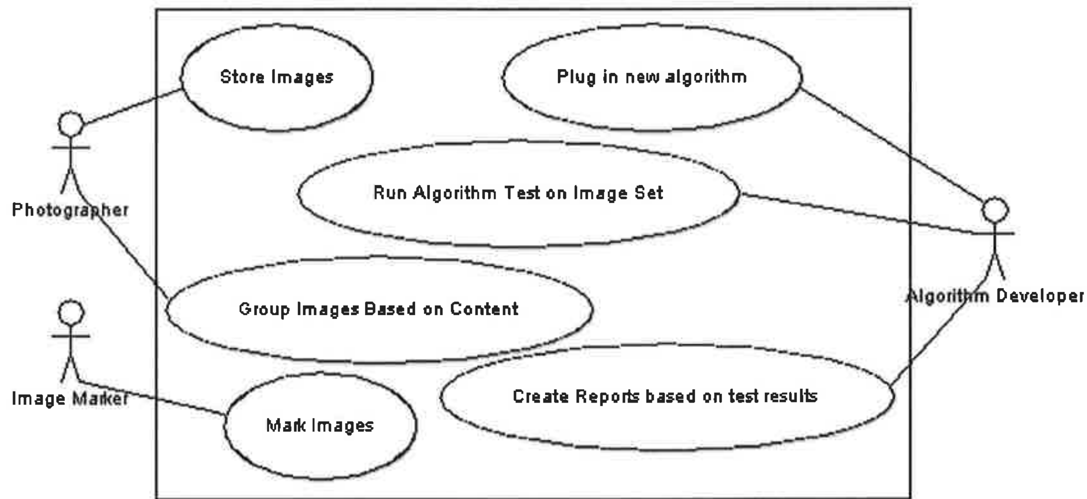


Figure 3.3: Overall Framework Use Cases

With the key requirements in place, it is now possible to propose the overall framework architecture that will be used for testing and reporting. The need was identified for a more succinct, powerful and integrated framework that would allow for the efficient and user-friendly testing of algorithms, right through the phases of database storage and management, ground truth recording and actual algorithm tests and reports.

With all of the above, as well as the requirements, in mind, a methodology has been derived for the testing and reporting of image processing algorithms. The framework proposed comprises of an integrated user friendly environment for algorithm performance evaluation and reporting. As an element of this overall framework, the need for a faster and more efficient set of image testing, marking, and categorisation tools has been identified, therefore the environment should consist of separate application components that will help "divide and conquer" the workload of individuals responsible for the testing of algorithms. The framework should thus promote and bring about the efficient, practical and precise testing and reporting of algorithm performance through the use of improved tools. When deployed in modern algorithm development environments, such a framework should help to speed up testing cycles by quickly testing and identifying problems early in an algorithm development lifecycle. The semi-automatic nature of the framework

IT Sligo An Institiúid Teicneolaíochta, Sligeach

means that algorithm performance is carried out quickly and easily, greatly reducing testing time.

As specified by the partner company; "To validate and test algorithms developed, a comprehensive set of automated/batch image processing tools and a series of specialized image databases must be provided". The framework will thus incorporate components to mark image-sets for testing purposes, run algorithm tests on image sets based on test parameters and generate reports on algorithm performance. Support for the categorisation of images and the analysis and editing of image metadata could also be provided. In order to validate and test algorithms comprehensively, a key element of the framework is the image database, containing images which possess a high degree of variability in terms of scale, location, orientation, pose, facial expression, lighting conditions, etc. Additional requirements specified by the partner company are that the framework may also be platform independent and accessible over the network to allow for testing across different sites or inter-company teams using the same database and application setup, irrespective of operating system and hardware requirements.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

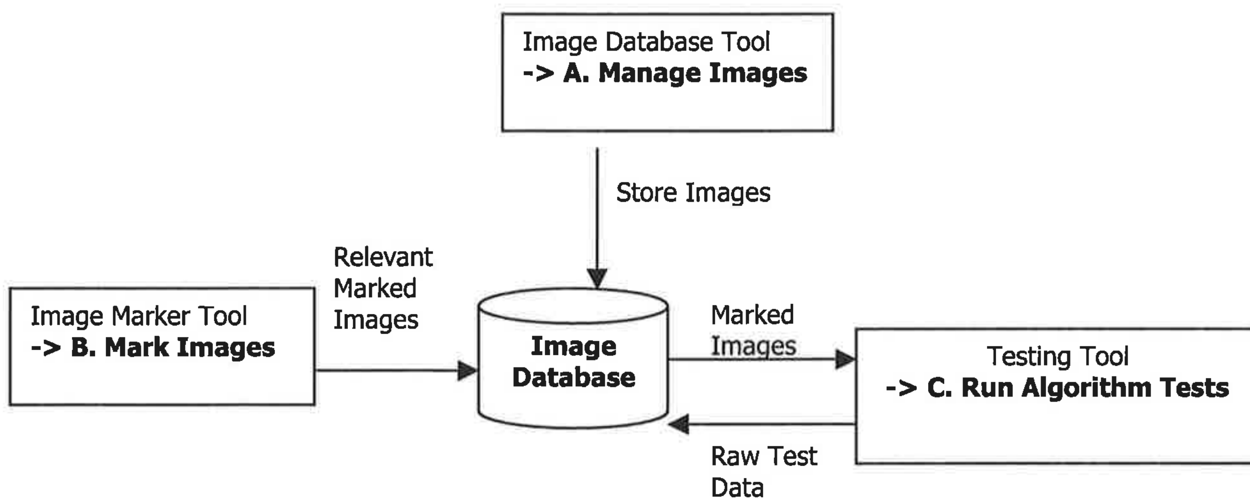


Figure 3.4: Framework Overview

3.3.1 Overview of Framework Components

After researching into and examining existing systems in the area of algorithm testing, an understanding was gained into the way developers work through the stages of algorithm development, testing, and reporting. It has been decided that the most efficient way for a user to carry out image marking, categorisation, testing and reporting would be to provide a complete environment consisting of all these components as individual tools. In other words, at the flick of a button the user must be able to move from the stage of marking into, for instance, the stage of testing. Or from the stage of database management and categorisation into the stage of marking. The actual development of imaging algorithms is the only stage that is not included in this framework, and it is not included because of the broad and unaccountable scope of the algorithm development process. The proposed system endeavours to divide the workload by providing separate tools, thus providing users with a clear distinction between different stages of the categorisation/marking/testing and reporting framework, and using the system should be clear, concise and simple.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

The application is thus composed of three tools, which will be examined one by one, before focusing on the tool that this particular thesis is concerned with.

3.3.1.1 The Image Database Component

This component will store and manage images in the image database. There is the need to be able to import images from a local disk or digital camera, browse through images stored in the database, and search for images by using a query. It is also possible to delete specified images. Image queries could be used to retrieve images that meet a particular criterion. The component should allow for the modification of image properties. Image properties should include items such as width, height, a description, last modification time and so on, as well as a set of markings that identify different features found in the images – that is, if they have undergone the marking process. An example of markings metadata includes items such as a face and two eyes. It makes sense to also somehow group particular images based on their content, and to allow a user to create a group, add particular images to a group and to retrieve images belonging to a particular group.

3.3.1.2 The Image Marking Component

The Image Marking Component is used to mark various features – for example human features such as face, eye, mouth - on the imported images, add properties to features, add properties to the image, and save all image properties to the image database. An intuitive graphical user interface will provide automatic and semi-automatic tools to facilitate the marking and categorization of various image objects in the image data-sets. Depending on the algorithm type, different features will be marked. If the algorithm is red-eye detection, then occurrences of red-eye must be makes in all images. Data gathered through use of this tool will be used later as ground truth data for the performance evaluation of algorithms.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

3.3.1.3 The Testing Component

The testing component allows the user to manage and run image algorithm tests. The need has been identified for the facility to execute tests. This means, an algorithm – such as face detection - is run on the relevant images, coming back with a list of detected faces and their locations. This forms the central point of the framework.

If using the testing tool to run a red-eye detection algorithm on images, we make the assumption that for each image in the image-test-set the location of every red-eye occurrence is identified or marked prior to running the red-eye detection algorithm; this means the ground truth data is readily available. It is important that all data gathered for ground truth are accurate, so the tool used for this task is of fundamental importance. The actual algorithm itself – known as the test code – will be executed on each image. For instance, after running a red-eye detection test, the test results should consist of a list of detected red-eye instances. Then for each image the coordinates of the marked areas should be automatically compared with the coordinates of the areas detected by the algorithm (the ground truth, as established in the Image Marker Tool). If the two match, the algorithm proves to be successful to give [a] a list of matching red-eyes, [b] a list of marked red-eyes that have not been detected and [c] a list of detected red-eyes that do not match any marked red-eye.

After running a test, test results will be available. It seems likely that results will be formatted and displayed onscreen, and later saved. Automating the testing process is very desirable during the development of an image processing algorithm. As programmers develop a new version of an algorithm, it has to be tested to conclude whether a distinct improvement occurred or not. The same image-test-set is used to test each new version.

Algorithm analysis can be carried out, and it is desirable that support be provided within the framework to save the test-result data in the database. With such a facility, new versions of the same algorithms can be tested and compared with previous saved results.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Results can be analysed when a different version of the same image processing technique is developed.

3.4 Discovering The Requirements For Image Marking

This thesis is principally concerned with the design and development of an image marking tool, therefore this section will attempt to explore the requirements of such a tool.

3.4.1 Use Cases

A basic use-case diagram (Fig 3.5) based on our initial requirements for the task of image marking shows that an individual appointed the task of marking images will mark the various features on in image with a graphical interface, and will be able to go to the next and previous images in the group of images to be marked. He or she will also be able to save markings and edited properties to the server. The use case "Add properties to a feature" could be a task such as specifying that a particular marked eye is an instance of Red-Eye.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

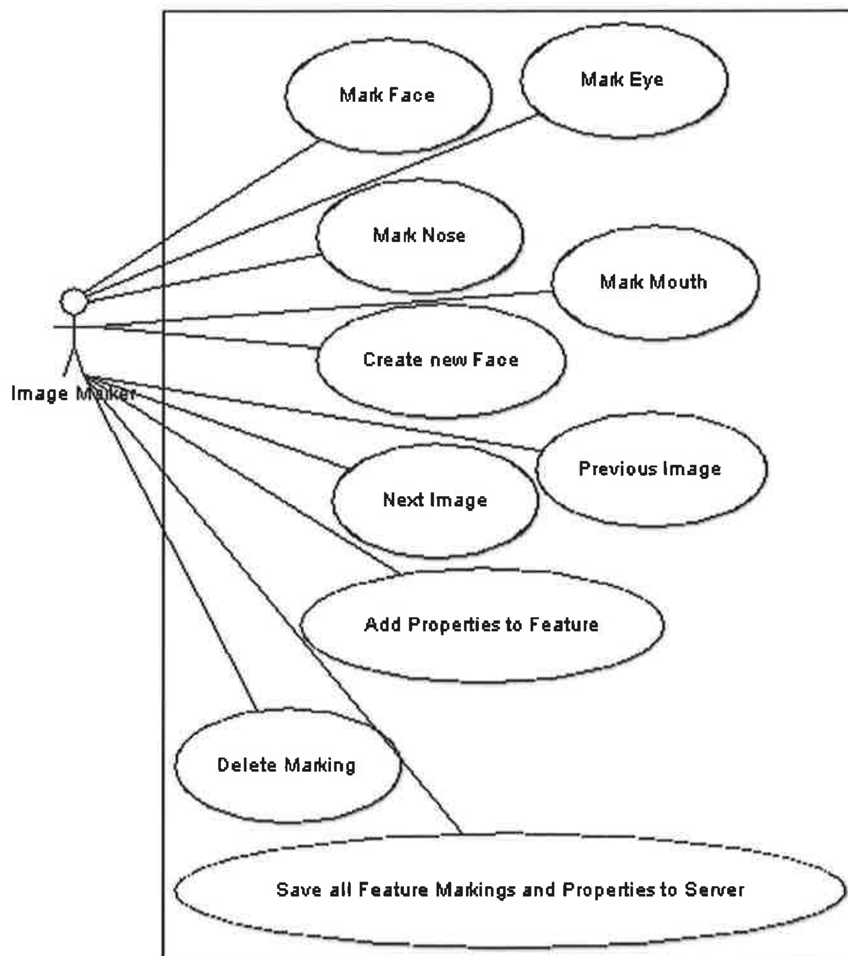


Figure 3.5: Image Marking Use Case

At the moment, marking is seen as a slow, non-intuitive and time consuming task, as found when reviewing literature on the subject: "To comprehensively and thoroughly evaluate an algorithm on real-life drawings, real-life ground truth is highly desirable. However, this type of ground truth is hard to obtain, as it requires manual measurements, which are labor intensive and error-prone. Moreover, manual ground truth input is somewhat subjective and may vary from one human to another" (Dori, WenYin, 1999:3). It is necessary to gather large quantities of ground truth data in order to effectively measure true performance: "Robust algorithms may require huge quantities of data. An algorithm with 99% reliability means an error rate of 1%, which in turn means that hundreds of test images may be required" (Courtney, Thacker, 2001:4). However,

IT Sligo An Institiúid Teicneolaíochta, Sligeach

ground truth data must be gathered for accurate algorithm testing to take place, so a suitable system is certainly required.

Below is a list of the problems identified in Chapter 2, along with the recommended solutions:

Problem	**Recommended Solution**
Ground truth is hard to obtain, as it requires manual measurements, which are labor intensive and error-prone	The envisaged marking tool will be fast and easy to use, automating as many tasks as possible
Manual ground truth input is somewhat subjective and may vary from one human to another	The marking tool will help to make the marking process more objective by using defined procedures and rules for marking images
Expansive databases of real life input must be used, making image marking more complex because the operator must draw images from various batabases	The marking tool will interface directly with a database tool, allowing the fast and easy retrieval and storage of images throughout the process. Navigating through masses of images will become an easy task
Ground truth will always have some residual error rate	While this problem is not eliminated by the marking tool, the goal of this system is to reduce the residual error rates in ground truth data by providing a tool that makes it clear to the user how he/she is establishing ground truth, thus engendering the accurate recording of ground truth data. Weaknesses in the creation of ground truth will be easy to identify should they occur

IT Sligo An Institiúid Teicneolaíochta, Sligeach

As Courtney and Thacker have said; "unless the data is synthetically generated, ground truth will always have some residual error rate (bias and imprecision) due to administrative or instrumental error." (Courtney, Thacker, 2001:4), so while the proposed system will not change this fact, one of its goals is to diminish imprecision due to instrumental error. Also, the system should help speed up the process of manually marking images, as well as categorizing images, with as little input as possible from the user. The proposed tool must therefore be efficient, interactive, intelligent and with some degree of automation where suitable. The tool will also be iterative, managing to mark scores of images in as short a time frame as is possible. The system will put all the necessary tools in the user's hands, and less manual work will be necessary.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

3.4.2 A Framework for Marking

Because this thesis is concerned with the marking tool, this area will be explored in depth in this section. An overview of the application is given in Figure 3.6:

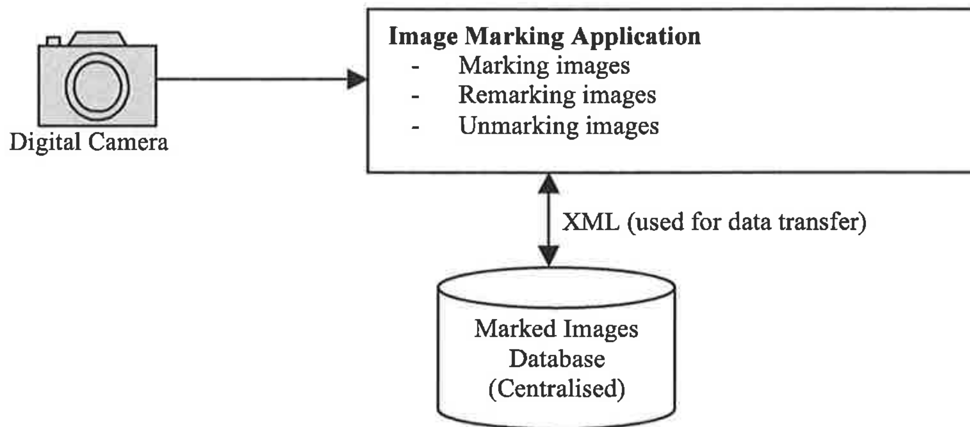


Figure 3.6: Image Marking Application Overview

The need has been identified for an intuitive and easy to use graphical user interface for the marking of image features, particularly faces and face features, as proposed in section 2.5 of Chapter 2 "Literature Review".

The job of a detection algorithm is to detect and/or correct certain features or objects within an image. Therefore, for testing purposes all the relevant features have to be marked on each image in the image-set prior to running the algorithm in order to establish ground-truth data. There is thus a need for automated tools to mange and facilitate (i) the marking of image features such as faces, face features and other zones around a person in an image; (ii) the categorization of images; (iii) the analysis and editing of image metadata. The marking tool aims to assist and speed up the process of image marking by fulfilling these requirements. It should have a user friendly interface that the user will be able to instantly pick up and use effectively.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

3.4.2.1 Interface Requirements

Based on the requirements discovered thus far, the proposed Marking Tool should consist of the following broad interface components:

- A hierarchy of all features in the image – this graphical control could display the parent/child relationship between related features, e.g. face->eye->retina making it easy for the user to see the logical relationship between features
- A graphical view of the current image – this view will show the colour image in full size, and should allow the user to zoom in and out as well as rotate, scroll and resize the image to fit the display
- Properties – a list of the properties for the current feature should be visible. The user should be ale to directly edit some of these properties where possible
- Other controls – Facilities such as importing a batch of images, going to the previous/next image in the list, undoing an activity and more, must be available if required.

Fig 3.7 shows an early concept for the user interface, taken from one of the sketches that were drawn up at the stage of requirements gathering. In this image, in the feature list on the left hand side, an eye is selected as the currently active feature. The user is in the process of marking an eye, and is selecting "red-eye" as the defect.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

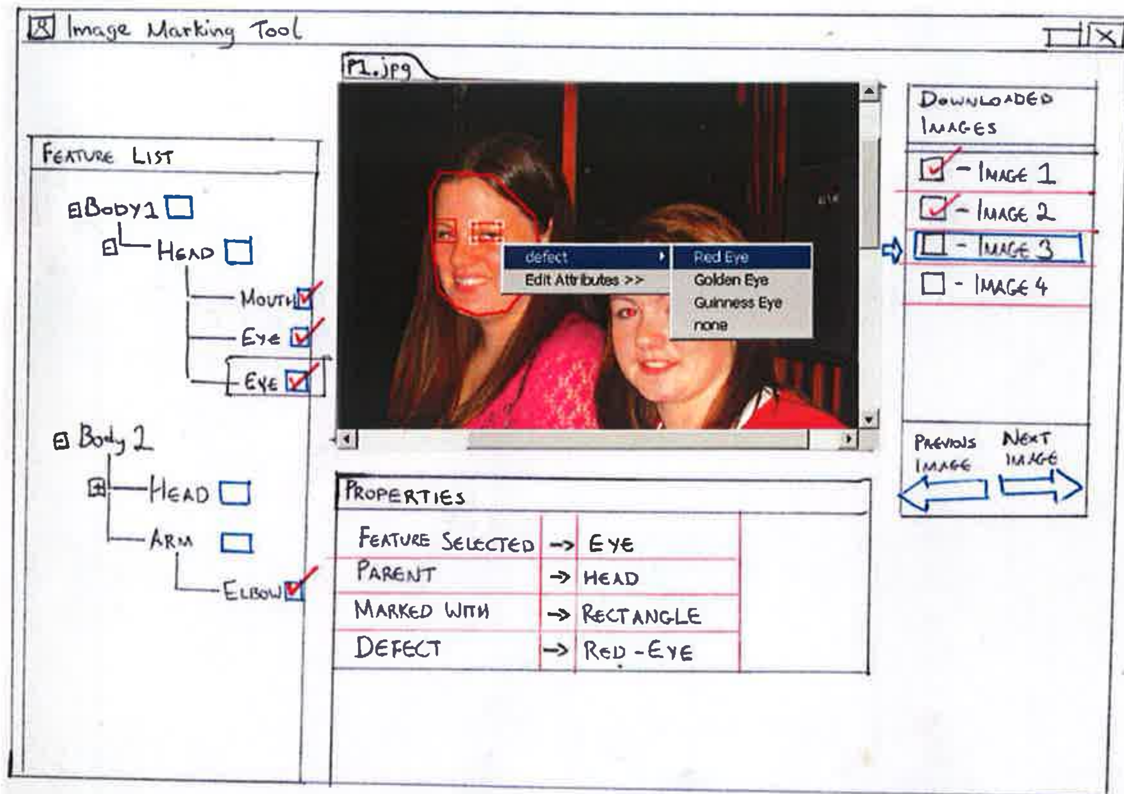


Figure 3.7: Image Marking Tool

3.4.2.2 Tasks to be performed by the Image Marking Tool

Step 1: Images to be marked will be loaded into the marking tool from the database and/or digital camera.

Step 2: The system should automatically know what features can be marked in a particular image, thus when the image is loaded the tools to mark these features will be readily available to the user. For instance, if the user is working with a set of photos showing peoples faces, and the algorithms to be tested are face detection, red-eye detection and nose detection, then tools should be available to mark faces, red- eyes and noses.

Step 3: The user will be able to select the feature he or she wants to mark, e.g. select eye to mark eyes in the image.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Step 4: A feature will be marked using a clear and colourful shape. For instance a red rectangle for a mouth, or a blue oval for a face.
Step 5: When a feature is marked, the markings are automatically stored as metadata for that image. It should be possible to have additional properties as well, such as the skin colour of the face, the name of the person.

It is necessary that the interface be intuitive and easy to use, allowing the user to resize markings, move markings around, undo markings and delete markings. Additional utilities could also be developed to help automate the marking of similar features across a range of images, e.g. if the user knows he or she has to mark all eyes, then it should be possible to ensure the eye tool is automatically selected for the next feature. When the user has finished marking a set of images, the newly marked areas and associated attributes for the image are saved to the image database.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

3.5 Overview of the Framework and Marking Tool

As has been seen in this chapter, the need has been identified for a powerful marking tool and categorization facility to help automate and streamline the process of manually marking images. The needs have been identified through researching current practices in the image algorithm development and testing arena, and the recommended solution aims to fulfill the perceived requirements.

3.5.1 List of Requirements for the Image Testing Framework

R1. The application will provide a complete framework for testing and reporting on the performance of image processing algorithms, consisting of several different modules
R1.1. Providing an Image Marking Tool for the manual marking of image-sets.
R1.2. Implementing an image database server to make the tool available worldwide.
R1.3. Providing an Image Database Tool for storage and categorization of image-sets.
R1.4. Providing a Testing Tool for testing developed algorithms and reporting on their performance.

R2. The Image Marking process will be user friendly

R2.1. With a friendly Image marking interface, easy to use for an individual holding no expert knowledge of image processing algorithm testing practices

R2.2. Providing to the user with coloured graphical tools to mark features in an image

R2.3. With fast and easy access to images and the ability to easily navigate through image sets

R2.4. Forgiving the user by providing facilities to undo tasks carried out incase mistakes are made while marking

R2.5. Allowing for the easy deletion and addition of features

R3. The Image Marking Tool will provide all functionality required to mark particular features in an image, and to add metadata to a feature if necessary

R3.1. Depending on the type of image-sets being marked, the user will be able to mark particular features, as described by the photographer who stored the images in the database

R3.2. If a photographer stored images with red-eye problems, and described facial features as the features of importance, then the user can mark facial features such as eyes, nose, face, mouth using the tool.

R3.3. When a user wants to mark a particular feature, the necessary tool for that feature should automatically become available for marking as specified when the images were stored.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

3.5.2 Goals of the Marking System

In line with the identified needs, the resulting system is to be developed with the following goals in mind:

- The speed at which images are marked will be faster and more efficient
- The accuracy and precision of actual marking data will be improved through the use of intuitive tools
- The iterative marking process will flow seamlessly from image to image, requiring as little effort or input from the user as possible to do tasks such as import images or save markings to an image, or go to the next/previous image

- The user will find it much easier to work through 1000s of images, and because less effort is required, the user will work faster and will be more productive
- The system will be intuitive and easy to learn and work with
- The image marking tool will work successfully with other framework components such as the image database tool and the testing tool to help improve and refine the entire image algorithm testing stage of development.

Using the approach outlined in this document may serve to change the way developers work through the different phases involved in testing and analyzing the performance of imaging algorithms. It should also greatly reduce development times. Imaging algorithms, be they face detection, face recognition, red-eye detection or other, are very complex entities, and consequently their development and testing is a non-trivial and involved affair. The bringing about of simplicity, modularization, conciseness, precision and some degree of automation and intuition to the process of such may very change the way such activities are carried out. It is likely, and quite possible, that accuracy, efficiency and ease of use will be increased greatly. Bottlenecks such as a large image database and non-specific marking tools may thus be overcome.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Chapter 4: Software Design

After understanding the requirements of the Image Marking process, it is now necessary to design software that will sufficiently fulfill all goals as set out in Chapter 3: Methodology. Based on the requirements of image marking, it is clear that the software must provide facilities to mark various objects. If objects such as faces and eyes must be marked, then these entities must be modeled in the system, and must be visually presented to the user in an easy to understand manner. The design chapter aims to elicit the software components and processes that are required for the framework, leaving aside precise implementation details for now.

In short, this chapter explores the possible architecture and design of the Marking Tool in line with the established requirements, including diagrammatic representations of the structure and processes of the system. The characteristics of objects that reside in the system are detailed. This chapter also details objects to a point where code can be directly derived from design diagrams.

4.1 Design at the Architectural Level

When working through the Methodology (Chapter 3), it was discovered that the best approach to the graphical user interface design for an image object was to represent the features contained within the image as a hierarchical tree structure (see Fig 4.1).

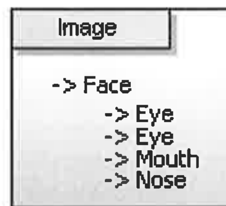


Figure 4.1: Tree Hierarchy

Such a visual representation makes for easy visibility of the logical parent/child relationships occurring within an image. Not only should this hierarchical structure be used for the visual interface representation of objects (items such as physical features, e.g. an eye) within an image, it should naturally be adopted when designing the objects at

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

the architectural level, allowing for easy and direct manipulation/representation of underlying object contents and structure. Thus, when modeling the image object on the client side (the Marking Tool) it seems best for an image to be composed of features. In turn, each feature within the image object may contain sub features, and such sub features may again contain more sub features (all at the discretion of the XML schema in use). Features may also contain properties, or useful information on the nature of the feature, e.g. "red-eye". For instance, an image has a face feature. The face feature has two eyes, a nose and a mouth. One of the eyes has an attribute "Red-Eye". The other eye may have a sub feature retina, and the retina may have an attribute "dust", and so on.

This chapter attempts to find a design that will bring some, if not all, of the aforementioned ideas to fruition. Highlights of the design process are exhibited by viewing documents used when designing.

4.1.1 Conceptual Class Diagram

Fig 4.2 depicts the conceptual class diagram, which helps to briefly represent the Image Marker Tool system architecture in terms of classes involved.

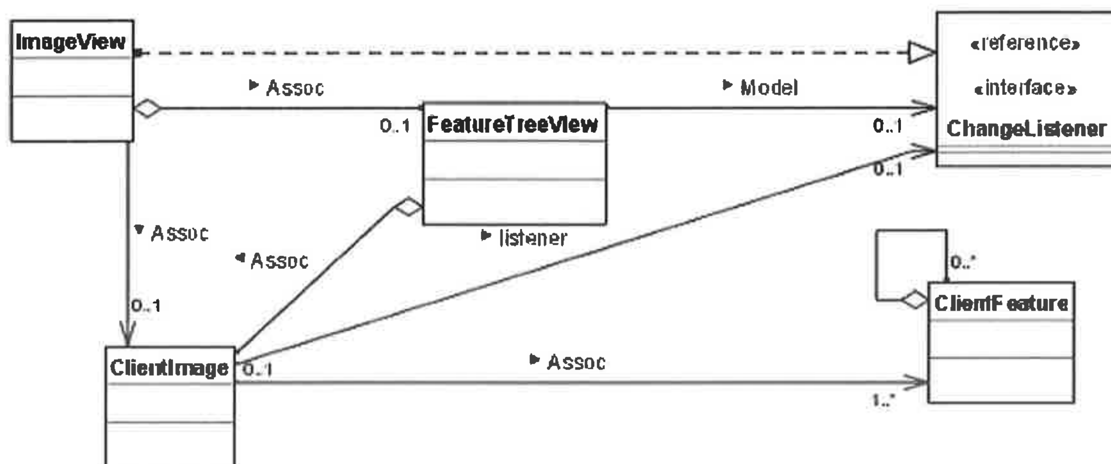


Figure 4.2: Image Marker Tool Conceptual Class Diagram

As seen in the diagram, the Image Marker Tool consists of five main objects – the ImageView class, ClientImage class, ClientFeature class, the FeatureTreeView class and a listener class. The **ImageView** class will be the core class, and it shall comprise the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

graphical user interface, as presented to the user. Part of the graphical user interface will, as discovered in Chapter 3, consist of a hierarchical structure listing features as parents and children - this will be the **FeatureTreeView** class. The FeatureTreeView class will in turn contain an instance of an Image, class **ClientImage**. The class is called ClientImage as it is the representation of a server image and it's metadata as presented to the client in this tool. The TreeView uses a ClientImage instance primarily so it can gain access to the feature metadata (markings and other properties) within the stored image, features that will then be used to populate the tree hierarchy. Thus, features are represented as the class **ClientFeature.** An instance of ClientImage will always contain one or more ClientFeatures, e.g. a ClientImage instance may contain two ClientFeature instances of Face. In turn, one of those ClientFeature faces may contain ClientFeature eye, ClientFeature nose, ClientFeature mouth, and so on. Also, the ImageView class will reference class ClientImage so it can visually display the image.

4.1.2 Class Diagram

It was obvious that the proposed application presented particular challenges regarding class design. However, after researching into available class design literature, it was obvious that "Design Patterns" presented some of the most insightful design ideas. The Model-View-Controller (MVC) architecture appealed to this writer in many ways. "MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input." (Gamma et. al, 1995) Thus, in the design for the Testing Tool, the ClientImage and Client Feature classes can be viewed as the model, with the ImageView as the viewer, and the FeatureTreeView as the controller, as seen in Fig 4.3.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

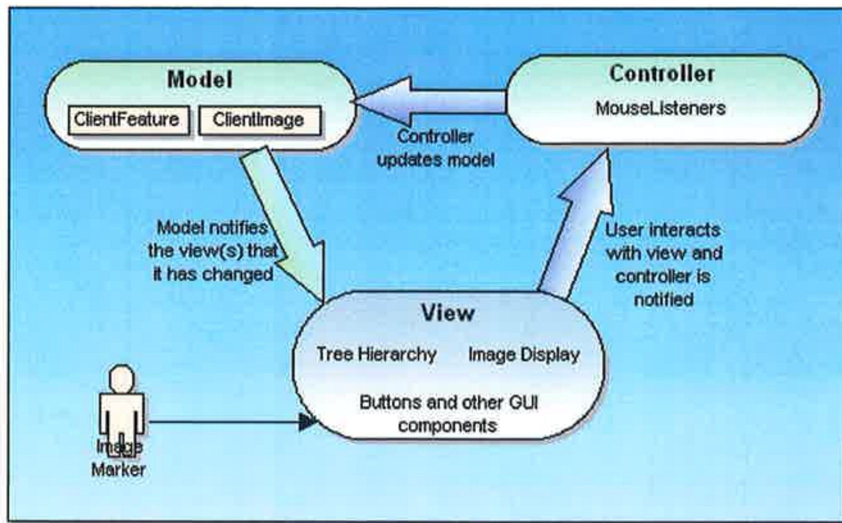


Figure 4.3: Model-View-Controller

According to Gamma et al. (1995), "MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself". Such a design concept requires a way for all objects to stay up to date with changes in the model data structure, and thus a listener class must be used, namely **ChangeListener,** as seen in Fig 4.2. When the structure in the ClientImage model changes, the listener will send out a message to all interested objects informing them that the state of the model has changed. This will fire update methods in all classes, and they will be updated with the latest data. All classes that deal directly with the model (ClientImage) thus need to subscribe to the ChangeListener class.

The likely sequence of events for the Marking Tool is as follows:

1. The user interacts with the view, for instance creates a new eye marking
2. The controller handles the event from the user interface via the ChangeListener
3. The controller accesses the model, adding the new eye marking to the correct leaf node of the ClientImage structure

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4. All parts of the view know that the model has changed, and so the view presents to the user an updated visual interface, taking it's data from the updated model
5. The view waits for further user interaction

Using the MVC architecture for the application as proposed by this thesis should provide the following direct benefits:

- The model will be more robust because it is a separate entity to the view. Components in the view are likely to require alteration at various stages, for instance after usability testing. The model is much less likely to be affected as it is not inherently linked to the view
- Code will be flexible and easier to maintain as the model code does not rely on any of the user interface components. Code reuse is also possible.
- The model can send out a message to all interested parties when it undergoes changes in structure, however it is still unaware of the view, meaning it is truly independent

Before MVC, graphical user interface designs often designed model, view and controller aspects together, resulting in highly coupled code that was inflexible and difficult to maintain and reuse.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.1.3 Sequence Diagram

The sequence in Fig 4.4 highlights the relations between objects in the context of how they will be manipulated by the user.

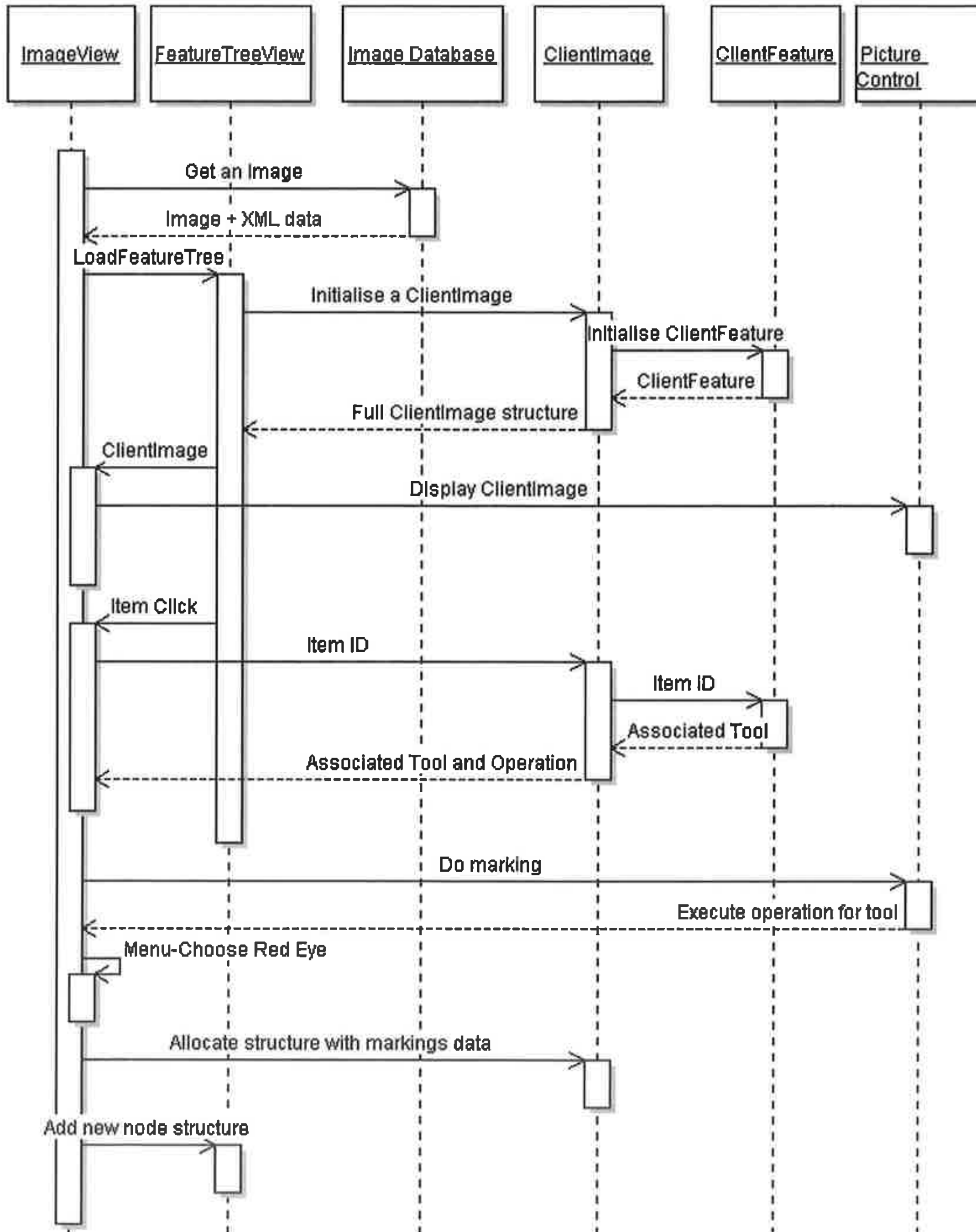


Figure 4.4: Sequence Diagram

IT Sligo An Institiúid Teicneolaíochta, Sligeach

1. The user asks ImageView to retrieve an image from the server. The image is returned, along with XML metadata specifying the hierarchy of features for the image.
2. After an image is loaded, immediately the application will attempt to load the feature tree hierarchy.
3. The FeatureTreeView needs to create an instance of ClientImage
4. A ClientImage is passed to the ImageViewer
5. The ImageViewer class displays the current image in a picture control
6. The user clicks on an item in the treeViewer hierarchy
7. The object id for the clicked item is passed to the ClientImage, and is found within the structure (e.g. face->eye).
8. The corresponding tool and associated properties for this object are passed back (e.g. point tool and red-eye)
9. Now the user carries out a marking on the picture control with the selected tool
10. A menu should pop-up showing the user what tool options are available after he or she has marked the image, e.g. set marking property red-eye to true, set marking property gold-eye to true, and so on.
11. The selected options are stored in the ClientImage structure.
12. The correct node in the tree viewer control is populated with structure data.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.1.4 Overall System Architecture

The Figure 4.5 details how objects as defined in the conceptual model above within the system communicate with each other.

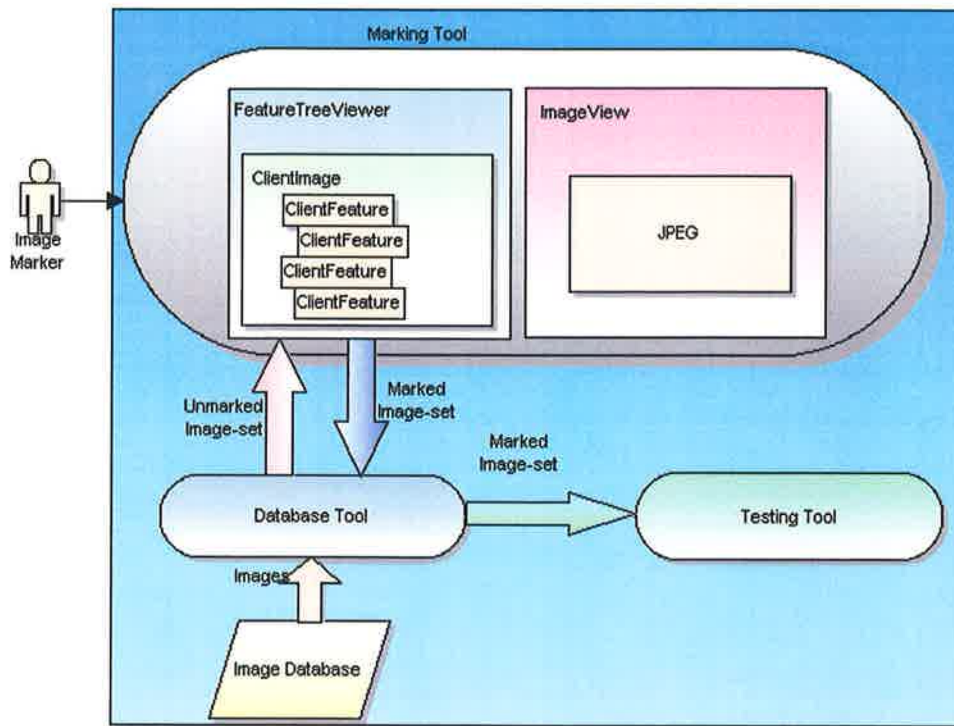


Figure 4.5: Conceptual Model

The diagram emphasises that the Marking Tool can be seen as the client, with the Database Tool as the server. The client is not very thin in this system, all application software and components will need to be installed there. Communication between the Marking Tool and the Database Tool however is quite efficient.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.2 Technical Design

This section describes the technical design for the Image Marking application.

4.2.1 Storage Mechanism – XML Database

The stored information for image sets not only consists of actual photographic images, but also image metadata, such as features marked in an image, image description and various other image attributes that may be defined as the need arises. Because the metadata stored for image sets needs to be flexible, database developers at the partner company decided on using an XML database for this part of the application. For the image testing and reporting framework:

- An XML database is used
- Images are stored physically in the XML database
- The XML document is stored in the XML database

According to The XML Database Initiative (2003), an XML database defines a logical model for an XML document, rather than actually defining the data in that document, as a standard relational database would. Documents are thus stored and retrieved using the logical model. Usually, the model includes elements, attributes, PCDATA, and document order. Also, a row in a relational database is the fundamental unit of storage, with an XML database an XML document could be seen as the unit of storage. Such a database system is a very good choice for the storage of images and their metadata, as is required by the testing and reporting framework for this project. The hierarchical structure of image metadata can easily be represented using an XML document, and it can be quickly transported between the application and the database as the file data can be serialised. Thus, communication between the Marking Tool and the image database will be efficient.

For the Marking Tool, the XML database will be used to get images from the server. The image will be used in the Marking Tool, markings will be added, and various types of metadata could also be added. After this process, the image is sent back to the server with all new metadata intact. Below we see some illustrations to help clarify this process.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

A. Getting an image from the image database (Fig 4.6)

1. The XML document for a particular image is retrieved from the XML database, it contains the full hierarchical structure for all features allowed in the image, and may contain markings
2. The XML document is sent to the marking tool (client) and here, a parsing tool splits up the XML into data components that can be readily used on the client.
3. The data can now be sent to the ImageView object – the main application window for the Marking Tool - where they are used in objects such as ClientImage and ClientFeature.

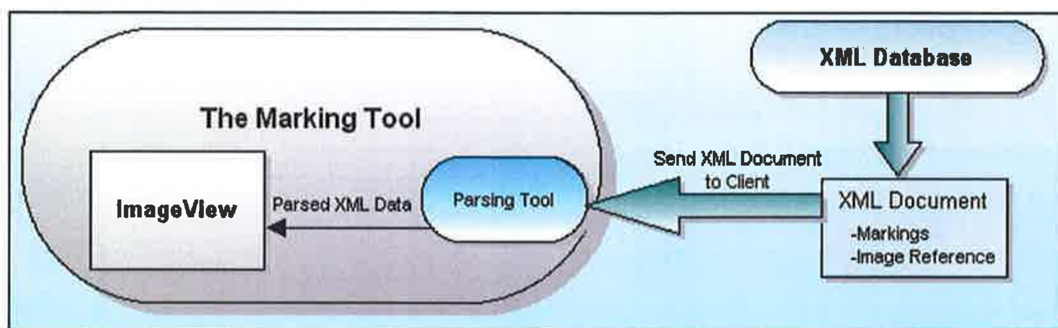


Figure 4.6: Getting an Image from the Image Database

B. Sending an image back to the image database (Fig 4.7)

1. The ImageView object – the main application window for the Marking Tool – sends marking data from ClientImage and ClientFeature objects back to the parsing tool.
2. The data components are collected together and converted to an XML document by the parsing tool.
3. The XML document is sent from the marking tool (client) to the XML image database

IT Sligo An Institiúid Teicneolaíochta, Sligeach

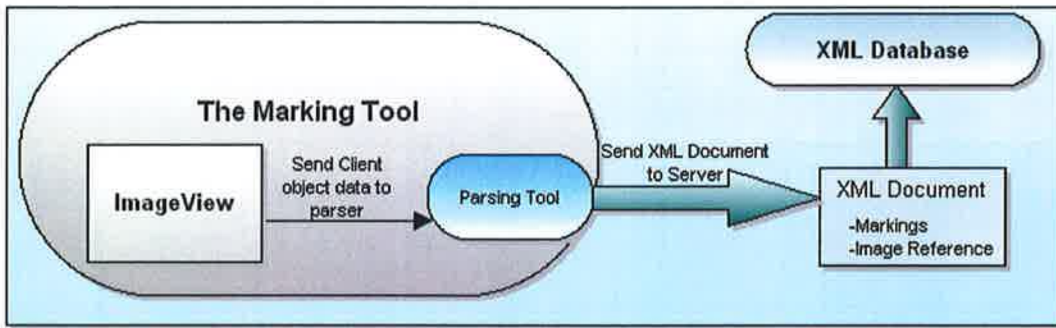


Figure 4.7: Sending an Image Back To The Image Database

In the above situation, it is not specified what technology is going to be used for the parsing tool, how it will integrate with the client, or how XML documents are retrieved from, or sent to the server. Such decisions will be explored in the implementation chapter of this thesis.

When designing the Marking Tool, it was necessary to confer and collaborate with the database development team at the partner company in order to bring about an XML document design that would benefit both ends. In order to do this, a good deal of design work was necessary, and this writer attended various meetings with company developers in order to elicit the real client-side requirements for server data, and to find out how client technical requirements and client/server interplay would impact the XML database schema.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

4.2.1.1 Database Design Exploration

After the client application (the Marking Tool) retrieves an image from the server, it will have access to all image metadata contained in the XML document. However, for the creation of objects such as ClientImage and ClientFeature, and to build up the necessary graphical representations of these objects for the user, sufficient data must be available in the XML document taken from the database. The initial database specification did not fulfill some of the requirements of the Marking Tool – it did not contain the full logical structure for features (ClientFeatures) contained within an image (ClientImage), along with all possible sub features for each feature, it only contained markings. Such a database structure meant that on the client it would be necessary to do a little extra work (illustrated in Fig 4.8):

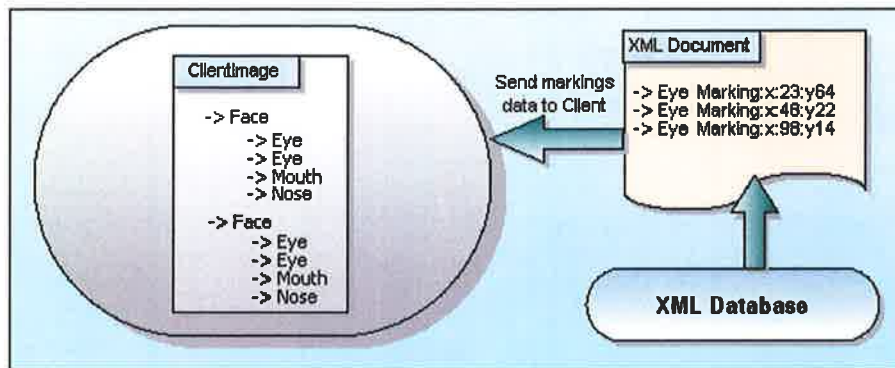


Figure 4.8: Send Markings Data To The Client

- Get the markings XML document from the server
- Create on the client the structure for an image (face, eye, eye, etc.) and create this structure with an empty tree hierarchy.
- Extract the markings from the XML document using a parsing tool
- Determine where in our client structure the database markings should go, then copy them to this location

IT Sligo An Institiúid Teicneolaíochta, Sligeach

There are two obvious problems with the above system:
1) The client must know the exact structure of a ClientImage, and must know how to create this structure. This will have to be hard-coded on the client side. If the storage structure or schema rules change, then code has to be re-written on the client, as it is dependant on the server.

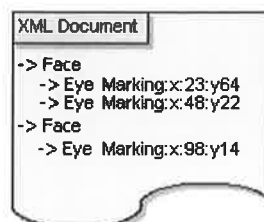
2) Because unmarked features are not stored on the server, the XML document obtained from it does not specify precisely where in the ClientImage hierarchy the markings reside in cases where parent features are not marked (if parent features are marked, then this is not an issue). For instance, if there are only three eye markings on the server, then does the client developer assume that the first two eyes are located in Face 1? Does the remaining eye marking get copied into Face 2 in the structure?

The database team at the partner company has particular requirements for the database – namely efficiency and storage of as little metadata as possible, so any design changes must make sense in the larger scheme of things, not just in the facilitation of the client application data requirements.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

4.2.1.2 An Agreed Solution for Client and Server

A design was agreed upon between client and server developers. Put simply, the use of a compulsory root feature would solve previous problems. Basically this means that the server must – at the very least – store one root feature. In the example we are working with of face and eye markings, it was decided that the most suitable feature to set as a root in this scenario would be the face.



Not only this, but on the server, all rules for the XML storage structure are defined, as well as the storage of actual data. If the rules for storage of data are changed on the server, the client will automatically pick-up the changes and adapt to the new structure. For example, if we are using the application for animals such as cats, the server may add a new child "whiskers" underneath the parent "face". When the client downloads the image with the new XML structure, it will automatically instantiate ClientImage and ClientFeature objects supporting the new structure. Similarly, it will go on to create user interface components – such as the tree view – with the new leaf nodes. Taking the cat example given earlier, a face will be available with children "eye, eye, nose, mouth, whiskers". The client is not dependant on the server structure, and this give a lot more flexibility to the server, as it can freely change the XML schema at will to support different types of marking entities and varying structures. If necessary, a completely new structure may be drawn up, perhaps to facilitate the use of the marking application on vehicles such as cars. In this case, ground truth data may need to be collected for number of wheels and doorknobs present in the photo. To change the scope of the marking application to deal with cars, the server developer only needs to change the XML schema for images of cars stored on the server. An image of a car, when downloaded to the client, should come with a structure like the following:

- ➔ Car body
    - ➔ Wheel
    - ➔ Wheel
    - ➔ Wheel
    - ➔ Wheel
    - ➔ Doorknob
    - ➔ Doorknob

Thanks to XML flexibility, such a change in the use of the marking application should have little or no effect on the client using the image and its XML. This is the optimal solution for the application, and will certainly prove to boost the maintainability, flexibility and robustness of the Marking Tool, as well as de-coupling code and reducing code dependency.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.2.2 Revised Class Diagram

Now that some vital design decisions have been made regarding the architecture and the content of objects used for the Marking Tool, an illustration is shown in Fig 4.9 showing some of the new proposed class contents.

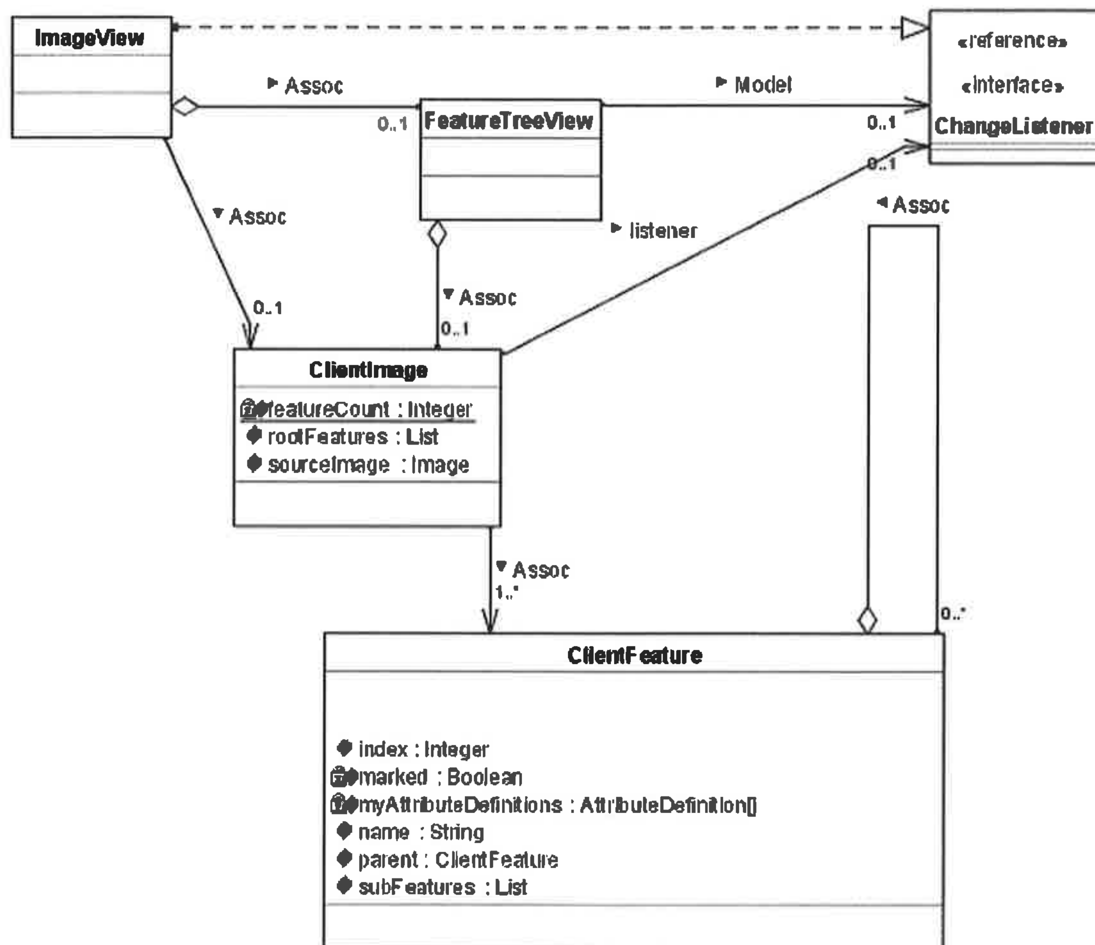


Figure 4.9: Revised Class Diagram

IT Sligo An Institiúid Teicneolaíochta, Sligeach

As seen in the class diagram above, a **ClientImage** object will simply contain:

- featureCount – the number of rootFeatures within the image
- rootFeatures – an list object filled with all the actual root features themselves (ClientFeatures)
- sourceImage –the image itself, taken from the server

An instance of a **ClientFeature** will contain:

- index - An index unique to this feature, could prove to be important when adding/deleting and traversing through the tree structure
- marked - A Boolean value to indicate whether or not this feature is marked (as decided earlier on, a feature need not be marked, it may just exist to support the correct tree structure).
- geometry – The actual marking data, containing the shape co-ordinates for where the image is marked
- myAttributeDefinitions – this array will contain all of the possible attributes this feature can posess, as defined in the XML schema
- name - A name for the feature, for example "face".
- parent – a reference to the parent feature so that backward traversal through the tree is possible
- subFeatures – a list containing this features actual subFeatures

The above class diagram was decided upon after trying out various different approaches and design ideas. While appearing simplistic, this is the very aspect that will make this design work effortlessly for the client application; XML data taken from the server is already in a hierarchical structure, so it is just a case of somehow parsing the data and copying it into the ClientImage data structure on the client. The client objects are then ready to go, with a model that is adaptable, concise and clear. The model, if used with a powerful programming language such as C++ or Java, will allow for complete control over all marked features and attributes within an image with minimal coding. Code

IT Sligo An Institiúid Teicneolaíochta, Sligeach

coupling should be reduced, and maintenance tasks will be made easy thanks to a design that engenders code modularity.

The class diagram seen in Fig 4.9 could be said to be the core of the proposed system – all code in our Marking Tool will be written around this fundamental design. If the classes for this part of the system are well conceived, then the rest of the system should follow the pattern set out by this initial design. The code developed will follow on from the example set out by ClientImage and ClientFeature in modularity and clarity.

4.2.3 Graphical User Interface Design for the Image Marking Tool

As discovered in the Chapter 3 - Methodology, and even more so at various points in design, the success and effectiveness of the marking tool largely depends on the quality of its user interface. Put simply, a good user interface will make the job of making images easy and interesting, and markings data gathered will be more likely to provide for solid and consistent ground truth data. On the other hand, of the interface is not good, the user will be less productive and using the tool will be dull and uninteresting. While it is a subject often overlooked in design, this writer places quite a bit of importance on the necessity of a little Human Computer Interaction (HCI) study, in particular the books of Raskin and Nielsen. According to Raskin (2000:198), good interface design results in some of the following benefits:

- Higher productivity for the user
- Increased user satisfaction
- Faster and simpler implementation
- Simpler manuals

To reiterate the core requirements of this tool; there is a need to mark images quickly and intuitively. Preferably this will be carried out with a good user interface and clearly defined components to show and represent different aspects of the marking process, the images being used, the available tools, settings, and so on.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

It was obvious from the requirements and research what the key tasks of this system would be, but we it was unclear what kind of design would be best, and how it would work. The interface must be user friendly and each component must be clean and easy to use. In order to do this, it will be important to look into how interface controls such as buttons, list displays, drop down selection boxes, and image displays will be used to help speed up use of the system. The use of icons, language and colour are also important consideration. Later on in this section such subjects will be explored.

4.2.3.1 The Previous Marking Tool

As mentioned in Chapter 1 – Introduction, at the start of this project the partner company presented to this researcher a basic marking application that was being used to mark sets of images. The tool was designed to create some basic markings, but did not have a lot of the functionality desired and was quite slow and cumbersome to use. Markings were saved to a basic text file, residing in the same directory as marked images – there was no remote database access or XML storage structure. As part of the new framework, a better and more versatile marking tool would be made available for the marking and categorisation of images. The previous tool, titled Image Feature Marker, was not used much as a focus point while developing its successor as a complete overhaul of the underlying technology and interface was necessary. However it is worth taking a look at the interface for the Image Feature Marker while developing the new interface. Fig 4.10 below shows the Image Feature Marker tool at work.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

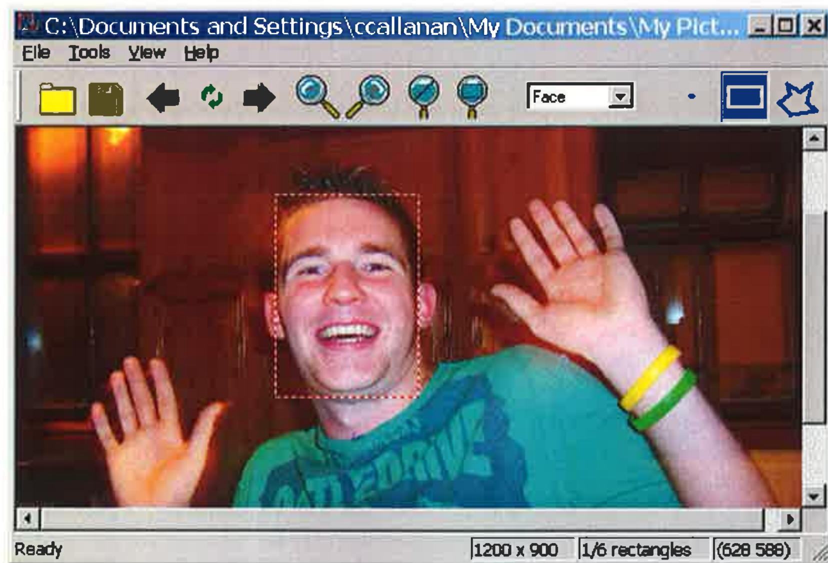


Figure 4.10: The Old Image Marking Tool

The Image Feature Marker was used by the company to manually mark hundreds of images on a day-to-day basis in order to facilitate the testing of algorithms against a ground truth.

The tool's functionality was as follows:

- Open an image file
- Mark a face
- Mark an instance of red-eye
- Mark lips
- View change (zoom in/out, view full size, view best fit, scroll)
- Mark image using a rectangle / polygon / point
- Previous image / next image
- Save markings

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Regarding the interface, obviously much improvement is necessary in order to make marking a more powerful, intuitive, automated and multi-layered process. Some of the following problems exist with the old Image Feature Marker interface:

- It is only possible to mark faces, redeye and lips. If this changes, and for instance noses must be marked, then the application implementation will have to change to support new interface controls
- There is no structure for features marked, therefore the user does not make any distinction with which individual in the photo is being marked
- It is not possible to delete a marking
- It is not possible to undo or redo markings
- It is not possible to copy and paste markings
- It is not possible to resize or move a marking
- The flow of the process from image to image relies on the directory structure, so the user will have to repeatedly change the directory structure to mark new sets of images. This really slows down the job, as the user is not only responsible for marking the images, but also must deal with directories of images and must import them into the tool one by one

The newly developed tool should aim to solve the above issues, as well as to invent an original and interesting tool to enliven up the job of the image marker.

4.2.3.1 Design Iterations

Early on in development a few mock-up prototypes were developed in order to "thrash out" the design. Different design ideas were investigated and attempted, some of which would make it into the final design stages, but many of which were left-aside for various reasons.

Even at early stages of user interface design, it was clear what we wanted from the design, and professional "tried and tested" image processing applications such as

IT Sligo An Institiúid Teicneolaíochta, Sligeach

PhotoShop™ (Fig 4.11), Matlab and Paintshop Pro became ever more useful reference points. For instance, the way PhotoShop™ presents an image to the user with a deceptively simple set of controls and display data was certainly something to aspire to; the selective revelation of the more intricate details and complex tools being the pinnacle of the programs ingenuity. This project, while not dealing with anywhere near as wide a range of components as the aforementioned, still required simplification of the user's job wherever possible, hiding from the user the complexities of the underlying system components.

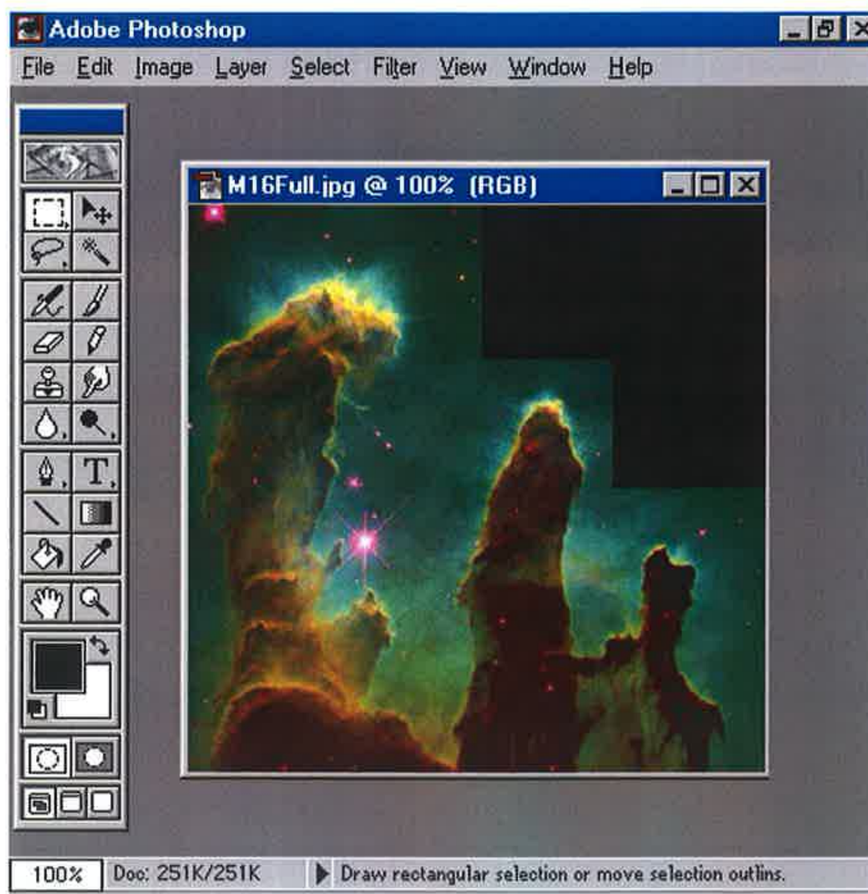


Figure 4.11: Photoshop

As seen in Fig 4.12, an early prototype consisted of a simple interface with buttons for going to previous image and next image, as well as some sort of tree hierarchy to show features. The key components identified were: Tree Viewer, Image Viewer, Image

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Properties, Image List View. What follows is a basic overview of each part, and how the design will fulfil particular requirements that were uncovered in Chapter 3 - Methodology.

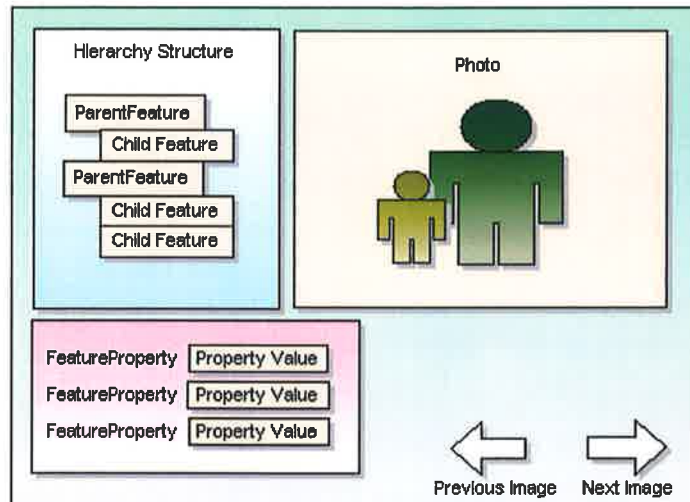


Figure 4.12: Interface Prototype

A. Tree View of Image Features and Properties

The Tree View window will be located at the top-left area of the screen. It will display a list of features, sub-features and properties in the shape of a tree. Where possible, a feature/sub-feature can be expanded and/or collapsed. If the user clicks on the feature, then information pertaining to its properties will be shown in another view (Feature Properties View). Some examples of properties: parent - face, defect – redeye. A user marks a feature by clicking on it in this view, and then moving the mouse to the ImageView – now that the tool for this feature is automatically selected, the user just presses down the mouse to draw the desired marking shape on the image. The shape is finished when the user releases the mouse.

B. Feature Properties View

The Feature Properties View will display all properties for the currently selected feature. It will be located below the photo viewing window. Feature Properties For example, if "eye" is selected in the Tree View, then properties associated with this eye will be visible

IT Sligo An Institiúid Teicneolaíochta, Sligeach

in the Feature Properties View, for example parent = face, tool shape = circle, XY co-ordinates = 231,521, width = 20, height = 10, problem = redeye.

The Feature Properties View should consists of a label for each property type on the left, and an editable value field in the right. The value field could also be a spin control (increase, decrease), text field or combo box.

C. Image Viewer

Quite simply, this viewer will show the currently loaded image. It must be possible to zoom in/out, scroll and resize. Also, some details on size and aspect ratio would prove useful, as well as cursor co-ordinate display.

D. Image List View

This component should show all images in the current loaded set – that is if the user is working with batches of images. If the user is working on a single image, this list is not necessary. However, it may prove useful for quick navigation, and for the user to see where about he/she is in the list.

Later on, after numerous iterations of the above design ideas, a more refined stage of design was reached by eliciting detailed requirements from users based at the partner company, as seen in the following sections.

4.2.3.2 The Feature Tree View

As found in the methodology, as features are marked, they will need to be visibly "active" onscreen in this hierarchical structure which has been termed the Tree Viewer. The main action the user will carry out in this view is the management of features – he/she can add markings to features using a drawing tool in the image viewer, he/she can delete the markings for a feature and he/she can edit the properties for a feature by clicking it and then using the Feature Properties View. In line with the principles set out in the Model-View-Controller architecture (Gamma et al, 1995), this tree structure acts as

IT Sligo An Institiúid Teicneolaíochta, Sligeach

the view for our objects. The tree will actually contain visual representations of all feature objects created on the client, feature objects that were instantiated based on the contents of the XML file downloaded from the server.

Regarding the interface for this component, it is evident that the user requires feedback to clearly state the following:

- The feature name, e.g. face, eye, nose, mouth.
- What feature is currently “active”, and is being marked
- What features are marked / unmarked
- If a feature is expanded - child features are visible
- If a feature is collapsed but can be expanded – has children

In addition to the above, it has been discovered that the user also requires the following facilities:

- Add a new root feature, i.e. an entire new tree based on the XML structure
- Delete a root node, along with all its sub features and their associated markings
- Filter the visible features to only show a) All marked features, b) All unmarked features, c)All features
- Delete the markings for the currently selected feature

To carry out the above tasks, it is necessary to find out how to best incorporate many elements into as small and simple a control as possible. The book “The Humane Interface” by Jef Raskin (2000) was quite a help in this area, as many important decisions now had to be made regarding “look and feel”.

One important idea regarding the use of the hierarchical structure is that as the user creates more root features, and marks more sub features, he/she is creating his/her own chosen structure, and this is part of the user’s content, not just the interface (Raskin, 2000:121). In line with the Model-View-Controller (MVC) architecture principles, these new nodes are part of the view of the model, and are more than just interface controls.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Thus, it is important to communicate to the user that he/she is dealing with the ever changing content layer of the MVC through the view.

Regarding icons, it had to be decided how much of the Tree Viewer interface would use just text, and whether the use of icons would help or hinder the look and feel of this component. Raskin (2000:169) has found that while text often gives the best visual clue, icons are effective when used sparingly – under a dozen at most. When designed, Raskin states that the icons must be visually distinct; large enough to be clear, and must do a good job at representing the appropriate control. Because icons are being considered for the Tree Viewer, consideration has to be taken regarding where to use them, and how many – they should only be used in a few situations where research has shown them to be advantageous. Otherwise, best to stick with text descriptions. Raskin (2000:63) also speaks about affordances and visibility. If the interface being developed here is to place importance affordances, then it will give some visual clue to the user as to the intended use of a particular control. For instance, to expand menu items, the user will see a [+] icon, and will thus attempt to "add on" extra features to the current feature. The result of studies conducted yielded the following design:

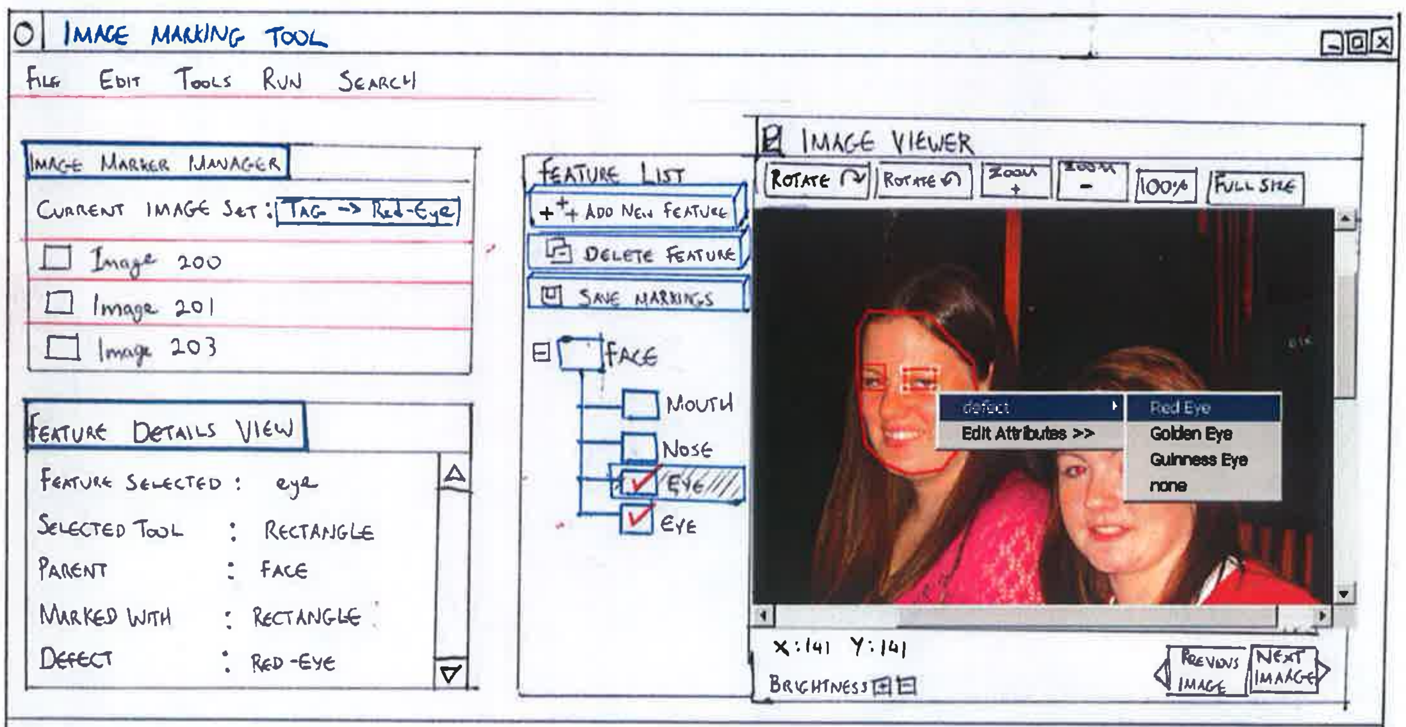


Figure 4.13: New Design Sketch

IT Sligo An Institiúid Teicneolaíochta, Sligeach

As can be seen from the interface screen in Fig 4.13, user familiarity (Sommerville, 2001:330) will play an important role in helping to speed up the use of this system, and it should prove relatively easy to use and learn. Users have encountered a comparable interface in every day applications such as windows explorer and most other hierarchical file navigation interfaces. To the left of each feature icon is a tick box indicating whether an image has been marked or not. Also seen above, after a user has finished drawing a marking on the image, a menu will pop-up allowing the user to instantly set a property for the marking. In this case, there is an instance of red-eye, so the user is choosing to set the defect as red-eye.

Of the heuristic discount usability principles – set out by Jacob Nielsen – the Tree Viewer adheres to the following particulars:

- The use of simple and natural dialogue, easily understood by the user
- Error prevention mechanisms
- Robust and clear error handling techniques when the user makes the occasional error

4.2.3.3 Image Viewer

From requirements, it is clear that the image currently being marked by the user must be displayed in a large viewer, dominating much of the display area. The main operations will be to view the markings on the current image, click on markings to select them and make them "active", drag and drop marking, resize markings, delete markings.

The user must have feedback from this part of the interface to clearly show the following:

- The current XY co-ordinates of the cursor
- The name of the feature currently being marked, and the tool being used to do this, e.g. eye -> rectangle
- Zoom ratio and percentage
- Tool being used – the mouse cursor should change so the user knows what tool is currently selected

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

In addition to the above, it has been discovered that the user also requires the following facilities:

- When all parts of the image are not visible onscreen, the user must be able to scroll up and down or left to right using scrollbars located at the bottom and right hand side of the image
- The user must also have "Next" and "Previous" buttons, allowing the user to navigate through the set of images
- Buttons to allow the user to zoom in and zoom out

Regarding usability for this component, the book "Software Engineering", by Ian Sommerville (2000:330) was quite a useful reference point here. In particular, Sommerville points out that the system should not surprise the user with any new or alien concepts, and should instead use a look and feel that the user is more likely to be familiar with, such as the look and feel of the Photoshop image viewer. Meaningful feedback must be provided when necessary, e.g. telling the user they have reached the end of the list of images. The interface for this view must also be consistent with the other interface components, and the user should be able to see a direct mapping between the features in the tree viewer and the actual markings for those features, which will appear in the image viewer as coloured shapes.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

4.2.3.4 Feature Details View

The feature details/properties view must present to the user, quite simply, the properties associated with the currently active/selected feature. The Property Viewer will be located below the photo viewing window. This viewer displays property values associated with the current property that is selected in the tree view. E.g. if we have selected the feature "eye" in the tree view, then the property viewer will display the corresponding properties for this feature, e.g. the feature has been marked with a rectangle, and it has an attribute defect, which holds the value "red-eye".

Certain properties may be editable, so the property viewer will contain value editors where necessary. After research it was found that edit boxes and spin controls such as

combo boxes are the best thing to use for this part of the program. Certain values will be for display purposes only, and the user will not be able to edit them, for example parent=face. Again, the property viewer is another "view" of the same "model". The data to be presented in this view is taken from the same model that supports the tree viewer and the image viewer, but this time the data we are interested in is specific to a particular feature – the inner data from the feature. For this, care must be taken to ensure the correct set of data is taken from the model, and that it is sufficiently presented. Sommerville (2000:335) states that as developers, we require some knowledge of the user's background before important decision are made relating to the presentation of information. Specifically, Sommerville sets out the following useful questions that help in making such decisions:

1. Is the user interested in precise information or in the relationship between different data values?
2. How quickly do the information values change? Should the change in a value be indicated immediately to the user?
3. Must the user take some action in response to a change in information?
4. Does the user need to interact with the displayed information via a direct manipulation interface?
5. Is the information to be displayed textual or numeric? Are relative values of information items important?

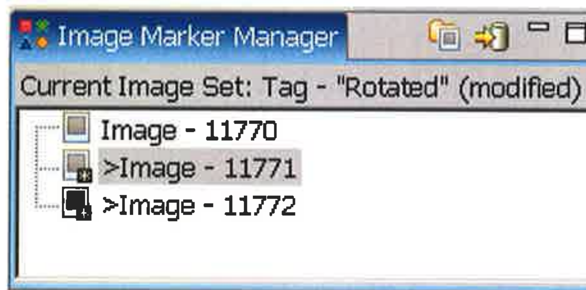
Attributes that are visible in this interface depend on the feature attributes specified in the XML schema for a feature.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

4.2.3.4 Image List View

It is necessary that the user see a list of images currently downloaded from the server, so this view will display the set of images the Image Marker Tool is currently working with. When the user wants to initially mark images, he/she will be able to select the set of images to work with from the images in the image database. Then the images will be loaded into the Image List View. Double clicking on an image in this list loads it into the Tree Viewer for marking.

The Image List view will look as follows:



4.2.3.5 Overall Image Marking Tool Interface

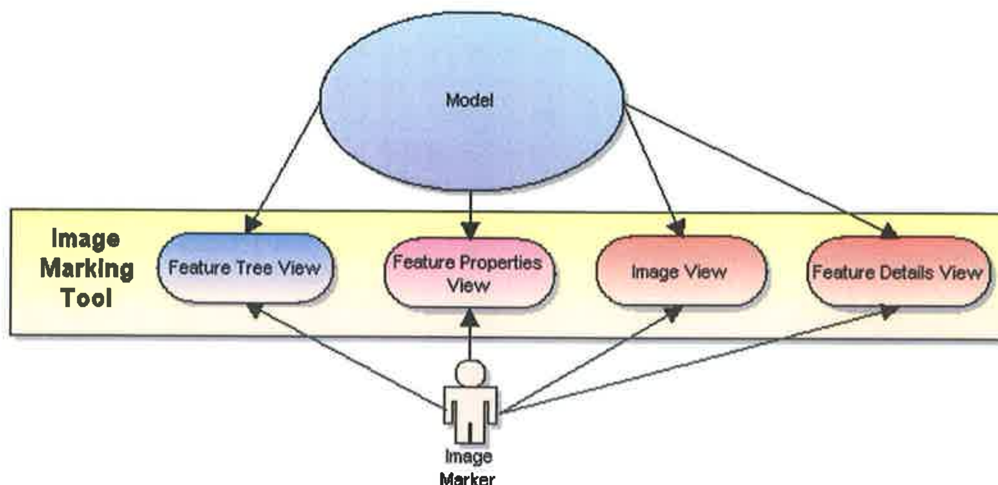


Figure 4.14: Views in the Image Marking Tool

As seen in Fig 4.14 above, the four views that have been conceived are all based on the original model – the data layer – and thus, are all quite simply different views or presentations of the same data.

When designing each of the views, importance was placed on the fact that each view must compliment the other, and in many cases one view will influence the behaviour of another. The following is a list of ideas that will help to ensure all views work together on the same model unitarily and seamlessly:

- When a user clicks and already marked feature in the tree viewer, it will be highlighted somehow in the image viewer, and its properties will be displayed in the feature properties view
- When the marking for a feature is deleted in the tree viewer, it must be instantaneously deleted in the image viewer
- When the marking for a feature is deleted in the image viewer, it must be instantaneously deleted in the tree viewer
- If a feature is edited in the image viewer (resized, moved, etc) then its new properties will be instantly updated in the feature properties view
- If an feature is clicked in the image viewer, then it will be highlighted in the tree viewer, and its properties will be made visible in the feature properties view

I. Separating the Presentation System from the Data

In order to get the overall design correct, some more HCI research was necessary. In particular, Sommerville's (2000:334) notion of separating the presentation system from the data, making it possible to separate the interaction style from the underlying entities that are manipulated through user interface. This basically means that the representation on the user's screen can be changed without having to change the processing mechanism – or the data - at work behind the scenes. The separation of presentation (view) interaction and user interface entities helps to support the desired MVC architectural design style. It is, according to Sommerville, also good software design practice to keep the software interface components separate from the information itself. "The MVC

An Institiúid Teicneolaíochta, Sligeach
IT Sligo

approach, first made widely available in SmallTalk, is an effective way to support multiple presentations of data. Users can interact with each presentation using a style that is appropriate to it. The data to be displayed is encapsulated in a model object. Each model object may have a number of separate view objects associated with it, where each view is a different display representation of the model" (Sommerville, 2000:335).

II. Highlighting, Indication and Selection

While looking into some overall usability principles for the whole application, again Raskin serves as a useful reference. Highlighting, indication and selection will be important elements in ensuring cooperation and consistency between all four views, as well as giving the user vital clues as to the state of the system, so the books of Raskin were consulted about this subject. According to Raskin (2000:105), for the most part, cognitive differences among applications lie in how selections are presented and how the user can operate on them. The following ideas are most useful:

IT Sligo An Institiúid Teicneolaíochta, Sligeach

- Highlighting - Raskin emphasises that the key function of highlighting is to allow the user to determine that the system has recognised a particular object as having special states. For instance in the Marking Tool, if a feature is highlighted in the tree view, then it can now be edited, moved, resized, or deleted in the image view.
- Indication - Raskin points out that indication is vital - the user must know at all times what he/she is pointing at by viewing the interface. For instance, if the user is working on an eye feature in the image view, it should be obvious to the user where this eye is located in the hierarchy, and there should not be any question as to which eye is currently being marked
- Selection – The highlighting that signals selection should be distinct from and more readily apparent than that used for indication. For instance if the user has selected the eye marking and wants to resize it, then there should be very obvious, bright selection points around the current eye marking, leaving no doubt in the users mind as to whether or not the desired marking is selected. Moving from one selection to another, it should be easy to see the changes in selection.

III. Colour

Another area Raskin (2000:172) places importance on is the use of colour. Colour coding can fail if too many colours are used, or if there are too many graphical symbols in each colour. One of the early design ideas for the Marking Tool was to use a different colour for each marking in the hierarchy; this would help to distinguish one marking from another and was initially thought to look nice. However, it proved to be a distracting design idea, and since then it has been decided to employ the colour red for all markings. If a marking needs to be highlighted, this can be easily done without colour change. If a marking needs to be distinguished as a parent, then it can be easily done without colour change – perhaps by marking the line width of the marking thicker.

Regarding the overall use of colour for the application, it was decided that the colour scheme for this program should be in line with the operating system that it is currently running on, so for windows the standard pastel grey will be used for most controls and backgrounds, with blue for window headers. There is no need to reinvent the wheel when is comes to colouring for an application such as this.

IV. Usability Heuristics

In keeping with Nielsen's heuristic discount usability principles, the design of the graphical interface obeyed the following principles:-

(i) The use of a simple and natural dialogue easily understood by the user

(ii) Clearly marked exits visible at all stages of the application's function

(iii) Error prevention mechanisms

(iv) Robust and clear error handling techniques when the user makes the occasional error

(v) Comprehensive help messages

(vi) Graceful recovery from internal error states, (i.e run-time exceptions/errors) and, in the event that such recovery is not possible, the implementation of a graceful

IT Sligo An Institiúid Teicneolaíochta, Sligeach

program exit allowing the user to perhaps save some of the data generated over the course of the session.

4.3 Design Conclusion

The design formulated for the Marking Tool should bring about the efficient implementation of an intuitive and concise piece of software. By taking user concerns into account, combined with the knowledge gained from literature composed by authors such as Raskin and Sommerville, a good design is formulated.

The user interface design is viewed as being quite important for this application, as the actual process of accurate ground truth acquisition is root aim for this thesis. Algorithms cannot be effectively evaluated if the ground truth data is inconsistent with actual physical features within images. As found while reviewing literature on the subject, the imaging industry lacks a sound methodology for acquiring appropriate ground truth data, and adequate methods for matching the ground truths with the recognized graphic objects" (Dori and Liu, 1999:1). But ground truth is known as being hard to obtain, requiring manual measurements that are labour intensive and prone to inaccuracy. Not only this, but ground truth input may vary from one human to another.

The design for the Marking Tool takes into account such problems, and allows for not only accurate but also fast ground truth acquisition. The design for the tool allows a user to:

- Mark various features (e.g. face, eye, mouth) on an image
- Add properties to features
- Add properties to the image
- Save all image properties to the image database

Data gathered through use of this tool will be used later as ground truth data for the testing of algorithms, and it is envisaged that such data will be of a more consistent and

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

accurate nature than that collected using older marking practices, such as the “Image Feature Marker” application that was being used at the partner company.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Chapter 5: Implementation

As identified in the design chapter, there is a need for an Image Marking Tool that has a user friendly, intuitive interface. The user should have the facility to import images from an online image database and mark image-sets to establish a ground truth for testing. The overall goal is the development of an image marking system that fulfils all design requirements and gives all the desired marking functionality to the user at the touch of a button.

The aim of this chapter is to communicate the means through which the Image Marking Tool design was implemented. This chapter does not make an attempt to cover the entire Image Marking Tool architecture, as there are in excess of forty classes used for the making tool alone. However, this section will tackle some of the fundamental areas of the architecture, such as the data model, interaction between the data model and the server XML data, user interface components and interaction between the user interface components and the underlying data model. A working knowledge of object oriented programming is assumed in an attempt to reduce the explanatory content of the chapter.

5.1 Approaching Implementation and Testing

Because of the complexity of the problem and the need for integration of many different disciplines ranging from data modeling to user interface studies to white box testing and analysis, the implementation of the Marking Tool was split up into the following stages:

- Selection of programming environment
- Development of the data model
- White box testing of the data model
- User interface research and development
- Integration of user interface components with data model
- Black box usability tests
- Interface redevelopment

IT Sligo An Institiúid Teicneolaíochta, Sligeach

The testing stages of development mentioned above will be detailed in Chapter 6 – Software Testing, while the implementation stages will be explored within this chapter.

5.1.1 Exploration of Programming Environments and Other Resources

Before starting into the actual implementation, it was important to decide what programming languages and development environments best match the developmental requirements and goals of the project.

A. Programming environment criteria

Prior to selecting a particular technology for the implementation of the Image Marking Tool, it is vital to determine a few simple but definite criteria as follows:

- Development productivity – some development tools allow for fast development and give the developer many aids that help to automate and speed up code creation and maintenance
- Efficiency of the resultant application – certain programming languages are known to produce fast efficient code (for example C) whereas some languages end up with slow processing time (for example Microsoft Visual Basic).
- Maintainability of the resultant application – if the language used is easy to understand and encourages developers to write modular code with distinctive components, then the code will be easy to edit and review should the necessity arise.
- Portability – can components of the final application be ported to a different application, and can the code be used on different operating systems, different development environments and in

These will be the criteria against which the development languages will be accessed.

B. The potential programming environments

Language 1: C++

IT Sligo An Institiúid Teicneolaíochta, Sligeach

The object oriented C++ language, which has an International ANSI/ISO Standard, is a highly capable and efficient programming language that is certainly capable of implementing the Marking Tool if used correctly. C++ is a superset of the C programming language, with advanced features that allow low-level access to memory. MFC (Microsoft Foundation Classes) also allow higher level programming development with C++. The model-view-controller architecture could certainly be implemented using MFC, where user interface controls such as tree control could be used for the model, a list control for properties, and a dialog for image display. For the data model C++ objects could be created and with the added benefit of C++ memory management techniques, code could prove quite efficient.

Language 2: Java

Java is another powerful language. Having first emerged in 1994, this modern development language has grown exponentially in the last decade, with many add-ons and new Java technologies being created to further extend Java's range of use. A cross-platform language, Java applications will run on any operating system so long as the operating system runs a Java Virtual Machine. Java is an object-oriented language similar to C++, but is somewhat simplified in order to eliminate language features that cause common programming errors.

There are many different Java IDEs (Integrated Development Environments), all of which offer different development tools that help programmers create more advanced Java applications at a faster rate. Well known IDEs such as Eclipse, NetBeans, JBuilder and IntelliJ make available to the developer many modules, such as code debuggers, form editors, object browsers, CVS functionality, and integration with other languages. For the Marking Tool, it would appear that the powerful Eclipse IDE offers the best choice of functions. With Eclipse, the developer can build an application that easily incorporates different plug-in components from a variety of vendors, meaning a much richer selection of possibilities available. Eclipse also uses the concept of different perspectives, allowing for the creation of views of the same data, or different interfaces to carry out different tasks on the same data. This idea fits in with the model-view-controller architecture, and

IT Sligo An Institiúid Teicneolaíochta, Sligeach

could very well facilitate a way for the image testing and reporting framework to achieve the notion of a single application with different stages of use.

C. Exploring Implementation Techniques

Before choosing a language and diving into the implementation of this large tool, some time was spent exploring possible implementation scenarios. Because C++ at first looked like the more likely candidate, a lot of time was spent implementing the data model using, as well as reading literature on programming tools that would allow for the fastest user interface development. After this, attention was drawn to the Eclipse Java development environment, which had previously been overlooked as a realistic platform for development due to its infancy and lack of documented use.

Technique 1: Write the data model in C++, using Visual C++ with MFC for the GUI

With this technique we would have access to a rapid application development environment that has a comprehensive graphical toolkit in the form of the Microsoft Foundation Classes (MFC). In addition, the following advantages are applicable:

- Wealth of MFC Windows classes readily available that can be adapted for use with the Marking Tool
- Using the C++ language with its memory management facilities means the end product will work efficiently
- The ability to directly program the Win32 API to speed up complex tasks
- MFC provides a Document/View framework that facilitates the creation of Model-View-Controller based architectures

Technique 2: Use the Eclipse development environment to create the data model and user interface, using Java compatible open source facilities where necessary From the project specification research it is evident that developing the application to be cross-platform would reap a few benefits. Particularly, it was obvious that the tools being developed as part of the Image Testing and Reporting Framework would be used on a

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

variety of operating systems depending on the preference of the algorithm developer or tester. The likelihood of heterogeneous use, as well as the necessity and definite advantage of using world wide open standards mean that Java and its associated technologies may provide the most attractive option.

Additionally, a GUI is required and must provide an intuitive and familiar interface to ensure the full potential of the marking tool is realised. Hence, a graphical interface to drive the tools is required to integrate all the parts and provide ease of use.

5.1.2 Selection of Programming Environment and Development Technique

The Java programming environment – for reasons of portability and familiarity – was chosen as the primary tool for implementation. The Eclipse development environment was a particular deciding factor, and made available a powerful and broad range of tools unseen in any other IDE. Use of Eclipse will require the mastering of a whole new Java development environment and an array of new development tools and widgets. This will undoubtedly enhance skills in programming techniques and should therefore be regarded a positive, and not a drawback.

After creating a few sample application using Eclipse, it was found that Eclipse, when combined with the JDT (Java Development Tools) was quite a fast and easy to work with development platform, offering features like a syntax-highlighting editor, incremental code compilation, a thread-aware source-level debugger, a class navigator, a file/project manager, and useful interfaces to standard source control systems such as CVS (Concurrent Versions System). Not only this, but Eclipse has advanced code refactoring and support for code unit testing with a plug-in known as JUnit.

One of the most attractive and interesting features of Eclipse however is that it is platform and language neutral. Along with support for languages such as Java and C/C++, there is also upcoming support for languages as wide ranging as PHP, Ruby, Python, Eiffel, and even Microsoft's C#. But the most useful aspect of Eclipse is the plug-in architecure and rich APIs supplied by the Plug-in Development Environment,

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

allowing constant extensions of the Eclipse functionality. Basically a Plug-In is Adding support for a new type of editor, view, or programming language is remarkably easy, given the well-designed APIs and rich building blocks that Eclipse provides.

5.2 Database Interaction and Interfacing with the Client

As discussed in Chapter 4 – Software Design, the decision was made to use an XML database for the Image Marking Application. The first step when implementing the client application was to consider the best options for database interaction.

5.2.1 Client Side Requirements

It is known that the objects developed on the client will work with data obtained from the server. On the server, data will be stored in XML format. After an image is downloaded from the server, the client must somehow parse the XML data to extract the relevant information. When the relevant information is obtained, it must be plugged into client side objects, such as ClientImage and ClientFeature, as discussed in design. Taking a look at what objects are required in the client application, it became clearer what kind of XML structure was required on the database side.

5.2.2 Database Specification

As mentioned in Chapter 4 – Software Design, section 4.2.1.1 - it is necessary for the server to hold at the very least a list of root features. Both server and client only need to agree on what a root feature *is* (this will be specified in the XML schema), and once this is established, the server data will mesh easily with the client data structures. The client simply looks at the data on the server, root feature by root feature, and copies the data from each server root into a newly created root feature on the client, known as a ClientFeature - see Fig 5.1. The data on the client must be as close as possible in structure and purpose as that on the server. Data that deals with the client implementation must be kept away from the server structure to lessen the possibility of code coupling and dependence.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

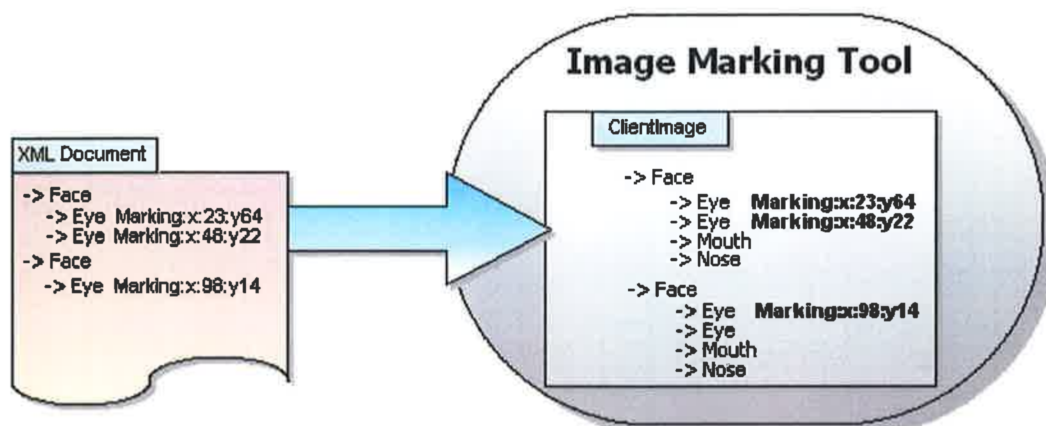


Figure 5.1: **Extracting Data from the XML Document**

The first version of the database used for this application possessed some characteristics that did not work well with the above client object design. Thus, a second version of the database was developed. The new database was an XML database, allowing for the retrieval and storage of images along with associated XML metadata. In order to successfully derive all necessary client objects from server data or - put more simply - to take the markings from the server and plug them into the client application, the following elements would be required:

A. ClientImage

An instance of ClientImage will contain the full structure for all markings in an image, as well as image attributes.

➔ rootFeatures – a list of all root features in this image. If using a schema where face is defined as the root feature, then all root features will be faces.

➔ Attributes – a ClientImage will contain a list of attributes that the client user will be interested in. Attributes specified will be things such as:

- Image ID
- Width
- Height
- Image Description

- File Name
- Created
- Last Modified
- Tag

➔ featureCount – the number of root features within this image

B. ClientFeature

An instance of ClientFeature will contain the actual marking for that feature, if it has been marked, as well as other data that must be gathered for algorithm testing purposes.

➔ subFeatures – a list of all child features in this feature. The schema will have defined the rules pertaining to what children are allowed.

➔ Attributes – a ClientFeature will contain a list of attributes. Different features will have different attributes. In our current example, a face should have attributes such as:

- Person's Name
- Colour of Skin
- Age group
- Defect - An eye feature has this attribute which allows the user to choose from a list of potential XML defined defects such as red eye, golden eye, or whatever defect the algorithm in question is being tested for:

➔ Name – the name of the feature, for instance face or nose.

➔ Attributes – a ClientFeature will contain a list of attributes that the client user will be interested in. Attributes specified will be things such as:

➔ Geometry – the actual marking data. Usually this will be in the form of a point, ellipse, rectangle or other polygonal shapes as defined by the user. This data must allow the application to pinpoint exactly where and how to draw the marking on top of the onscreen image.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

In the second version of the database, the XML structure on the server was tailored to provide the data needed to effectively run the marking tool. The object contents described above will be common to both client and server. On top of this, an additional XML solution was derived for client/server database interaction based on the data model. This solution was given the name PFML – photographic feature markup language – and was created to bridge the gap between the client application and the server data, and to allow for the ease of data transportation between the two. In short, the client will download an image along with its PFML content. Then the client will have the facility to use the PFML API functions to retrieve useful data from the PFML content, such as image attributes and image markings.

5.2.3 PFML Server Side Implementation

5.2.3.1 Image Markup

PFML is an XML based language to describe features and artifacts in a photographic image. This takes the form of a feature hierarchy, inside which features can be embedded in other features. Example:

```
<pfml version="1.0">
  <face>
    <attribute name="skintone">white</attribute>

    <!-- face outline -->
    <geometry>
      <LinearRing>
        <Point x="x0" y="y0" />
        <Point x="x1" y="y1" />
        ...
        <Point x="xn" y="yn" />
        <Point x="x0" y="y0" />
      </LinearRing>
    </geometry>
```

IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
    <!-- first eye -->
    <eye>
      <attribute name="defect">redeye</attribute>
      <geometry>
        <Circle x="123" y="233" r="10" />
      </geometry>
    </eye>

    <!-- second eye -->
    <eye>
      <attribute name="defect">redeye</attribute>
        <geometry>
          <Circle x="174" y="228" r="9" />
        </geometry>
    </eye>

    <!-- nose -->
    <nose>
      <geometry>
        <LinearRing>
          <coordinates>....</coordinates>
        </LinearRing>
      </geometry>
    </nose>

    <!-- mouth -->
    <mouth>
      <attribute name="state">open,smile,teeth</attribute>
      <geometry><LinearRing><coordinates>...</coordinates></LinearRing>
        </geometry>
    </mouth>

  </face>
</pfml >
```

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

5.2.3.2 Storing image markup in the image XML file

On the database server side, there exists an image storage schema. The PFML markup structure, as described 5.3.3.1 will simply sit in the middle of the image storage XML file as follows:

```
<image version="1.0" id="123" width="3234" height="1200">

  <!-- Standard attributes -->
  <created>1032000000</created>
  <lastmodified>1033000000</lastmodified>

  <title>This is my title</title>
  <description>...</description>
  <creator>J Bloggs</creator>
  <date>25 Aug 2004</date>
  <filename>original-filename.jsp</filename>


  <!-- tags -->
  <tag name="redeye" />
  <tag name="indoor" />
  <!--.. repeated as necessary -->

  <!-- User defined attributes, all under user element -->
  <user>
  <indoor>true</indoor>
  </user>

  <!-- EXIF data from image header, all under exif element -->
  <exif>
  <shutter>100</shutter>
  <aperture>5.6</aperture>
  </exif>

  <!-- Computed attributes, all under computed element -->
  <computed>
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
  <histogram> tbd </histogram>
  </computed>

  <!-- Feature markup -->
  <pfml version="1.0">
    ... feature markup ...
  </pfml>
</image>
```

The above XML file is what will be downloaded to the client as an "image".

5.2.3.3 Attributes

As seen above section, the image storage XML schema uses various attributes. Standard attributes are based on the Dublin Core standard which defines 14 attribues which each document should have. For this application, attributes such as date, title, description and creator may be used. More information on Dublin Core Attributes can be found in Appendix A.

5.2.3.4 Elements and attributes in the schema

PFML provides a schema with a set of rules for the storage of images on the server. Rules for the feature tree hierarchy are specified with regard to allowed children and max number of children of any type allowed, for instance:

- A ClientImage will contain a list of attributes that the client user will
- A feature element comprises zero or more attribute elements
- A feature element comprises one or more geometry elements as defined by the schema
- A feature element comprises zero or more feature elements (sub-features) as defined by the schema
- Geometry element comprises one of (and only one) Point | Circle | LineString | Polygon geometries.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

The possible geometry types include point, rectangle, linear ring, line string and ellipse. See Appendix B for a more detailed specification on geometry types.

5.2.3.4 A look at the XML schema itself

As already discovered, the schema defines the properties of an images and its features. It also defines relationships between features (eg face can have at most two eyes):

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

        <xs:element name="attribute">
                <xs:complexType>
                        <xs:attribute name="name" type="xs:string"/>
                </xs:complexType>
        </xs:element>

        <xs:element name="eye">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element              ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="mouth">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element              ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="nose">
                <xs:complexType>
                        <xs:sequence>
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
                        <xs:element              ref="attribute"
        minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="face">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="attribute"
        minOccurs="0" maxOccurs="unbounded"/>
                        <xs:element   ref="eye"   minOccurs="0"
        maxOccurs="2"/>
                        <xs:element   ref="mouth"   minOccurs="0"
maxOccurs="1"/>
                        <xs:element   ref="nose"   minOccurs="0"
maxOccurs="1"/>
                    </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="head">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element              ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                        <xs:element   ref="face"   minOccurs="0"
maxOccurs="1"/>
                    </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="body">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element              ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                        <xs:element   ref="head"   minOccurs="0"
maxOccurs="1"/>
                    </xs:sequence>
                </xs:complexType>
```

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

```
        </xs:element>
        <xs:element name="dust_spot">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element                ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <!–pfml element encapsulates all feature markup -->
        <xs:element name="pfml">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element    ref="eye"    minOccurs="0"
maxOccurs="unbounded"/>
                                <xs:element   ref="body"   minOccurs="0"
maxOccurs="unbounded"/>
                                <xs:element                ref="dust_spot"
minOccurs="0" maxOccurs="unbounded">
                                </xs:element>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <xs:element name="image">
                <xs:attribute      name="version"      type="xs:NMTOKEN"
fixed="1.0" />
                <xs:attribute name="id" type="xs:integer" />
                <xs:complexType>
                        <xs:sequence>
                                <xs:element                ref="attribute"
minOccurs="0" maxOccurs="unbounded"/>
                                <xs:element   ref="pfml"   minOccurs="0"
maxOccurs="1"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
```

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

```
</xs:schema>
```

5.2.4 Building the Data Model on the Client

In order for the client application to make use of the PFML and the functionality it provides, the PFML library is installed on the client. After an image is downloaded from the server, the API provides the client with access to various classes and methods that interface with the PFML data attached to the image. This allows the client to carry out many useful tasks, such as parsing the entire markup structure to gather metadata about the image. The result of this will be the creation of a ClientImage instance on the client, populated with all attributes and all markings as they are found on the server, as seen in Fig 5.2 below. In this section, the set of steps carried out on the client to build up the data model will be explored.

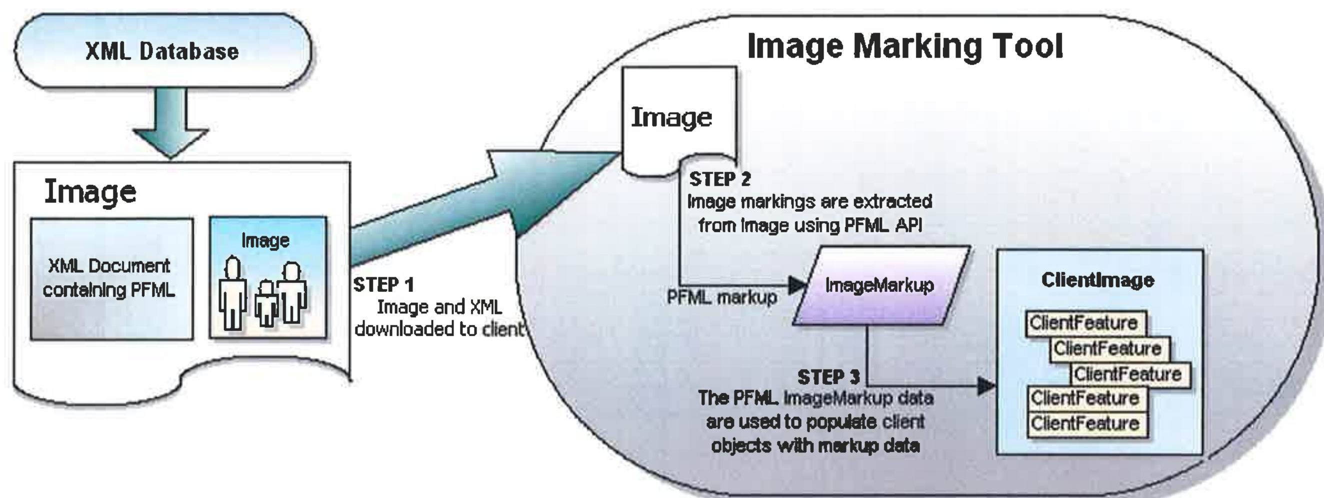


Figure 5.2: **Using XML Data to Create Client Side Data Structures**

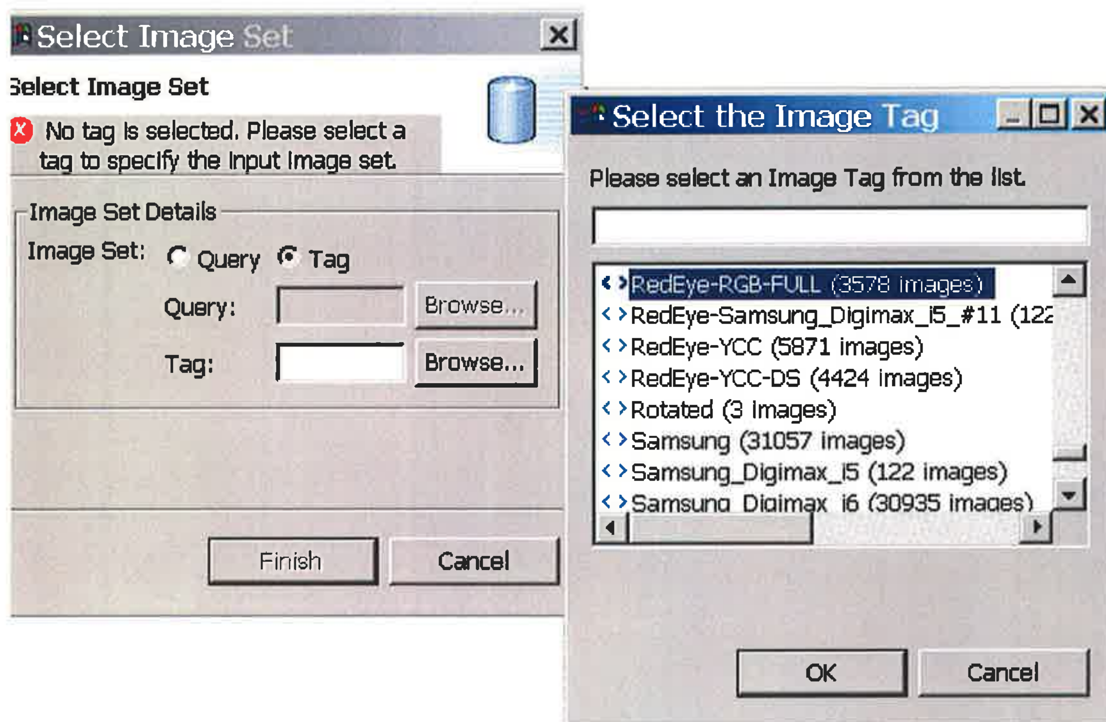
The above set of steps are simply reversed if the aim is to store new markings on the server, that is: extract markings from the client, convert to PFML ImageMarkup format, store in the image structure, and send the image back to the server.

Note: For simplification, Fig 5.2 shows only one image being downloaded to the client. In most cases, a set of images will be downloaded, and each image will be dealt with in turn by the user.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

5.2.4.1 Step 1 – The Image and its XML are downloaded to the client

The client downloads an image, or a set of images, from the server based on a query or a tag. Images are tagged based on their content, so for instance if twenty images are taken, all showing females with red-eye occurrences, all images would be given a tag such as "RedEye-Females001". The user chooses a tag as follows:

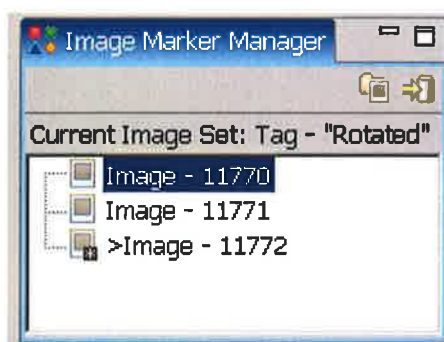


A query may also be used to retrieve images, for instance:

//feature[@class='face'][count(feature[@class='eye']) = 2]

The above query should return all images where two eye makings are present within a face instance.

After choosing the image tag, or querying the server, all of the images in the chosen set are downloaded to the client and cached in a temporary folder for fast access should the client need to re-mark a photo. XML files containing the image markup are retrieved from the server each time an image is loaded, and stored back onto the server each time an image is saved. Visually, the user interface component Image Marker Manager displays all downloaded images in the image list view as follows:



5.2.4.2 Step 2 – Get image markings from the image

After the user selects an image from the image list view, the loadImage(Image img) method is executed in a class on the client called Image Marker Manager. The loadImage method instantiates the ImageView, which will display the image, and also the FeatureTreeViewer class, which will display the feature hierarchy for the image.

A) Load an Image

After the FeatureTreeViewer class is created, it executes the following code:

IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
Image image = imageSetManager.getCurrentImage();
String xmlDir = ImageTestingPlugin.getAbsolutePath
                  ("config/featureDefinitions" );
ClientImage clientImage = new ClientImage( xmlDir, image );
```

Basically, the image set manager hands the FeatureTreeViewer a copy of the image structure. The xml directory for feature definitions (a more detailed schema, crafted for the client) on the client is also set, as this will be used later on when creating ClientImage structure. An instance of ClientImage is then created, and it is initialised with the xml directory for feature definitions, as well as the actual image.

B) Create the Data Model – ClientImage

Within the ClientImage constructor, the following lines are executed:

```
xmlPath = new File( xmlDir );
addMarkings( image.getMarkup() );
```

A feature definition basically defines how each feature must be used: what children a feature can have, how many instances of a particular child feature can be present, and what possible attributes can be set for that feature. Therefore, the client first needs access to all the client side feature definitions in order to build up the ClientImage/ClientFeature structure correctly, obeying schema rules, and with all the possible attribute setting for different features.

The method image.getMarkup() will return an instance of the PFML ImageMarkup. Thanks to the fact that the client has access to the PFML API, various methods can be executed on this ImageMarkup instance, allowing the client programmer to easily parse down through the XML structure, capturing components that are of use.

The addMarkings method basically carries out all that is necessary for Step 3, that is to say the ImageMarkup data are used to populate all client objects, or in this case the ClientImage class is saying “add all the markings inside image.getMarkup() to me”.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

5.2.4.3 Step 3 – Instantiate client objects and provide them with server metadata

Now it is time to take a look at what happens inside the addMarkings method in the ClientImage class:

A) Get Features from the markup

In order to get all root feature instances from the markup, the following method is executed:

```
Feature[] markupRootFeats = markup.getFeatures();
```

B) Go through each feature in the markup, creating a new root ClientFeature for each one

For each feature in the markupRootFeats, it is necessary to create another root ClientFeature on the client, and to instantiate all subfeatures of each root feature, ensuring the structure on the client matches exactly with that of the server. The method addEmptyFeatureTree(feature name) creates an empty root ClientFeature, as well as instantiating all sub features that should exist within that root ClientFeature. How does the addEmptyFeatureTree(name) know how many children to instantiate, and what attributes to instantiate, when it is only given a name from the markup? Answer: It will use the name it has been given to look up the feature definition for that feature, and it will create the structure based on that. For example, if the name is face, then the corresponding feature definition will state that two eyes are allowed as children, a nose is allowed, a mouth is allowed, a geometry ellipse is allowed for marking, an attribute personName can be set, etc.

```
for( int i = 0; i < markupRootFeats.length; i++ ) {
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
    // PART I - For each root feature add a new empty feature tree which
    // we will populate with the markings from a root feature.
    addEmptyFeatureTree( markupRootFeats[ i ].getName() );

    // PART II - Add the root feature to the new image markup
    ImageMarkup individualMarkup = new ImageMarkup();
    individualMarkup.addFeature( new Feature( markupRootFeats[ i ] ) );

    ClientFeature lastRootFeat = (ClientFeature) rootFeatures.get( i );
    lastRootFeat.markSubFeatures( individualMarkup );
}
```

As seen in Part I of the code, a for loop is used top go through each feature found in the markup features. The method addEmptyFeatureTree (name) creates the corresponding root ClientFeature for the markup feature.

C) For each feature in the markup, copy its marking into the corresponding ClientFeature

In Part II of the for loop, for each new root ClientFeature created, it is necessary to copy the marking data and other attributes from the corresponding markup feature.

```
for( int i = 0; i < markupRootFeats.length; i++ ) {
    // PART I - For each root feature add a new empty feature tree which
    // we will populate with the markings from a root feature.
    addEmptyFeatureTree( markupRootFeats[ i ].getName() );

    // PART II - Add the root feature to the new image markup
    ImageMarkup individualMarkup = new ImageMarkup();
    individualMarkup.addFeature( new Feature( markupRootFeats[ i ] ) );

    ClientFeature lastRootFeat = (ClientFeature) rootFeatures.get( i );
    lastRootFeat.markSubFeatures( individualMarkup );
}
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

As seen in Part II above, a object called individualMarkup is created. This object is created so that the markup for the current root feature (markupRootFeats[i]) can be isolated, and sent into the corresponding ClientFeature – if the entire markup structure was passed down into the ClientFeature structure, it would be difficult to determine which markup feature is being copied.

The lastRootFeat variable will contain the most recently instantiated root ClientFeature (as instantiated in the method addEmptyFeatureTree), and this is the ClientFeature that is currently of most interest, as it is the one for whom marking data is being obtained.

Thus, all that needs to be done is to execute the method markSubFeatures on the lastRootFeat object, passing it as argument individualMarkup – all subfeature of ClientFeature lastRootFeat will now be marked with all data in individualMarkup.

All of the above tasks are carried out for each feature found in the markupRootFeats, ensuring that all root feature taken from the server are instantiated as root ClientFeatures for this application.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

5.2.4.4 Inside the method ClientFeature.addEmptyFeatureTree(String rootFeatureName)

The method addEmptyFeatureTree was mentioned in the previous section, 5.3.4.3, but it is worth taking a look at *how* this method actually builds up the data model on the client.

```
public void addEmptyFeatureTree( String featureName ) {
      // Use DB to create full template
      myDB = getFeatureDefinitionDB();

      root = myDB.getDefinition( rootName );

      ClientFeature rootFeat =
            new ClientFeature( root.getName(), myDB, null );

      this.addRootClientFeature( rootFeat );
}
```

A) Get a reference to the FeatureDefinitionDB

Because the entire empty hierarchy consisting of ClientFeature objects must be instantiated, it is necessary to have access to the feature definitions for each feature type, thus ensuring that all objects are consistent with the database schema. The myDB variable is an instance of a FeatureDefinitionDB. The FeatureDefinitionDB singleton class is instantiated inside the method getFeatureDefinitionDB, using the specified XML path for feature definitions (mentioned in section 5.3.4.2, a), as follows:

```
myDB.init( xmlPath );
```

B) Instantiate the root ClientFeature

To add an empty feature hierarchy for the current root feature, all that needs to be done is to call the ClientFeature constructor, passing it simply the name of the root feature, a

IT Sligo An Institiúid Teicneolaíochta, Sligeach

reference to the feature definition database, and the parent of this feature. Because we are adding a root feature, parent is set to null.

Inside the ClientFeature (name, myDB, root) constructor, the following occurs:

I) The name argument is assigned to **this** ClientFeature instance

II) The parent argument is assigned to **this** ClientFeature instance

III) The XML feature definition for the current feature is retrieved by executing the following:

```
myDefinition = fdDB.getDefinition( name );
```

IV) Find out what geometry (marking) is allowed for this feature, and set it:

```
if( myDefinition.isGeometryAllowed( "Rectangle" ) ) {
geometry = Geometry.newGeometry( Geometry.RECTANGLE );
this.setCurrentGeometryType( geometry );
}
```

The above check will be carried out for each geometry type, until an allowed type is found. If no allowed geometry type is found, a "Geometry Type Not Supported" exception is thrown.

V) Not only does the feature definition database object define information on the features, it also specifies what attributes should be present within a particular feature. To get an attribute definition the following code is executed:

```
myAttributeDefinitions = myDefinition.getAttributeDefinitions();
```

VI) Now that it is known what attribute variables should be added for the current client feature, the following loop is executed:

```
for( int i = 0; i < myAttributeDefinitions.length; i++ ) {
String attrName = myAttributeDefinitions[ i ].getName();
      setAttribute( attrName, "" )
}
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

As seen above, each attribute is created on the client, and set with a null value. The setAttribute(name, value) method actually creates a new Attribute instance for this ClientFeature, and executes a fireUpdate(this), to notify all interested listeners that the attribute has been updated (more on this later).

VII) Lastly, all sub features for this client feature must also be instantiated, meaning that some recursion must occur. In brief, the following steps are carried out:

- Use the definition for this feature to find out what subfeatures are allowed: `myDefinition.getAllowedSubFeatures()`
- Find out *how many* instances of this subfeature should be created:

```
int maxCount = myDefinition.getSubFeatureMaxCount
(allowedSubFeatName);
```

- Create maxCount number of this subfeature. To create a feature, the following is executed:

```
        ClientFeature subFeat =
        new ClientFeature(allowedSubFeatName,this );
this.addSubFeature( subFeat );
this.fireNewRoot( subFeat );
```

As seen, the method addSubFeature is called, and this calls the ClientFeature constructor, recursing back into the above code when more subfeatures are found for each feature. Also note that fireNewRoot is called, notifying all listeners that a new root has been added to the structure (more on this later).

5.2.4.5 Inside the method ClientFeature.markSubFeatures(ImageMarkup m)

Now that an understanding is gained into the workings of the addEmptyFeatureTree method, it is time to find out how the newly created ClientFeature tree hierarchy is populated with markings from the markup structure. Taking another look at the previously mentioned for loop in the ClientImage.addMarkings method:

IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
for( int i = 0; i < markupRootFeats.length; i++ ) {
   // PART I - For each root feature add a new empty feature tree which
   // we will populate with the markings from a root feature.
   addEmptyFeatureTree( markupRootFeats[ i ].getName() );

   // PART II - Add the root feature to the new image markup
   ImageMarkup individualMarkup = new ImageMarkup();
   individualMarkup.addFeature( new Feature( markupRootFeats[ i ] ) );

   ClientFeature lastRootFeat = (ClientFeature) rootFeatures.get( i );
   lastRootFeat.markSubFeatures( individualMarkup );
}
```

The aim of Part II above is to plug the markup data for the current root markup feature into ClientFeature lastRootFeat. After the markSubFeature method is executed, lastRootFeat should contain all markings that were previously stored on the server for this particular feature in the image. markSubFeatures is another recursive method which will go through each and every feature in the tree.

The markSubFeatures method:

```
// Marks this feature and all the sub features that are marked in the
//specified markup.

protected void markSubFeatures( ImageMarkup markup ) {
      // PART I - Mark all subfeatures
      for( int i = 0; i < subFeatures.size(); i++ ) {
            ClientFeature currFeat = subFeatures.get(i);
            currFeat.markSubFeatures( markup );

            if( markup.hasFeature( currFeat.getName(), true ) &&
                  !currFeat.isMarked() ) {

                  Feature markedFeat = markup.getFeatureAndRemove
                  (currFeat.getName(), true );
```

```
                if( markedFeat.getGeometry() != null ) {
                // copy data from the marked feature
                // into the corresponding clientfeature.
                currFeat.setCurrentGeometryType(
                      markedFeat.getGeometry() );
                }
          currFeat.addData( markedFeat );
          }
}

// PART II - Now mark this feature + add attribute data
if( markup.hasFeature( this.getName(), false ) &&
!this.isMarked() ) {

      Feature markedFeat = markup.getFeatureAndRemove(
      this.getName(), false );

      if( markedFeat.getGeometry() != null ) {
      // copy data from the marked feature into this
            // clientfeature

this.setCurrentGeometryType(markedFeat.getGeometry());
      }

      this.addData( markedFeat );
  }
}
```

A) Part I – Mark all sub features

For each subfeature in the subfeature list in this ClientFeature instance, do the following:

- Mark its subfeatures recursively by calling `currFeat.markSubFeatures( markup )`
- The method `markup.getFeatureAndRemove(currFeat.getName(), true )` searches to see if there are any markings for the current feature in the markup based on the current feature name (e.g. face). It there are, this method removes the

corresponding feature from the markup using The getFeatureAndRemove method. This ensures that after the feature markup is stored in the client structure, it is removed from the markup structure – this helps to avoid markup duplication, as another instance of a feature may also have the same name, e.g. face.

- If the geometry for the markup feature is not empty it means the feature has been marked, so copy the data from the marked feature into the current ClientFeature by using `currFeat.setCurrentGeometryType(markedFeat.getGeometry()).`
- Lastly, the addData method is used to copy all attribute data from the markup into the current ClientFeature.

B) Part II -- Mark this *feature*

Now, *this* ClientFeature must be marked, that is, the ClientFeature instance we are working in at the moment. Again, a similar set of steps takes place:

- If *this* feature is found in the markup, then get it from the markup using `markup.getFeatureAndRemove(this.getName(), true ).` The getFeatureAndRemove method ensures that after this feature markup is stored in the client structure, it is removed from the markup structure.
- If the geometry for the markup feature is not empty, and the feature has been marked, then copy the data from the marked feature into this clientfeature instance by using `this.setCurrentGeometryType(markedFeat.getGeometry()).`
- Lastly, the addData method is used to copy all attribute data from the markup into this ClientFeature.

5.2.4.6 Database Architecture

While the focus in this section is on how the data model is created on the client, a brief overview of the PFML technology is useful in summarizing the PFML functionality that has already been explored while creating the data model. Also, it is worth taking a brief look at the technology used to retrieve images from the server.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

A) A look at PFML - Photographic Feature Markup Language

Fig 5.3 shows all PFML API classes and methods that are used on the client for various tasks. Because PFML is implemented in Java, the architecture is broken down into the various Java packages in the illustration.

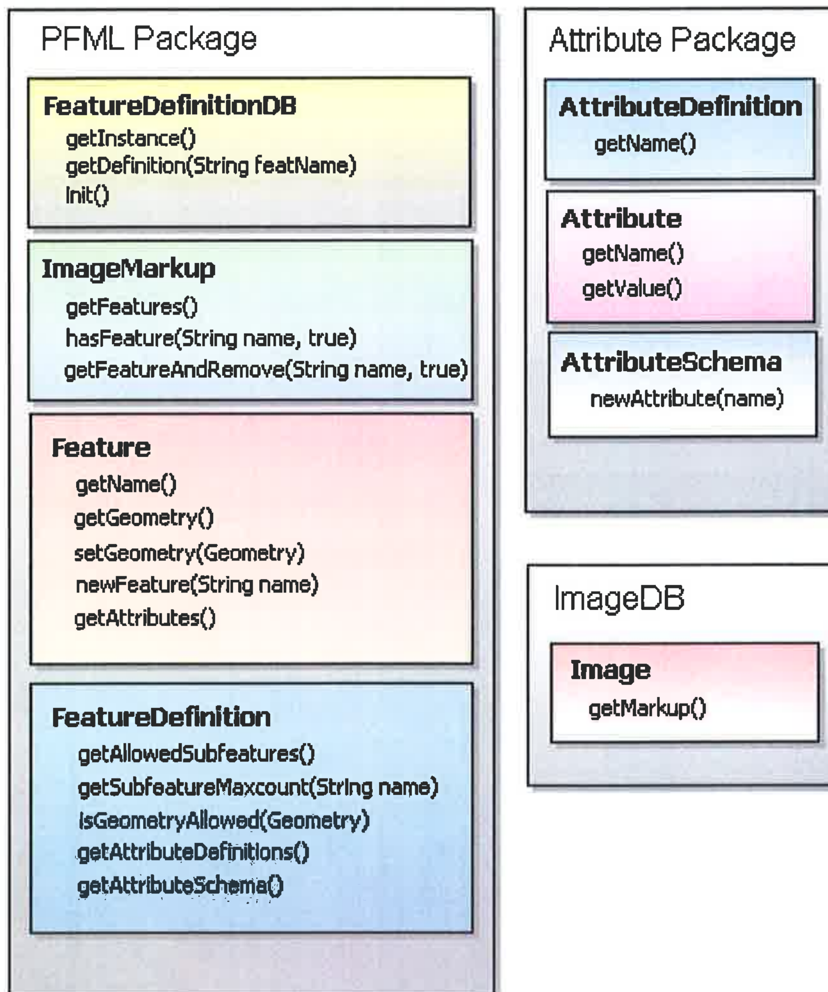


Figure 5.3: **PFML API Packages Used**

In terms of Java packages that make up the API, the PFML package, Attribute package and ImageDB package are the key PFML API packages used on the client, and the methods listed in Fig 5.3 are the ones that proved fundamental in building the data model for the client application. In addition to those, other packages also exist, but their focus is on background activities such as server interaction, marshaling and unmarshaling of the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

image files from the server, XML parsing, etc. Thus, it is out of the scope of this chapter to go into too much depth in this area. But what *is* worth exploring is how the client application interacts with the server image files.

As previously mentioned, the PFML API resides on the client application and allows for interfacing with the XML database images after they have been downloaded. Using the API, the client sends requests to ImageDB image instance (see Fig 5.3) for data, and receives back image metadata in the form of image markup. The ImageMarkup object, part of the PFML package in the API, converts raw XML data into a format that is immediately useful on the client. As seen in earlier sections of this chapter, the PFML will allow the developer to efficiently parse through the image metadata XML files, gathering relevant metadata and storing that data in objects for the client. Because PFML handles the parsing and marshalling of XML files, the client developer is free to focus on the more pressing task of building objects of the client, and getting those objects to work with the user interface.

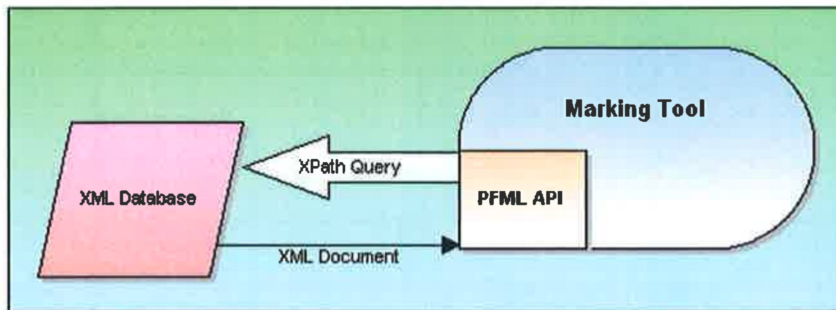
B) XPath

It is known that an XML database is used on the server, storing images as well as their associated XML documents. When XML documents and images are retrieved from the server, PFML helps the client to easily extract useful information from the XML. But how are images actually retrieved from the database in the first place? The XML is just sitting on the server, and needs to be queried somehow.

XPath, a technology for searching and querying raw XML data, provides the solution to this problem. XPath (XML Path) is a W3C (World Wide Web Consortium) recommended syntax, and allows for the querying of indexed XML data, as well as the earch and retrieval of information within an XML document structure. According to W3C (1999), instead of being an XML syntax, XPath instead uses a syntax that relates to the logical structure of an XML document.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

In the client application, an XPath query is sent to the XML database for an image or a group of images, and the XML image document(s) are returned. The retrieved XML image document also specifies an associated image file, usually in jpeg format, and this will also be downloaded to the client.



After being downloaded, the PFML API is responsible for storing the image XML, parsing the XML and for providing methods that let the client developer manipulate the image XML.

5.2.5 Overview of the Database and Data Model

In this section, the database side of the application was explored. It was seen how the client can instantiate the data model, consisting of ClientImage and ClientFeature objects, based on data held in the XML database. The use of the PFML API was exhibited through the exploration of some programming code that is executed upon downloading an image. While examples given only show a small side of the data model architecture used for this application, they should provide a clue as to the importance of the client object design, the server XML schema design and the means to the efficient exchange of data between the two using the PFML technology developed in conjunction with the partner company. Additionally, the significance of this data model on the client will be further reinforced when graphical user interface components are explored later on. Some of the immediate benefits include:

I) Low coupling and high cohesion

Code developed for the data model – mainly ClientImage and ClientFeature - will work in isolation of the user interface, and does not truly depend on any of the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

client application classes. While the model may include some methods that are useful for client side functionality, such as the firing of an event so that graphical components will be updated when a new root is added, application specific data is kept away from the model.

II) Portability

If there was a requirement to develop another application using the data model mentioned in this chapter, porting it over should not prove very difficult. Perhaps small changes may need to be made, such as extra functionality or additional storage types if required, but by and large the model would remain the same.

III) Modularity

When implementing the model, it was fundamental that

Modular systems are known to be easier to maintain and update in the long run. Not only this, but it was found that thinking in terms of modularity at each stage of the design and implementation of this solution reduced complexity and provided a fast and concise approach to the problem.

IV) Standardisation

Because the model works with W3C standards to retrieve images from the database

IT Sligo An Institiúid Teicneolaíochta, Sligeach

5.3 Implementation of the Client Interface

Initially, when deciding what programming environment to use for this project, the ability to quickly develop advanced user interface components was imperative. While the Microsoft Foundation Classes (MFC) architecture provided many reusable classes and interface controls, it was found that from a developer's point of view their complexity and reliance on Microsoft developed technology was a drawback, and limited code portability. Eclipse exists at the other end of the spectrum entirely; it is cross-platform

and open ended, allowing just about anybody with some programming experience to add another Eclipse Plug-In to further extend the IDE.

This section will provide details firstly on how Eclipse helped with the development of user interface components, and secondly on how the data model (as explored in section 5.3 Database Interaction and Interfacing with the Client) was used within the interface.

5.3.1 Interface Construction and Controls

A. GUI Components on the Eclipse Platform

The official Eclipse website declares: "The Eclipse Project is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools". It could be said that Eclipse is more than just a Java IDE, it is more like an open platform allowing for tool integration, as well as a fully functional Java IDE in the form of the Eclipse plug-in components. When an Eclipse based application initializes, it discovers and activates all of the plug-ins that have been configured for the workstation. The components allow for the extension of Eclipse functionality. The Eclipse platform is able to performing any function that has been added to it by the plug-ins it currently contains.

Such is the nature of Eclipse, development Marking Tool relies on the use of various plug-in components that will enhance the user interface of this application. Plug-ins are developed using the plug-in development environment (PDE), as well as a set of Java development tools (JDT).

The following elements of Eclipse became of fundamental importance when developing the Marking Tool:

- Extensibility Model – because the Marking Tool application is built using Eclipse's plug-in architecture, it will itself be a plug-in that sits on top of the Eclipse desktop.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

- SWT – the standard widget toolkit will be used for developing application graphics. SWT, developed as a part of Eclipse, allows Java applications to access native operating system functionality and resources. This means if a component such as a message box is written in Java using SWT, then when the code is run on Windows the message box has a Windows look and feel (widget colour, size, shading, etc.). If the message box appears on the Apple Mac OSX operating system, then it has an OSX look and feel. Using SWT for the Marking tool, it was possible to incorporate a wide range of events, layout managers and widgets. SWT even uses native operating system specific components such as drag and drop. Older Java widget toolkits such as AWT and Swing may also be used in Eclipse, however SWT was written as part of Eclipse because the entire Eclipse platform, along with all its plug-ins, have been written in SWT.
- JFace - for building the Marking Tool graphical user interface, SWT is often used along with JFace – Eclipse's user interface framework. JFace provides programmable components to the developer such as dialogs, tree views, filters, action listeners, toolbars, toolbar managers. JFace certainly helps to speed up interface development, and as well as this
- Perspectives, Editors and Views – Eclipse provides a framework for the creation of:

  - Views – as shown in the Model-View-Controller architecture, a single data model may have many views. Similarly, a program – or a single perspective – may provide to the user different views of the same data. Eclipse provides Views to easily implement this functionality
  - Editors – an editor allows the user to edit object information, so most editors will allow the user to create, edit, save and delete data.
  - Perspectives – a single application may provide different functions, each function can be divided up into a perspective that is, in itself, a fully featured program. This allows the developer to "divide and conquer" the application development, and also provides to the user various clear and concise interfaces

IT Sligo An Institiúid Teicneolaíochta, Sligeach

- User control – the user can stack, tile, move and arrange views and editors at will within a perspective. So a perspective is actually made of views and editors. Only one perspective is visible within a window. If the user wishes to move to another perspective, a part of the application dealing with different functionality, then he or she must save data within the current perspective and choose a different perspective.

With Eclipse, an application is built starting from the Eclipse workbench. From here, it is necessary to choose what plug-ins best suit the needs of the developer, and then add them to the application, and adapt them to suit the requirements of the implementation. Plug-ins that are not required can be removed from the workbench.

B. Using Eclipse to Implement the Marking Tool Interface

In this section, the use of Eclipse in the development of the Marking Tool user interface is explored. It should be noted that all of the classes explored in this section are subclasses of the Eclipse ViewPart class. This means they inherit all the functionality of ViewPart, such as the ability to define a new view that is displayable within an Eclipse perspective. All of the classes explored in this section are views within the Image Marking Tool perspective.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

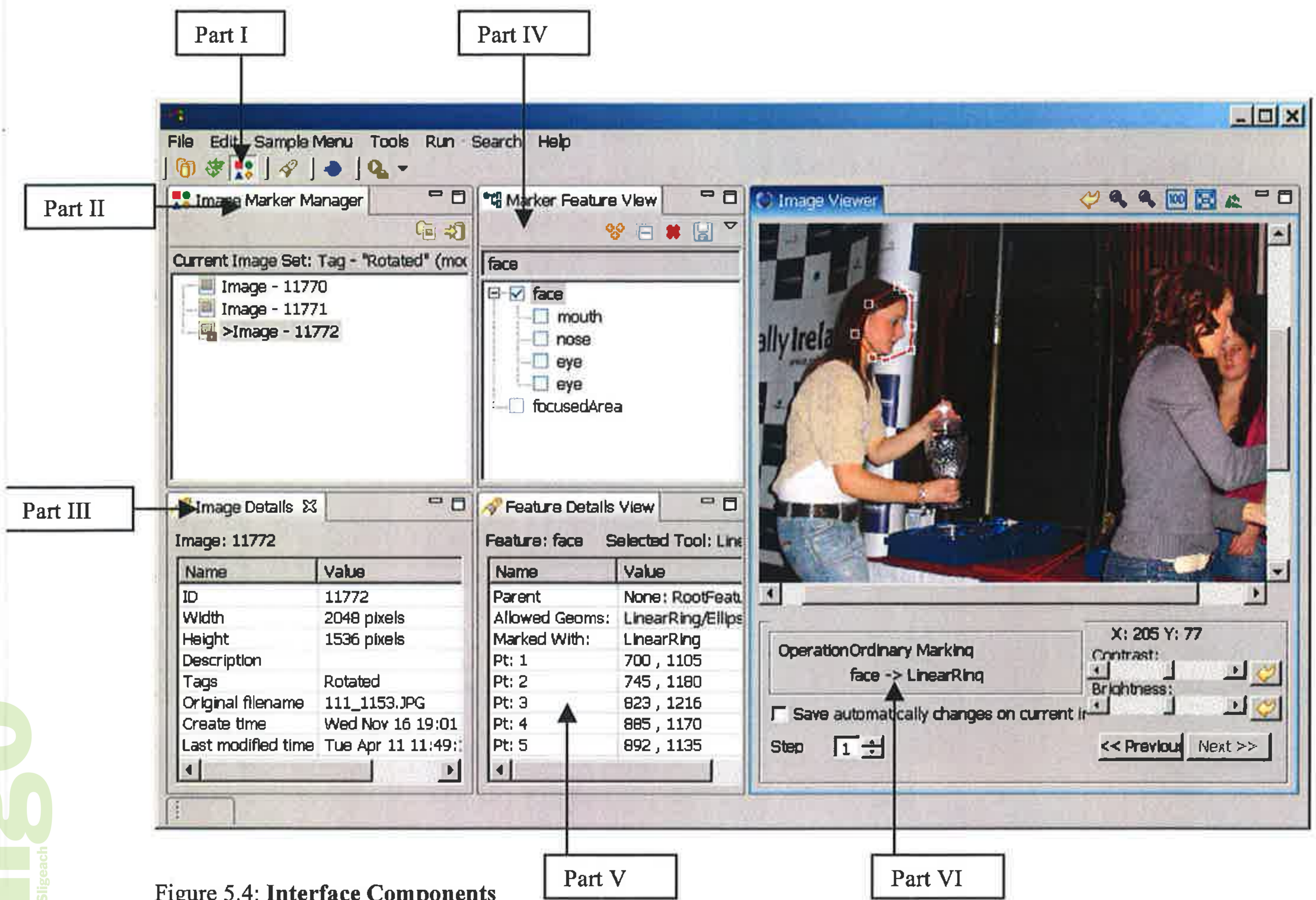


Figure 5.4: **Interface Components**

Part I – The Image Marking Tool Perspective

In the top left corner of the screen, a small panel shows the different perspectives that make up the Testing and Reporting framework. The third one from the right – the Marking Tool - is highlighted as currently selected. Thus, the perspective (application component) currently being used is the Marking Tool.

The following perspectives are available in the Testing and Reporting Framework:

- Image Database Tool
- The Testing Tool
- The Marking Tool

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

As discussed in Chapter 2 – Methodology, three different components are used for the overall framework, and thanks to the Eclipse platform's architecture, these elements can be implemented as perspectives. As previously mentioned, a perspective contains a collection of views and editors.

Part II – The Image Marker Manager

The ImageMarkerManager class is a child of the Eclipse class ViewPartn this case, the view defined will display the list of images downloaded from the server.

The ImageMarkerManager view does the following:

- Instantiates an instance of FeatureTreeViewer (the tree view class)
- Gives the tree view a reference to the ImageView class
- Loads an image into the tree view
- Tells the ImageView class to load the current image
- Allows the user to enter a query to retrieve images from the image database
- Displays the list of images that have been returned from the server
- Downloads the actual image from the server and hands it to the ImageView class for display
- Saves new image markings to the server after the user finishes marking

Part III – The Image Details View

The ImageDetailsView class accesses image metadata from the currently loaded image and displays it in a simple table, using a grid layout. Examples of properties displayed are: width, height, description, date.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Part IV – The Marker Feature View (FeatureTreeViewer)

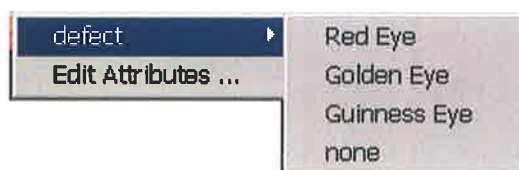
The FeatureTreeViewer class contains a TreeViewer instance which is used to display all features within an image. The features appear in a hierarchical structure. After the ImageMarkerManager tells the FeatureTreeView that a new image has been loaded, the FeatureTreeView populates it's nodes with all the marked (ticked) features from the image.

Part V – The Feature Details View

After a feature has been selected in the tree viewer, it's properties are displayed in the FeatureDetailsView component. This class uses a grid layout with two columns to display various properties and their corresponding values. This view should display items such as: the geometry used to mark the feature, the defect if one is present (e.g. red-eye), co-ordinates for the marking.

Part VI – The Image Viewer

The ImageViewer class displays an image to the user. If a feature is currently selected in the tree view, then the user can use the mouse to draw a marking on the image in the location of the specified feature. For instance, if the user wants to mark a red-eye he/she will select eye in the tree view, and then draw an eye marking across the instance of red-eye he/she visually finds in the actual image. After the user has finished drawing the marking using the left mouse button, he/she will release the left mouse button, and when this happens a menu pops up to allow the user to quickly set an attribute as follows:



In the first pop-up menu, the user chooses to either set a defect (if defects can be set for the feature in question) or else go into the edit attributes dialog. If the user wants to set a defect, then an additional menu is displayed allowing the user to select the defect – in this

IT Sligo An Institiúid Teicneolaíochta, Sligeach

case red-eye, golden eye and Guinness eye are displayed. On the other hand, if the user wishes elects to choose "Edit Attributes", the following dialog is displayed:

The ImageView class is the largest and most complex class for the Testing Tool as it contains all of the logic that dictates how markings are displayed under various circumstances (rotation, zooming, scrolling), as well as ensuring that events are fired to tell other views when a marking is moved or resized. It also makes it possible to undo and redo all marking activities.

5.3.2 Integrating the Data Model with the View

Now that the implementation for the interface design has been explained, it is worth looking at how the model-view-controller architectural design will help in tying the graphical interface components together with the underlying data model. Fig 5.5 shows a broad outline of how the various pieces hang together.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

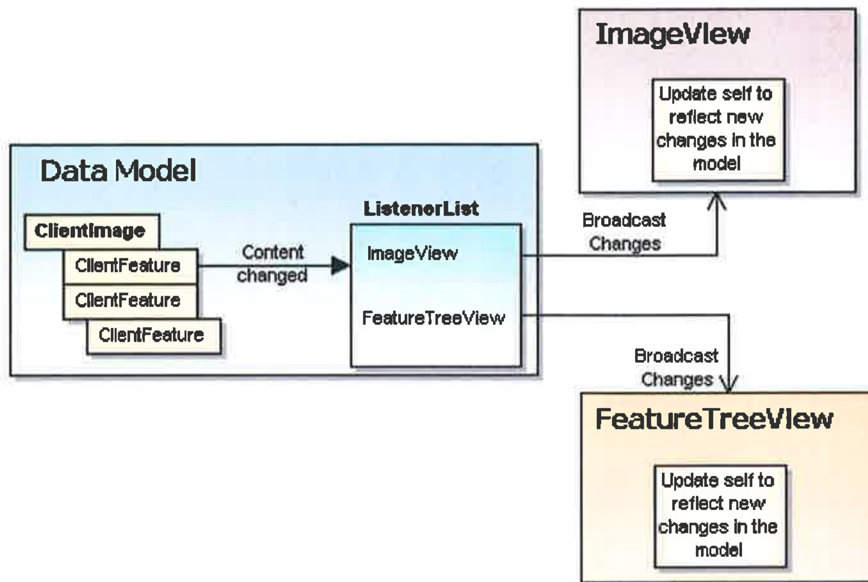


Figure 5.5: **Data Model and View Integration**

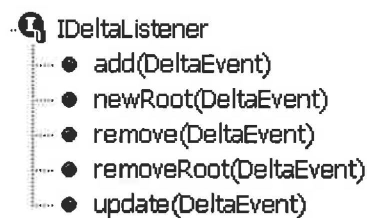
While quite an over simplification, the above diagram will serve as a good starting point for the exploration of the IDeltaListener implementation.

5.3.2.1 Broadcasting changes in the model data to all interested parties

Fig 5.5 shows the data model, consisting of a ClientImage and its child ClientFeatures. When the content or structure of the model changes in any way, other parts of the application must be made aware of this change. In the case of the Marking Tool, it is vital that ImageView and FeatureTreeView are notified of changes in the model so that the user can be presented with an interface consisting of up to date visual representations of the underlying objects. Thus, FeatureTreeView and ImageView will explicity ask to be added to the data model's ListenerList structure. When a change occurs in the data model, all components found in the ListenerList will be sent details of the data change. It is then up to the individual components to decide for themselves how best to deal with the changes. For instance, the FeatureTreeView may choose to tick the "marked" tickbox opposite a feature if this feature has just been marked. The ImageView may write a

different implementation to deal with the same event, for instance to highlight this new marking on the screen.

To implement this functionality, it was necessary to create an Interface named IDeltaListener, consisting of the four main events that an instance of IDeltaListener can receive, or rather methods that can be defined by all implementations of this class. In Fig 5.5, the ListenerList in the data model component actually consists of IDeltaListener instances. Each individual ClientFeature in the model will contain its own ListenerList, and within each ListenerList are a group of objects that have asked to be notified of events for that particular object. In our case, most ClientFeatures will have a ListenerList containing ImageView and FeatureTreeView. If something occurs to change the contents of the ClientFeature, then it is that ClientFeatures responsibility to iterate through each element within its ListenerList, calling the relevant IDeltaListener method in each case to notify the interested object what changed have occurred. When writing the IDeltaListener class, the main question was; what change events in the model should a listening object be interested in? The solution was defined as follows:



Thus, each implementation of IDeltaListener will be able to receive notification when a marking is added to a ClientFeature in the structure, when a new root ClientFeature is added to the structure, when a marking is removed from a ClientFeature in the structure, when a root is removed from the structure and when the data for a ClientFeature is updated or changed. Next, the implementations for model and view events will be explored.

On the Model Side

The implementation of the model has already been explored, now focus will be placed on understanding how listener events have been implemented. When an event occurs, such as a new root, it needs to be "fired" so that it can be broadcasted to interested objects.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

ClientImage

A ClientImage is mostly used as a container for a ClientFeature structure, usually consisting of a set of "root" features. Therefore the only changes in a ClientImage that will be of importance will be the creation of a new root ClientFeature, the removal of a root ClientFeature. There will also need to be a method to add listeners to the ListenerList for a ClientImage. The following methods have been implemented:

- ➔ `protected void fireNewRoot(Object newRoot)`

  When a new root is added to the ClientImage in the method addRootClientFeature(ClientFeature newFeat), fireNewRoot(newRoot) is called, and the argument newRoot consists of the new ClientFeature. Within the method, the following loop will tell all listeners in the ListenerList that a new root has been added:

  ```
  for( Iterator iter = listenerList.iterator(); iter.hasNext(); ) {
        IDeltaListener listener = (IDeltaListener)
        iter.next();
        listener.newRoot(                    event                    );
        }
  ```

  It is up to each individual listener to decide how to implement its newRoot method.

- ➔ `protected void fireRemoveRoot(Object root)`

  When a root is removed from the ClientImage in the method removeRootFeatureAt(ClientFeature rootToRemove), fireRemoveRoot(root) is called, and the argument root consists of the ClientFeature that must be removed from the structure. Within the method, the following loop will tell all listeners in the ListenerList that a root has been removed:

  ```
  for( Iterator iter = listenerList.iterator(); iter.hasNext(); ) {
        IDeltaListener listener = (IDeltaListener)
        iter.next();
        listener.removeRoot(                    event                    );
        }
  ```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

It is up to each individual listener to decide how best to implement its removeRoot method.

➔ `protected void fireNewRoot(Object newRoot)`

When a new root is added to the ClientImage in the method addRootClientFeature(ClientFeature newFeat), fireNewRoot(newRoot) is called, and the argument newRoot consists of the new ClientFeature. Within the method, the following loop will tell all listeners in the ListenerList that a new root has been added:

```
for( Iterator iter = listenerList.iterator(); iter.hasNext(); ) {
        IDeltaListener listener = (IDeltaListener)
        iter.next();
        listener.newRoot(                event              );
        }
```

It is up to each individual listener to decide how to implement its newRoot method.

➔ `public void addListener(IDeltaListener listener)`

In order for any object to be added to the ListenerList for a ClientImage, it must call the method addListener on the ClientImage. The ClientImage will then do the following:

```
        if(   !this.listenerList.contains(   listener   )   )   {
        this.listenerList.add(              listener              );
}
```

If the listener supplied by the calling object (usually it supplies itself, using **this**) is not already in the ListenerList for the ClientImage, it will be added.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

ClientFeature

ClientFeatures are the key building blocks for the model. As such, if any changes occur whasoever, all listeners will need to be notified. There will also need to be a method to add listeners to the ListenerList for a ClientFeature. The following methods have been implemented:

- ➔ `protected void fireAdd(Object added)`

  When new markings are added to a ClientFeature in the method mark(Geometry geom), fireAdd(added) is called, and the argument added consists of the current ClientFeature with its updated data. Within the fireAdd method, the following loop will tell all listeners in the ListenerList that an add event has occurred, and the new ClientFeature will be supplied as the event argument:

  ```
  for( Iterator iter = listenerList.iterator(); iter.hasNext(); ) {
        IDeltaListener listener = (IDeltaListener)
        iter.next();
        listener.add(                    event                    );
        }
  ```

  It is up to each individual listener to decide how to implement its add method.

- ➔ `protected void fireRemove(Object removed)`

  When markings are deleted from a ClientFeature in the method unmark(), fireRemove (removed) is called, and the argument removed consists of the ClientFeature that must be removed from the structure. Again, as in the fireAdd method described previously, all listeners in the ListenerList will be informed of the remove event by executing `listener.removeRoot( event )` in each case.

- ➔ `protected void fireUpdate(Object updated)`

  When markings in a ClientFeature are changed in any way, fireUpdate (updated) is called, and the argument updated consists of the ClientFeature whose contents have changed. Again, the ClientFeature in question will inform all listeners in the ListenerList that there has been a change in content by executing

ITSligo An Institiúid Teicneolaíochta, Sligeach

`listener.update( event )` in each case. When the listener receives the update call, it must do some processing to update itself with the new object contents.

➔ `protected void fireNewRoot(Object newRoot)`
This method is called from the ClientImage fireNewRoot method, and most implementations of it will call addListener for the current ClientFeature, as well as for all subfeatures found within it. This ensures that listeners added to each node in the entire structure, and will thus receive notification should the model content change. Traversing through all subfeatures and executing addListener for each one could be done on the listener side using simple code such as the following:

```
feature.addListener( this );
ClientFeature[] subFeatures = feature.getSubFeatures();
for( int i = 0; i < subFeatures.length; i++ ) {
    addListenerTo( subFeatures[ i ] );
}
```

➔ `public void addListener(IDeltaListener listener)`
In order for any object to be added to the ListenerList for a ClientFeature, it must call the method addListener on the ClientFeature. This method adds the object to the current ClientFeatures ListenerList. The ClientFeature will do this as follows:

```
    if( !this.listenerList.contains( listener ) ) {
    this.listenerList.add( listener );
}
```

If the listener supplied by the calling object (usually it supplies itself, using **this**) is not already in the ListenerList for the ClientFeature, it will be added.

As seen, the implementation for event handling on the model side covers all potential structural and content changes. Listeners can be registered, and will be notified in turn when a change occurs, or is “fired”.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Now that the workings of the model have been explored regarding its handling of events, it is worthwhile to look at a few examples of how the various viewers will respond to events. As mentioned in the previous section, each viewer will react differently when it is notified of an event change in the model.

<u>On the TreeView side</u>

When an image is loaded, class FeatureTreeView instantiates the ClientImage structure (thus instantiating all root features and their child features). FeatureTreeView also has an instance of TreeViewer called viewer. The TreeViewer class is a concrete viewer that will work alongside the SWT GUI controls, handling update events and changes to the structure.

FeatureTreeView

In the loadImage method of class FeatureTreeView, the ClientImage structure is passed back to the TreeViewer instance:

```
viewer().setInput(                    this.currentImage                    );
viewer().expandAll();
```

The above code basically plugs the ClientImage structure into the TreeViewer, and expands all nodes so the user can see the entire structure with all ClientFeatures and child ClientFeatures.

Content for the TreeViewer is handled by class FeatureTreeContentProvider, this link is established as follows:

```
viewer.setContentProvider(new FeatureTreeContentProvider());
```

The content provider takes responsibility for mediating between the viewer's model (ClientImage) and the viewer itself, keeping track of content changes within the TreeViewer and ensuring that when the user does something such as expand nodes or collape nodes that the correct data is on display. Now that the content provider has been

IT Sligo An Institiúid Teicneolaíochta, Sligeach

set for our TreeViewer, it should watch out for changes in the model (if the necessary listener methods have been implemented).

FeatureTreeContentProvider

Because class FeatureTreeContentProvider implements the IDeltaListener interface, it should provide implementations for all methods in class IDeltaListener – namely add, newRoot, removeRoot, remove and update. The code written for these methods should do something useful when changes occur in the model. While it is not necessary to explore the implementation of all the listener methods, it is worth looking at a couple in order to get the general idea of how this part of the interface interacts with the model.

First of all, it has been seen that when an image is loaded, the ClientImage structure is instantiated within class FeatureTreeView pertaining to the XML rules on the server. FeatureTreeContentProvider has been linked to TreeViewer, which is in turn linked to FeatureTreeView, so any changes in content must go through the FeatureTreeContentProvider.

Before being able to receive events from the model however, the FeatureTreeContentProvider, which has access to the full ClientImage structure, must add itself as a listener to each and every feature and all subfeatures within the ClientImage structure. This is using the following methods, the second of which goes into recursion:

```
protected void addListenerTo( ClientImage image ) {
    image.addListener( this );

    for (Iterator iterator = image.getRootFeatures().iterator();
            iterator.hasNext();) {
        ClientFeature aimage = (ClientFeature) iterator.next();
        addListenerTo( aimage );
    }
}

protected void addListenerTo( ClientFeature feat ) {
    feat.addListener(this);
    ClientFeature[] subFeatures = feat.getSubFeatures();
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
        for (int i = 0; i < subFeatures.length; i++ ) {
            addListenerTo( subFeatures[ i ] );
        }
    }
```

The lines of code `image.addListener( this )` and `feat.addListener(this)` do the actual work of adding `this` – the current instance of FeatureTreeContentProvider – to the ListenerList of the relevant ClientImage/Feature. As already seen, the addListener(obj) method of ClientImage or ClientFeature simply does the following:

```
this.listenerList.add( listener )
```

In this case, the listener will be an instance of FeatureTreeContentProvider.

Now that the FeatureTreeContentProvider is set up to listen out for events that are fired in any of the model's nodes, it is now time to write code to do some processing if an event occurs. To show how this was implemented, we will look at two examples:

a) If the model fires an add event after data is added to a ClientFeature, then this method will be executed. It calls the TreeViewer's update method, supplying it with the receiver of the event - event being the model change, and getTarget() returning the receiver of this event, the changed ClientFeature. So the viewer will be refreshed with the new ClientFeature data.

```
    public void add( DeltaEvent event ) {
        viewer.update( event.getTarget(), null );
    }
```

b) If the model fires a removeRoot event after a root feature is removed from the ClientImage structure, then the removeRoot method will be executed in FeatureTreeContentProvider. The method calls the removeListenerFrom method, supplying it with the receiver of the event - the ClientFeature that was removed. Looking at the removeListenerFrom method, it is seen that the removeListener method is not only executed for the ClientFeature who fired the event, it is also executed for each child of

IT Sligo An Institiúid Teicneolaíochta, Sligeach

that ClientFeature. This is because once a root is removed, FeatureTreeContentProvider must not only remove itself from the listener list of the root feature that was removed, but also from all children, as they have also been removed. There is no need to worry about updating the user interface for the TreeViewer, as this will be automatically handled when the content provider is informed of a change in the structure.

```
public void removeRoot( DeltaEvent event ) {
    removeListenerFrom( (ClientFeature)event.getTarget() );
}
protected void removeListenerFrom( ClientFeature feat ) {
    feat.removeListener( this );

    ClientFeature[] subFeatures = feat.getSubFeatures();

    for (int i = 0; i < subFeatures.length; i++ ) {
        removeListenerFrom( subFeatures[ i ] );
    }
}
```

On the ImageView side

Again, as class ImageView implements the IDeltaListener interface, it must also provide implementations for the methods in class IDeltaListener – namely add, newRoot, removeRoot, remove and update. The code written for these methods should do something useful in the ImageView class when changes occur in the model. In general, the task here will be to ensure that all markings drawn on the ImageView correspond with the data stored in the model. If a new marking is added to the model, it should appear on the ImageView. If a marking is removed, it should not be visible in the ImageView. If a marking is moved or resized, the ImageView should show the marking in its new position, or at its new size.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

ImageView

a) First of all, after an instance of ImageView has been instantiated, a ClientImage structure will be loaded into it. Before being able to receive events from the model however, ImageView must add itself as a listener to each and every feature and all subfeatures within the ClientImage structure. This is done using the following methods, the second of which goes into recursion:

```
protected void addListenerTo( ClientImage image ) {
    image.addListener( this );

    for( Iterator iterator = image.getRootFeatures().iterator();
            iterator.hasNext(); ) {
        ClientFeature aimage = (ClientFeature) iterator.next();
        addListenerTo( aimage );
    }
}

protected void addListenerTo( ClientFeature feat ) {
    feat.addListener(this);
    ClientFeature[] subFeatures = feat.getSubFeatures();
    for (int i = 0; i < subFeatures.length; i++ ) {
        addListenerTo( subFeatures[ i ] );
    }
}
```

b) Now that the ImageView class is set up to listen for events in the model, implementations must be written for add, newRoot, remove, removeRoot and update.

- ➔ If a new root is created in the model, then this method is fired in the ImageView class. The method recursively adds the ImageView class to the ListenerList of the new root feature, and to all of its subfeatures. The addListenerTo method works the same as that listed above in a) of this section.

```
public void newRoot( DeltaEvent event ) {
  addListenerTo( (ClientFeature) event.getTarget() );
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

}

➔ When a marking is added to a feature in the model, the add method is fired on all listeners in the ClientFeature's ListenerList. The ImageView class will handle an add event as follows: Set the variable newMarkingsAdded to true. Get the receiver of the add event by using the getTarget() method, then cast this into a ClientFeature instance. Get the actual marking from the ClientFeature instance and store it in list structures that will be used for display purposes later on in the ImageView class. Then, after checking that the quickMark facility is not being used, execute treeView.selectFeat(clientFeature) to ensure that this recently updated feature is currently selected in the TreeViewer control. Now all that needs to be done is to repaint the image, and redisplay all geometries, to ensure that the ImageView class presents to the user all recent changes.

Regarding the use of this method – if the user draws a marking on the ImageView display, when mouseUp occurs, the method ClientFeature.mark(geometry) is called for the ClientFeature being marked in the ImageView, and this method actually fires an add event, which is then sent back to the ImageView. This call back protocol works quite well, and ensure that not only the ImageView, but all the other Views will know about the changes that were made in the model.

```
public void add( DeltaEvent event ) {
  if( !newMarkingsAdded ) {
      newMarkingsAdded = true;
  }
  ClientFeature clientFeature = (ClientFeature)event.getTarget();
  displayedMarkings.add( clientFeature.getMarking() );
  geomFeatMap.put( clientFeature.getMarking(), clientFeature );
  if( (!(quickMarkUtil.getQuickMarkAllOn()))
          || (!quickMarkUtil.getQuickMarkUnMarkedOn()) ) {
      treeView.selectFeat( clientFeature );
      paintImage();
      displayGeometries();
```


IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
    }
  }
```

- When a marking is removed from a feature in the model, the remove method is fired on all listeners in the ClientFeature's ListenerList. The ImageView class will handle a remove event as follows: Get the currently selected feature in the TreeViewer. Now, in the for loop, iterate through all onscreen displayed markings until the one found is identical to the one that has been removed in the model. When found, remove this marking from the displayed markings, remove it from geomFeatMap (a structure used in ImageView for keeping track of mappings between features and geometries), the repaint the image, and redisplay all geometries. Now all that needs to be done is to repaint the image, and redisplay all geometries, to ensure that the ImageView class presents to the user all recent changes. Lastly, we select in the TreeViewer the ClientFeature whose marking was most recently removed.

Use of this method is quite similar to add above, and when the user selects a marking in the ImageView and hits the delete button, ClientFeature.unMark() will be called. This in turn will fire a remove event in the model, triggering the ImageView's remove implementation as seen below.

```
public void remove( DeltaEvent event ) {
  Geometry marking =
((ClientFeature event.getTarget()).getPreviousGeometry();
  ClientFeature selectedFeature = getSelectedFeature();
  for( int i = 0; i < displayedMarkings.size(); i++ ) {
       // Find Marking to remove in displayedMarkings List and
              // GeomFeatMap
       if( displayedMarkings.get( i ).equals( marking ) ) {
           displayedMarkings.remove( i );
           geomFeatMap.remove( marking );
           paintImage();
           displayGeometries();
```

```
        }
    }
    treeView.selectFeat( selectedFeature );
}
```

➔ When a root is removed, the removeRoot method calls the recursive method removeListenerFrom, which ensures that the ImageView is not registered as a listener in the removed root feature, or any of its subfeatures, which will also have been removed from the model structure.

```
public void removeRoot( DeltaEvent event ) {
    removeListenerFrom( (ClientFeature) event.getTarget() );
    paintImage();
    displayGeometries();
}
```

➔ When any of the details for a ClientFeature are changed, it fired an update event. In the ImageView this event is handles by calling the updateFeatureDetailsView() method on the FeatureTreeView instance, ensures that the most recent data is onscreen in the FeatureDetailsView display.

```
public void update( DeltaEvent event ) {
    getTreeView().updateFeatureDetailsView();
}
```

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

5.3.2.2 User interaction with the FeatureTreeView and ImageView

The previous section explored how changes in the model are handled in the views. However, what was left out was the issue of how the model structure actually gets created and altered in the first place. The answer to this question is; through the graphical user interface with the use of various controllers. The user interacts with controls in the user interface and changes the contents of the model as he/she uses the application's user interface, or view. Fig 5.6 attempts to give an overview of the most important user initiated exchanges that will occur between the two views – ImageView and FeatureTreeViewer. It also shows how the model will be affected by user interaction with these views. So whereas the previous section was centred around the model, in this section our exploration is centred on the user interaction with views, or rather the View-Controller part of the Model-View-Controller architecture.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

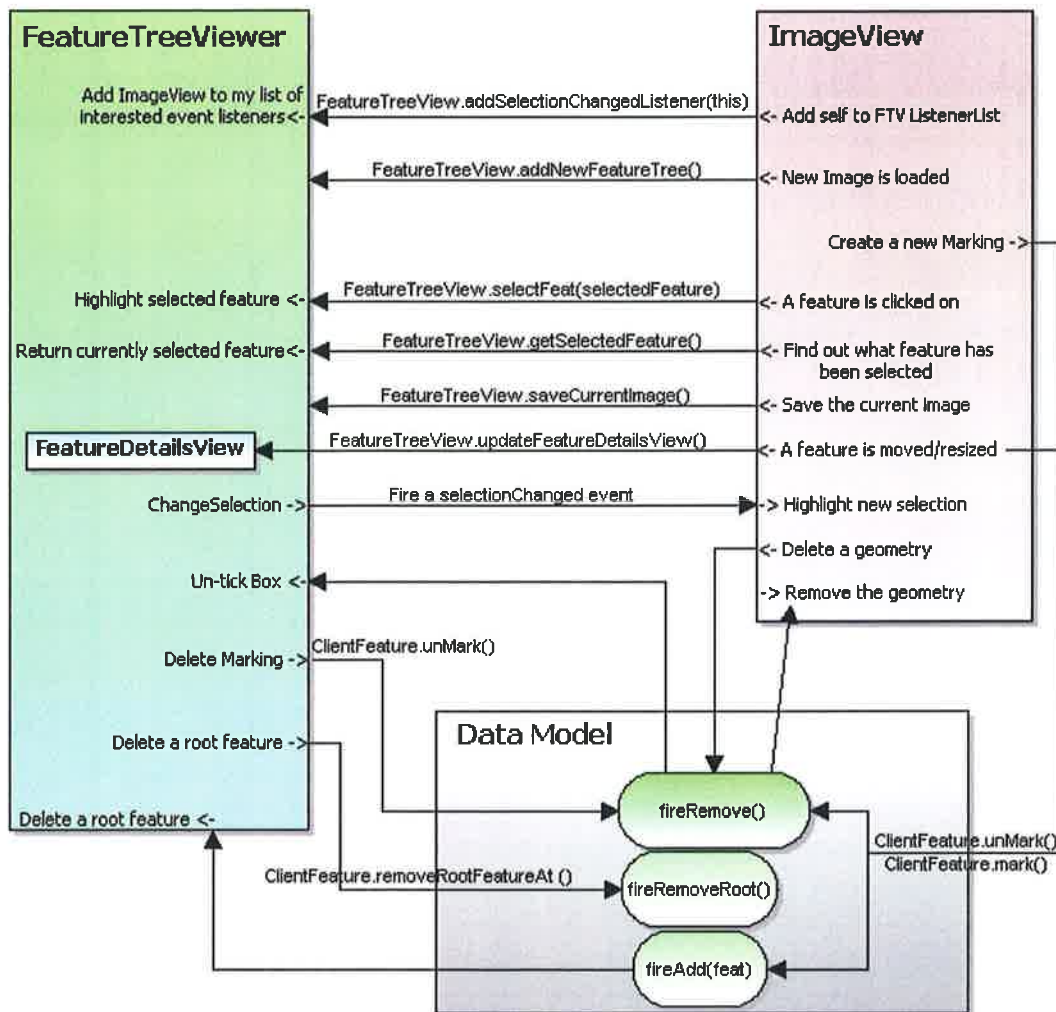


Figure 5.6: **Exchanges Between the Views**

Fig 5.6 above attempts to show user initiated changes that will affect both views, as well as the model in some cases. For the sake of simplification, the above diagram leaves out some system details, such as fireUpdate() which will be called and fireAddRoot() which is similar to fireRemoveRoot(). In any case, the details of the entire architecture illustrated in Fig 5.6 will be tackled below.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

ImageView

Features present in ImageView include:

A. Click on a geometry

When the user clicks on a geometry shape to select it in the ImageView, this feature must be selected in the FeatureTreeView. To do this, the ImageView class has a reference to the TreeView class, and it simple executes the TreeView.selectFeat(feature) method. This ensures the same feature is selected in both views.

B. Move a geometry

If a geometry is moved to a new location in the ImageView, then the FeatureDetails view must be updated to show the new geometry data. First of all, the ImageView class must call ClientFeature.unmark() and ClientFeature.mark(newMarking) on the ClientFeature that has been moved, updating it with the new markings. These ClientFeature methods will in turn call fireRemove() and fireAdd(feature) in the ClientFeature so that all listeners will be updated with the new marking. After this, the method getTreeView().updateFeatureDetailsView() is called in ImageView, updating the tree view's instance of FeatureDetailsView with the new geometry information.

C. Resize a geometry

If a geometry is resized by the user using the mouse in the ImageView, then the FeatureDetails view must be updated to show the new geometry data. Same as above, the ImageView class must call ClientFeature.unmark() and ClientFeature.mark(newMarking) on the ClientFeature that has been moved, updating it with the new markings. These ClientFeature methods will in turn call fireRemove() and fireAdd(feature) in the ClientFeature so that all listeners will be updated with the new marking. After this, the method getTreeView().updateFeatureDetailsView() is called in ImageView, updating the tree view's instance of FeatureDetailsView with the new geometry information.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

D. Delete a geometry

If the user clicks on the geometry in the ImageView, and hits the delete key to remove it from the photo, then – over in the tree view - the tickbox opposite the node for this feature must be unticked to show that the feature is now unmarked. To do this, the user hits the delete button and the method selectedFeature.unmark() is called, selectedFeature being the ClientFeature to which the unmark method is applied. This unmark() method will fire a remove event in the ClientFeature. Because the corresponding tree view node for this feature is registered as a listener for that ClientFeature, also its remove method (as discussed earlier) will be called, and this will be handled with in the tree view by unticking the box opposite the node.

FeatureTreeViewer

A. Change the selection

When the user clicks on a ClientFeature node in the FeatureTreeViewer, it becomes the currently selected item. It must be ensured that the marking for this feature is highlighted in the ImageView (if this feature is marked). The information in the FeatureDetailsView must also be updated to show data for the newly selected feature. In order to carry out all this, the ImageView class must actually be registered as a listener in the FeatureTreeViewer class – similar to the way ImageView and FeatureTreeViewer are registered as listeners in the ClientImage/ClientFeature structure. This means that when the user selects a different node, the fireSelectionChanged() event is fired in the FeatureTreeViewer. When it is fired, the implementation for selectionChanged(SelectionChangedEvent event) is executed in the ImageViewer class, and this method casts the event argument to a IStructuredSelection instance, from which the first element should be the ClientFeature that must be highlighted in ImageView.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

B. Delete the Marking for a particular selection

If the user selects an item in the tree view, and then hits delete (either the onscreen delete button, or the keyboard delete button) then the marking is removed from this feature. To do this, FeatureTreeViewer calls the method ClientFeature.unMark() on the feature concerned. This method will in turn fire the remove event for all listeners in the feature's ListenerList:

- FeatureTreeViewer's content provider class is in the ListenerList, so the remove method is triggered in the FeatureTreeContentProvider class, and this method simply updates the tree view interface with the latest version of the feature. Because this feature is not marked anymore, the tickbox opposite this node will be unticked. FeatureTreeViewer will also update the FeatureDetailsView by calling upDateFeatureDetailsView().
- ImageView is also in the ListenerList for the ClientFeature. Thus, when ImageView finds out the remove event has occurred it will update itself accordingly: as seen earlier, it will get the geometry from the removed ClientFeature and will remove it from the displayedMarkings list. Then the ImageView will be repainted without the removed shape.

C. Delete a root

If the user selects an item in the tree view and hits the "Delete Root" button, then the root feature – in our given examples it is a face – will be removed from the tree, along with all of its sub features. For this to happen, after the button is clicked, it triggers the method deleteSelectedRoot() in FeatureTreeViewer, which then calls currentImage.removeRootFeatureAt(cfeat.getIndex()), where currentImage is the ClientImage instance from which the root must be removed. The method ClientImage.removeRootFeatureAt(index) fires a removeRoot event, and as explored earlier, both ImageView and FeatureTreeView have implementations to deal with the removeRoot:

IT Sligo An Institiúid Teicneolaíochta, Sligeach

- In ImageView, all markings for the removed root feature and all of its children will no longer be displayed
- In FeatureTreeViewer, the node representing this feature will be removed from the onscreen structure, thus removing all sub-nodes.

D. Create a new root

If the user clicks the "Add Root" button, then a new unmarked root feature will be created, along with all of its sub features. This does not affect the ImageView. However, the structure of the ClientImage being used must be changed, so FeatureTreeViewer calls the method addNewFeatureTree("face"), within which is called this.currentImage.addEmptyFeatureTree("face"). The ClientImage.addEmptyFeatureTree(featName) method, as explored earlier, adds another root feature to the ClientImage structure, and automatically creates empty features for all possible sub features as defined in the XML schema.

5.4 Integration with the Image Testing and Reporting Framework

Now that the image marking tool has been implemented, it fits into the overall framework as an Eclipse defined perspective. This ensures that the overall project requirements of having an integrated overall testing framework are fulfilled.

5.4.1 Java Perspectives

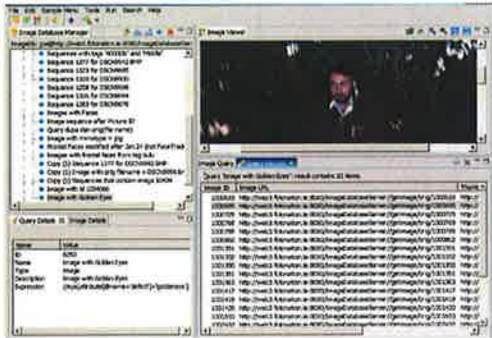
Separate application components have been developed and implemented within the same application as Eclipse perspectives simplifying the task of the user. The following benefits are immediately obvious:

- A user carries out image marking, categorisation, testing and reporting within a single complete environment
- Each application component is seen as an individual tool
- The workload of the user is divided
- Users are given a clear distinction between different stages of the marking, testing and reporting framework

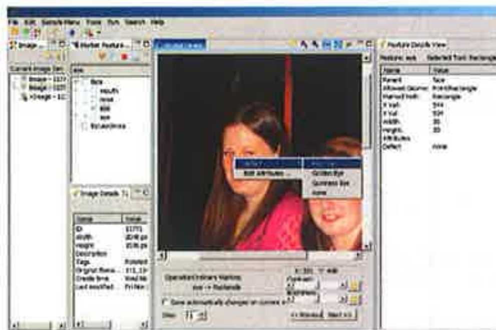
IT Sligo An Institiúid Teicneolaíochta, Sligeach

The perspectives available are:

I. The Image Database Tool

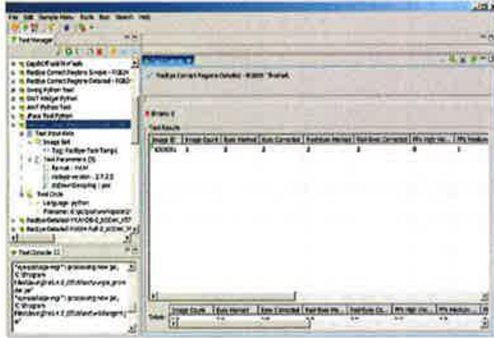


II. The Marking Tool



IT Sligo
An Institiúid Teicneolaíochta, Sligeach

III. The Testing Tool



Here is a scenario to illustrate how the different perspectives come into play:

- A user takes photos of his car with a digital camera, and then transfers them to the computer
- The user loads the images into the database tool, and places them all under the tag "car"
- The user then opens the marking tool and searches for images by tag. When he/she finds images with the tag "car", he/she selects them to be loaded into the marking program
- The user marks all the images using the marking tool, tools will be made available to mark vehicle objects (based on a predefined XML schema).
- Next, the user opens the reporting tool and runs a test on the marked images. The test reveals how many marked vehicle objects were detected by the vehicle detection algorithm. Detailed reports can then be viewed and printed out to determine how the algorithm has performed.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

5.4.2 Sharing Data Throughout The Application

After an image database has been set up, images are uploaded using the Image Database Tool. In this tool, images can be imported, exported, assigned tags. Facilities are available in the database tool application to allow the user to query by tag or by search criteria, to specify images for retrieval. Usually queries will specify something to the effect of "Find all images that contain instanced of red-eye". The database created for this project contains a large set of images with variability in scale and location, faces in various poses and colour differences. Images were captured using digital cameras, and continuously uploaded to the database to help diversify the test set.

The images uploaded in the Image Database Tool are used in the Marking Tool. The Marking Tool queries the image database and gets back a set of images. This set of images is marked, and thus metadata is added to images.

Lastly, algorithms are run using the Testing Tool. Again, algorithms are run on the same set of images. Results of algorithms are also compared to the image ground truth data that was gathered using the marking tool.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

5.4.3 Goals Fulfilled

- The framework should thus promote and bring about the efficient, practical and precise testing and reporting of algorithm performance through the use of improved tools
- The Marking Tool allows for the gathering of much more accurate image markings, which means accurate ground truth data, which promotes a consistent a more reliable algorithm testing process
- The semi-automatic nature of the framework means that algorithm performance assessment is carried out quickly and easily, greatly reducing testing time
- The framework thus incorporates tools to mark image-sets for testing purposes, run algorithm tests on image sets based on test parameters and generate reports on algorithm performance

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Chapter 6: Testing

In this chapter, an overview is given of the testing techniques employed for the Marking Tool application.

6.1 Testing Strategy

Software testing is essentially a part of software development, and as such must be carefully conducted in order to uncover serious errors in code and in interface design. When discovered anomalies are corrected, there should come about an increase in the completeness and consistency of a given application. A few questions had to be asked when formulating a test plan for the Marking Tool application. Questions such as; what types of testing must take place? What application components must be tested? How long should be spent testing them? How thorough must the tests be? For this type of application would white box (implementation and code based) testing or back box (interface and specification based) testing be preferable? A little research was necessary to find out what the best approach to testing would be.

"Realistic test planning is dominated by the need to select a few test cases from a huge set of possibilities. No matter how you try, you will miss important tests. No matter how careful and thorough a job you do, you will never find the last bug in a program, or if you do, you won't know it" (Kaner et al, 1993:18).

One of the certainties about testing any system is that it is not possible to test every possible combination of uses for each and every system component. Therefore, a realistic number of tests must be carried out within a practical allocated timeframe. The challenge with the Marking Tool is that many different facets of application development must be dealt with – there is the data model, there is the processing that goes on when data is taken from the interface, validated, and stored in the data model, there are various interface components that must each be tested for consistency, there is client/server interaction, there are data retrieved from XML files and used at various points in the application, and data must also be stored back to the server. Thus a single approach to

IT Sligo An Institiúid Teicneolaíochta, Sligeach

testing may not necessarily cover all the bases for this particular application. If using unit tests, all aspects of the data model may be inspected and tested, but what happens when the data model interacts with the user interface and the user attempts to store and retrieve data through the data model? Similarly, the user interface may look good, but if the data collected through it are not dealt with and processed correctly, anomalies will occur underneath the surface. Looking at all these aspects, it appears that the best approach to testing is one where different techniques are combined together to ensure testing has been conducted as thoroughly as possible. The hybrid approach described by Roper in "Software Testing"(1994) offers a good solution by combining both white box and black box techniques. For the Marking Tool, the hybrid approach to testing brought about the following set of stages:

1. Carry our functional testing on the user interface (black box testing)
2. Carry out unit testing on the model objects (white box testing)
3. Conduct usability tests with users to validate system design and identify

Because time was an issue throughout the final testing stages of the project, usability tests were often conducted alongside the functional code testing (white and black box). In both stages 1 and 2 above, the ultimate aim is to reveal bugs in the code and to rectify processing errors. This is done by the creation of tests cases – test cases should attempt to execute each and every possible combination of real-world uses that could occur when the application is operated by normal users. In stage 3 above, the aim is to find out whether or not the design works; is it intuitive, easy to understand, efficient, does it present unseen difficulties, etc. Overall, a mixture of black box, white box and usability tests should help to reasonably test and validate the application code, specification, and design for the Marking Tool.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

6.2 Functional Testing

Both white box and black box testing will be employed in order to test the functionality of the proposed application. Such testing should help to accurately evaluate whether or not the implementation satisfies the specified requirements accurately.

A. Black Box Testing

This method of testing checks that the code in the program is working correctly based on the behavior of the program. The tester does not look at actual code, but instead checks to see if the program carries out a task successfully or not. To build up a set of tests the specification documents for the Marking Tool application are consulted, and these documents cover all the inputs, outputs, and program functions for the application. The test data should ascertain whether or not the application is behaving as expected in all possible circumstances, and also if it fulfills the specified requirements.

Functional tests for the Marking Tool were carried out as follows:

I). Mark an image

Functional cases

F1:	Select feature to be marked in feature tree view. Mouse down on the image in the ImageView, draw shape, mouse up. Ignore pop-up menu and change selection.
F2:	Select feature to be marked in feature tree view. Mouse down on the image in the ImageView, draw shape, mouse up, select a property from pop-up menu.
F3	Select feature to be marked in feature tree view. Mouse down on the image in the ImageView, draw shape, mouse up, select "Edit Attributes" from pop-up menu. In edit attributes enter [VALID DETAILS]

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Invalid Cases

I1:	Select feature to be marked in feature tree view. Mouse down on the image in the ImageView, draw shape, mouse up, select "Edit Attributes" from pop-up menu. In edit attributes enter [INVALID DETAILS]
I2:	Select an already marked feature in the feature tree view. Mouse down on the image in the ImageView, attempt to draw shape, mouse up.

II). Delete Feature Markings

Functional cases

F1:	Select an item in the feature tree view. Press the delete key on the keyboard
F2:	Select an item in the feature tree view. Press the delete delete button located at the top of the feature tree view.
F3:	Click on a shape in the ImageView. After it has been selected, press the delete key on the keyboard
F4:	Click on a shape in the ImageView. After it has been selected, press the right mouse button down, and from the pop-up menu, select "Delete Marking".

Invalid Cases

I1:	Select an unmarked item in the feature tree view. Press the delete key on the keyboard
I2:	Select an unmarked item in the feature tree view. Press the delete delete button located at the top of the feature tree view.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

III). Test Undo and Redo

Functional cases

F1:	Mark four features in a face. Create a new face, mark four more feature within it. Undo the last six actions. Mark one more feature in the first face. Mark one more feature in the second face. Undo the last four actions.
F2:	Mark three features in a face. Create a new face, mark three more feature within it. Delete a marking from the first face. Delete a marking from the second face. Undo the last four actions. Delete a feature in the first face. Undo the last action. Delete a feature in the first face. Add a feature in the second face. Undo the last three actions.
F3:	Mark four features in a face. Create a new face, mark four more feature within it. Undo the last six actions. Mark one more feature in the first face. Mark one more feature in the second face. Undo the last four actions. Redo the last four actions.
F4:	Mark three features in a face. Create a new face, mark three more feature within it. Delete a marking from the first face. Delete a marking from the second face. Undo the last four actions. Redo the last four actions. Delete a feature in the first face. Undo the last two actions. Redo the last two actions. Delete a feature in the first face. Add a feature in the second face. Undo the last two actions. Redo the last two actions.

Invalid Cases

I1:	Mark two features in a face. Create a new face, mark two more feature within it. Undo the last six actions. Redo the last six actions.

An Institiúid Teicneolaíochta, Sligeach

IV). Edit Markings (move, resize various points, redraw)

Functional cases

F1:	Mark a feature with a linear ring consisting of ten points. Using the mouse, drag-and-drop each point to a new location. Execute redraw methods (Zoom in x 4 and zoom out x 2. Rotate clockwise x 2. Rotate anti-clockwise. Scroll up and down. Scroll left and right).
F2:	Mark a feature with a rectangle. Using the mouse, drag-and-drop each point to make the shape larger. Execute redraw methods.
F3:	Mark a feature with a rectangle. Using the mouse, drag-and-drop the entire shape to a new location. Execute redraw methods.
F4:	Mark a feature with a point. Using the mouse, drag-and-drop the point to a new location. Execute redraw methods.
F5:	Mark a feature with an ellipse. Using the mouse, drag-and-drop each point to make the shape larger. Execute redraw methods.
F6:	Mark a feature with a rectangle. Using the mouse, drag-and-drop the entire shape to a new location. Execute redraw methods.

V). Zooming In/Out, rotation, scrolling and resizing

Functional cases

F1:	Place five different markings on the image. Zoom in x 5. Zoom out x 15. Zoom in x 30. Press "Fit to screen" button located at the top of ImageView. Press "Actual size" button located at the top of ImageView.
F2:	Place five different markings on the image. Rotate clockwise x 2. Add two new

IT Sligo An Institiúid Teicneolaíochta, Sligeach

	markings. Zoom out x 3. Show actual size.
F3:	Zoom in x 20. Place two markings on the image. Zoom out x 10. Place two new markings on the image. Zoom out x 5. Scroll to the right by approx. three inches. Place two new markings on the image. Show actual size. Rotate clockwise x 3. Place a new marking on the image. Zoom out x 3. Zoom in x 10.
F4:	Zoom in x 20. Place two markings on the image. Show actual size. Rotate 3 x anti-clockwise. Place a marking on the image. Scroll to the right. Scroll down. Zoom out x 10. Place a marking on the image. Scroll up. Scroll down. Zoom out x 2.

Invalid Cases

I1:	Zoom in x 200.
I2:	Zoom out x 200.

VI). Delete a root node

Functional cases

F1:	In the tree view, create three root features using the "Add root" button at the of the tree viewer. In each root feature, mark three sub features. Select the middle root feature and click the "Remove root" button located at the top of the tree viewer. Select the bottom root feature and click the "Remove root" button located at the top of the tree viewer. Create two more root features using the "Add root" button at the of the tree viewer. For the two new root features, mark one sub feature within each. Now remove the top root feature, then the bottom, then the middle.
F2:	Add ten new root features. Mark each of the root features in turn. Delete the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

	fifth root feature down. Delete the last root feature. Delete the first root feature. Delete the last root feature. Delete the second last root feature. Add two new root features. Delete the first root feature. Delete the first root feature.

Invalid Cases

I1:	One root feature is present – delete this root feature. Now press the "Remove Root" button again three times. Press the "New Root" button. Mark the root feature. Now press the remove root button two times.

VII). Setting properties (from both treeview and imageview)

Functional cases

F1:	In the tree view, create one root features using the "Add root" button at the of the tree viewer. Create a marking for the face feature. Select the face marking in the image viewer, and right click on it. From the pop-up menu, select "Edit Attributes". In the dialog box that appears, enter name "Pat Reynolds"
F2:	In the tree view, create one root features using the "Add root" button at the of the tree viewer. Create a marking for the face feature. Right click on the face feature in the tree viewer, and from the pop-up menu, select "Edit Attributes". In the dialog box, enter name "Pat Reynolds".
F3:	In the tree view, create one root features using the "Add root" button at the of the tree viewer. Create a marking for the face feature. Right click on the face feature in the tree viewer, and from the pop-up menu, select "Edit Attributes". In the dialog box, enter no name and delete any default characters for the name.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

VIII). Image navigation

Functional cases

F1:	Load a new set of images into the image list view. Click the first image in the sequence. Click the "Next image" button located at the bottom of the ImageView. Mark one of the root features in the image. Click next image, mark a root feature in the image. Click previous image. Click next image.
F2:	Load a new set of images into the image list view. Click the first image in the sequence. Click the "Next image" button located at the bottom of the ImageView. Continue to click next image until the last image is reached. Mark a root feature in the image. Click next image. Click previous image. Click previous image. Click next image.
F3:	Load a new set of images into the image list view. Click the first image in the sequence. Mark a root feature in the image. Click the "Next image" button located at the bottom of the ImageView. Mark one of the root features in the image. Click next image, mark a root feature in the image. In the image list view, click the first image in the list. Click next image. Click previous image.

Invalid Cases

I1:	After loading a set of images into the image list view, select the last image in the list. Mark a root feature in the last image. Click next image. Click next image. Click previous image.
I2:	After loading a set of images into the image list view, select the first image in the list. Mark a root feature in the image. Click previous image. Click previous image. Click next image.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Note: For all above cases, the assumption is made that a query for a set of images has already been sent to the database, and a set of images has successfully been downloaded to the client application, appearing in the image list view.

The functional cases listed above provided clear and effective tests, covering as large a domain of application functionality as possible within a reasonable timeframe. It can be clearly seen how the established test cases are in line with the requirements for the application.

Using the Functional Cases and Invalid cases helped reveal erroneous program components and problem areas in the application. Ultimately, all test cases were carried out without anomalies, and invalid cases did you lead to unexpected behaviour. While such testing is fundamentally "integration testing", because the application was developed with modularity in mind, error correction was made easy, and errors were resolved by isolating a problem to a particular module. Earlier design decisions proved to be quite useful in this regard, especially when tackling the interaction between the data model and the various views in their respective modules.

Here is a sample of some errors revealed through functional testing:

- When an image was scrolled up or down and then rotated, markings did not relocate to the correct rotated coordinates on the image
- If an image was zoomed in on and then scrolled, markings were not consistently displayed in the same location on the image
- Often when clicking on an item in the tree viewer, the corresponding feature marking was not always selected correctly. Sometimes the index retrieved for the image viewer item was incorrect
- Using the application continuously for more than one hour caused it to crash
- It was not possible to add face attributes

IT Sligo An Institiúid Teicneolaíochta, Sligeach

B. White Box Testing

With white box testing, actual code modules are both inspected and executed in order to reveal errors and ascertain that all required tasks are being carried out in a correct manner. Such testing should take place before application components such as the core data model and graphical user interface components are integrated (such integration testing takes place later on) in order to keep errors to a minimum.

The most fundamental forms of white box testing for this application are:

- Developers Tests – such tests are conducted by the developer as he or she writes actual code, and adds new functionality to old code. Code should be checked and verified wherever possible, and testing is conducted in an organized manner.
- Unit Tests – unit testing checks the correctness of several application classes and components, often in an automated manner. External data sources like databases and XML documents are all used in these tests.

A combination of both of the above testing categories was used, and this served to satisfactorily reveal errors at early stages of development.

<u>Unit Testing In Practice</u>

While developers tests can be easily conducted at the developers discretion, a more regimented way of conducting unit testing is required. Unit testing can become a time consuming affair, and it is often difficult to ensure that new code changes will not interfere with previous components of code.

Thankfully, because the Eclipse programming environment is used for the development of the Marking Tool, a useful code testing framework known as JUnit is made readily available. JUnit helps to speed of and automate the testing of code components, helping to ensure that code behaves as expected. It was found that JUnit allows for the fast creation of unit tests, and also allowed for some regression testing when new code was added to the application, for instance a new method in one of the data model objects.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

With JUnit, testing is carried out by going through the following steps:

- Write test cases in Java
- Compile the test cases
- Run the resultant classes with a JUnit Test Runner

The best way to explore how unit testing was conducted for the Marking Tool is to go through some examples, showing how JUnit works.

Working with JUnit

Both ClientImage and ClientFeature were tested using JUnit and their corresponding JUnit test classes are good examples of how this testing framework can be used. In this section, a few of the more relevant tests will be explored. JUnit tests proved quite important in ensuring that the data model on the client was consistent and was behaving as expected.

I. Initialising JUnit

A test is created in JUnit by extending the *junit.framework.TestCase* class. For the Marking Tool a class `ClientFeatureTest extends TestCase` was created. This classes main method calls the *junit.textui.TestRunner.run(Class)* method and passes the test class (`ClientFeatureTest`) as an argument. The run method invokes any method that begins with *test*. Thus, for each method that needs to be tested, there will exist within ClientFeatureTest various methods beginning with test, e.g. testGetClientGeometry, testEditMark, testGetMarkedFeatures, and so on.

Within the constructor for ClientFeatureTest, all necessary objects are set up such as a list of test images and the testing directory. It also contains a couple of debug assertions to ensure that there are images in the image list – ClientImage and ClientFeature cannot be tested unless we have some images to test on.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

II. JUnit Tests

The actual test code uses methods from the TestCase super class to assert particular conditions throughout the test run. In most cases, some processing is performed and then values are compared or checked in order to ascertain whether object contents are as expected or otherwise. If an assertion fails, it is reported. Examples of assertions include the following:

- assertEquals(int expected, int actual)
- assertTrue(String message, boolean condition)
- assertNotNull(Object object)

Looking at test code, each test method is named after the object method that it aims to test. For instance, testGetAttribute tests the getAttribute method in ClientFeature.

In the first test example the getChildAt (index) method is tested. First of all, a rootFeatureList is created consisting of a single root feature "face". This list is handed into the addEmptyFeatureTree method, which then initializes all subfeatures for the root feature face. testClientImage.getChildAt(0) returns the ClientFeature at element 0 within testClientImage feature tree. It is already known that this should be a face, as it was added, so assertNotNull is called to make sure a null value was not returned. If a null value was returned then this will be reported.

```
public void testGetChildAt() {

	System.out.println(">>>>>> TEST: editMark <<<<<<<");

	ArrayList rootFeatureList = new ArrayList();

	String rootName = "face";

	rootFeatureList.add((Object)rootName);

	testClientImage.addEmptyFeatureTree(rootFeatureList);

	ClientFeature cf = testClientImage.getChildAt(0);
```

IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
        assertNotNull(cf);
        System.out.println("Got child at 0 " + cf.getName());
}
```

The following piece of code shows how the getAttribute method of class ClientFeature is tested. Before anything is tested, it must be ensured that the feature tree within the testClientImage contains ClientFeature objects, so again, the feature tree is initialized with a single face root feature. When this happen, default attributes for each feature are set. One of these attributes is the skintone, so if the getAttribute method needs to be tested, then it is certain that it should return a value when asked for the skintone. testClientImage.getChildAt(0) returns the ClientFeature at element 0 within testClientImage feature tree (it is already known that this should be a face, as it was added). Next, the getAttribute method is executed with the “skintone” argument – thus the method should return the attribute whose name matches “skintone”. assertNotNull(att) ensure that an attribute is returned correctly. If not, it will be reported.

```
public void testGetAttribute() {
        System.out.println(">>>>>> TEST: getAttribute <<<<<<");
        ArrayList rootFeatureList = new ArrayList();
        String rootName = "face";
        rootFeatureList.add((Object)rootName);
        testClientImage.addEmptyFeatureTree(rootFeatureList);
        ClientFeature cf = testClientImage.getChildAt(0);
        Attribute att = cf.getAttribute("skintone");
        assertNotNull(att);
}
```

The next test method shows how the editMark method is tested. As before, the ClientImage object is initialized with a single root face feature. After that has been done, an ellipse marking is created and is added to the ClientFeature in the feature tree. The

IT Sligo An Institiúid Teicneolaíochta, Sligeach

data for the marking is printed to the screen, then the marking is changed, and the method editMark is called on the ClientFeature, handing it the new ellipse marking. To ensure that the ClientFeature has been updated correctly using editMark, `assertEquals(newEl, cf.getMarking())` is called, and this will report success if there is a match, and ClientFeature has been updated correctly with the new marking, or false if both arguments are not equal.

```
public void testEditMark() {
	System.out.println(">>>>>> TEST: editMark <<<<<<");
	ArrayList rootFeatureList = new ArrayList();
	String rootName = "face";
	rootFeatureList.add((Object)rootName);
	testClientImage.addEmptyFeatureTree(rootFeatureList);
	ClientFeature cf = testClientImage.getChildAt(0);
	Ellipse newEl = new Ellipse(12,12,12,12);
	cf.mark(newEl);
	newEl = (Ellipse) cf.getMarking();
	System.out.println("PreEdit:The geometry of child 0 is " +
	newEl.getX() + " : " + newEl.getY() + " : " + newEl.getHeight() +
	" : " + newEl.getWidth());
	newEl.setX(24);
	newEl.setY(24);
	newEl.setWidth(24);
	newEl.setHeight(24);
	cf.editMark(newEl);

	assertEquals(newEl, cf.getMarking());
	newEl = (Ellipse) cf.getMarking();
```

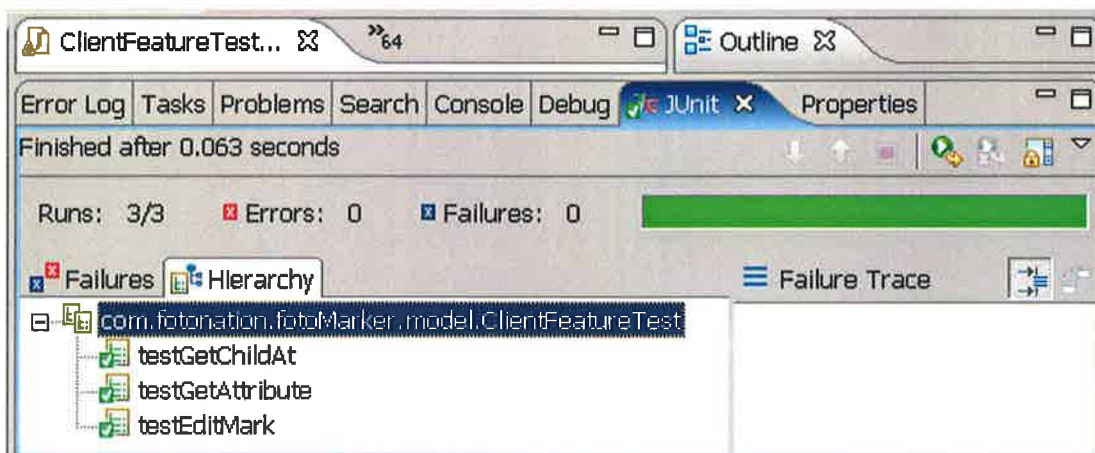

IT Sligo An Institiúid Teicneolaíochta, Sligeach

```
        System.out.println("PostEdit:The geometry of child 0 is " +
        newEl.getX() + " : " + newEl.getY() + " : " + newEl.getHeight() +
        " : " + newEl.getWidth());
}
```

After running the above three tests, the output in the Java console is as follows:

```
>>>>>> TEST: getChildAt <<<<<<<
Got child at 0 face
>>>>>> TEST: getAttribute <<<<<<<
>>>>>> TEST: editMark <<<<<<<
PreEdit:The geometry of child 0 is 12 : 12 : 12 : 12
PostEdit:The geometry of child 0 is 24 : 24 : 24 : 24
```

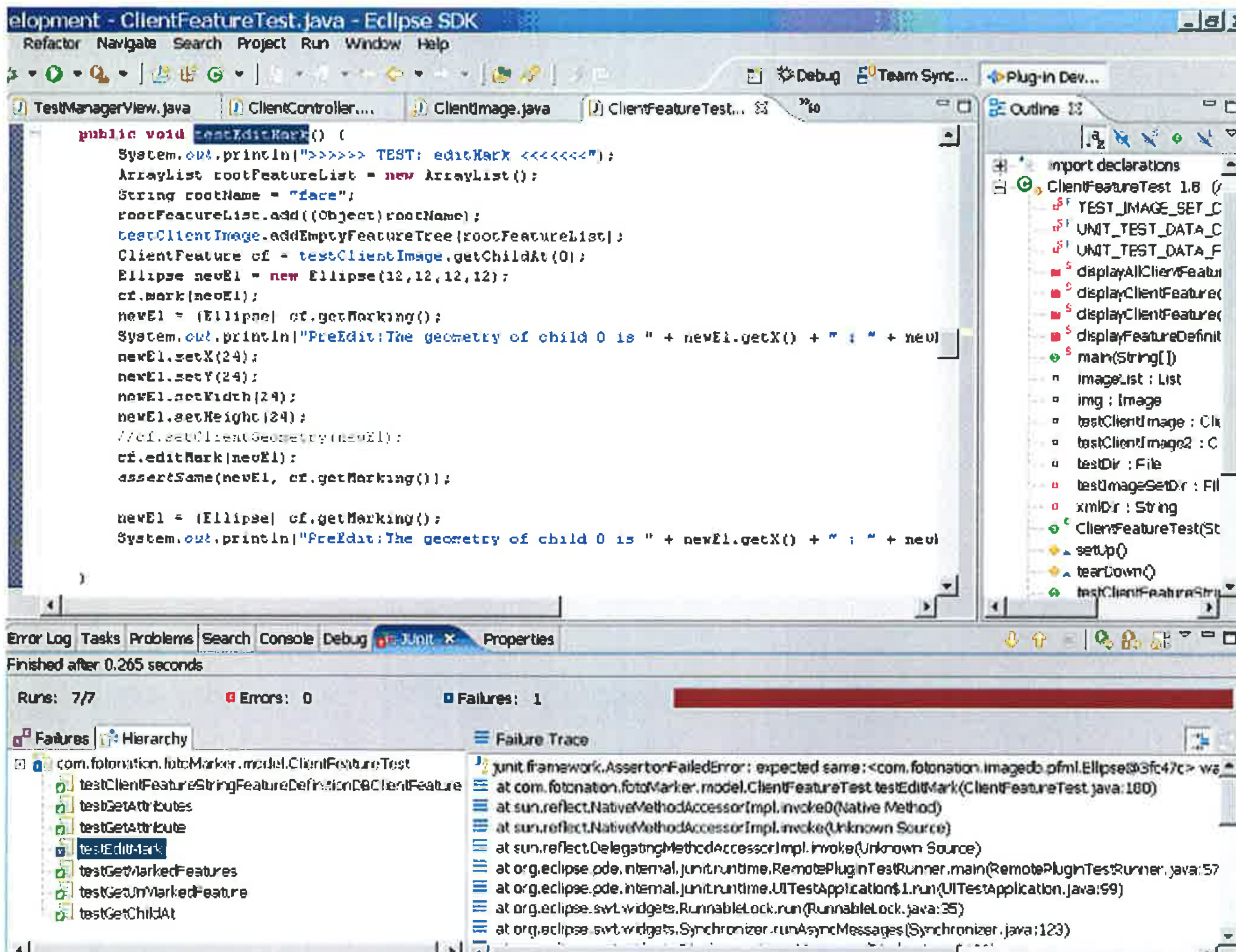
The JUnit interface shows that three tests have been successfully run with no failures or errors:



The above is just a small sample of the JUnit testing carried out for the Marking Tool. For the ClientFeature object alone, many other tests had to be carried out, such as: TestGetMarkedFeatures, TestGetUnMarkedFeature, testGetAttributes, testSetClientGeometry, displayAllClientFeatures, displayFeatureDefitnitions. For the ClientImage class tests also took place, for instance: testGetMarkup, testGetRootFeatures, testEquals, testAddEmptyFeatureTree.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

When there are problems, the JUnit interface makes it quite clear what part of the code is at fault, as seen below:



In the case above, one of the assertions within testEditMark had failed.

White Box Conclusion

Having used JUnit to develop tests early on in the process of implementation helped to understand the relationships between objects as well as the danger areas where errors are most likely to occur. Thanks to such white box testing techniques, it became easy to see how the tested program components would integrate when new components had to be

IT Sligo An Institiúid Teicneolaíochta, Sligeach

added. Additionally, for future maintenance of code new developers can easily gain an understanding into the various classes and methods by simply running JUnit tests.

6.3 Usability Testing

In this section, methods used to test the usability of the Marking Tool will be discussed. The research of Jakob Nielsen provided many useful starting points in the development of usability tests for this project, and ultimately led to some interesting and unforeseen application changes.

6.3.1 A look at usability

According to Jakob Nielsen, "Usability is a quality attribute that assesses how easy user interfaces are to use. The word "usability" also refers to methods for improving ease-of-use during the design process." (Nielsen, 2003). While developer tests and unit testing have helped to reveal logical errors in code and problems with functionality, there is also a necessity to assess the user interface. Although the system was initially designed by working with sample users and eliciting requirements from them, how accurate is the end result in representing those requirements within an actual system? And how do we define usability itself? In Nielsen's "Usability: 101" article (Nielsen, 2003), the following six quality components are given to help describe usability:

- **Learnability**: Is it easy for users to learn the system, and can tasks be accomplished easily?
- **Efficiency**: When the user is familiar with the system, is it possible to work quickly and carry out tasks with greater speed?
- **Memorability**: Is it possible to easily memorise the user interface components so that when the user returns to it after a period of non-use it is possible to quickly become proficient again?
- **Errors**: Does the user make a lot of errors using the system? Do these errors interfere with the system functionality, and can the user recover from the error?
- **Satisfaction**: Is the system satisfying and fun to use?
- **Utility**: Does the system help to carry out the tasks that users want?

IT Sligo An Institiúid Teicneolaíochta, Sligeach

The first five components refer to ease of use, but the last one refers more to actual functionality; as Nielsen would have it, there is little point in a system that is easy to use if it does not carry out the necessary tasks. Similarly, if a system carries out all desired tasks but is overly difficult to use, the system is not desirable.

6.3.2 Improving Usability

It is vital that the usability of the Marking Tool is somehow tested and improved if necessary. Thus, a study of the usability was required. According to Nielsen (2003), the most useful method is user testing. Based on Nielsen's guidelines, the following usability tasks were carried out for the Marking Tool:

- At the partner company a group of representative users were found, some members of which would be the end users of the system
- Users were requested to use the system to carry out their task – that of marking images – and were requested to carry out a set of steps to ensure all aspects of the system have been used
- Users provided reports on their use of the system, and were encouraged to voice any problems that were encountered

To help carry out the above, a questionnaire (Appendix C) was also provided to users after they had spent a good deal of time with the system. As well as this, hallway usability tests were carried out. This is where a person is selected randomly in the office as a testing candidate for the application that has just been developed. The feedback obtained from such usability testing helps to give an overall impression of how successful the system is in achieving its goals. A certain distance was maintained between developer and user in order to ensure that users were not influenced in any way.

6.3.3 Usability Test Results

After conducting usability tests, and consulting questionnaire results, it was found that the application had plenty of room for improvement. The application was installed at the

IT Sligo An Institiúid Teicneolaíochta, Sligeach

partner company, and eight people used the software for a couple of weeks to mark images. From observing the way users approached and used the application, and also the usage scenario for the most common task – mark all faces and eyes – a lot of useful information was gained regarding what features were unnecessary, and what ones may need to be improved. It was found that some of the features were not used very much, demonstrating perhaps that they were not appealing to the user and needed rethinking, or possibly removal from the application. Not only this, but in some areas the application exhibited undesirable behaviour that not been explored in functionality tests. Below is a sample of the usability testing findings.

Problems:

- Markings can be dragged and resized outside of the image boundaries, creating an erroneous visual effect that was found distracting and confusing
- It is possible to mark outside of the image canvas
- It is difficult to figure out what marking will be selected with a mouse click when there are multiple markings lying on top of each other

Desired Functionality:

- A copy and caste functionality is desired for the image marker – should not have to create every marking from scratch, especially when many eye markings are the same size and shape
- When navigating through images, the application should remember some of the previously entered properties as a lot of the time, people in the images may reoccur many times, so items like age, sex, skintone, etc. should not have to be re-entered time and time again. These details should be rememberd from one image to another by using something like a tickbox to ask the application to remember such details
- Within the application, there should be support for contrast and brightness adjustment for the image-viewer. If an image is too dark or too bright, a separate

IT Sligo An Institiúid Teicneolaíochta, Sligeach

application must be opened to brighten it, then it has to be imported all over again.

- For images that are part of a video stream, the application should predict where the next markings will be located
- It should be possible to add/remove a point in a LinearRing (polygon)
- The last attribute values used for a feature should be remembered
- It should be possible to move an entire LinearRing shape
- It should be possible to continue marking a closed polygon
- It should be possible to move an entire feature tree
- It should be possible to copy and paste an entire featuretree
- There should be support for PNG, TIFF and GIF images
- It should be possible to add an attribute to a face to state the horizontal orientation of the person, for instance frontal, left-profile, left-semi-profile, right-profile, right-semi-profile
- It should be possible to specify the age group of an individual, for instance baby, child, adult, old-age

Functionality Not Used Regularly:

- The QuickMark facility – allowing the user to mark many features of a particular type at a fast rate – was not used very much. Users do not want to mark all eyes, or all faces – they generally want to mark a face, then eyes

6.3.4 Usability Evaluation

After examining the usability test results, the application design was revisited and features that came under scrutiny in the usability test results were altered. Then additional usability tests took place to ensure that all implemented changes went towards improving the user experience.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

6.4 Conclusions of Test

One of the main benefits of letting user research drive design is that you do not have to spend time on features that users do not need. Early studies show you where to focus your resources so that you can ship on time.

The tests shown above were all executed successfully and the Marking Tool application is now in a fit state to be used seriously in the partner company.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Chapter 7: Conclusion and Recommendations

7.1 The Research Journey

The specific purpose of this research was the development of advanced tools to help streamline and automate the development of image processing algorithms. The project's principal motivation being the acceleration of algorithm testing and reporting procedures which are currently time consuming and often counter-intuitive, and also more efficient retrieval of images by using a specialised online image database.

A thorough literature review was carried out, helping to establish the context of the task of image marking and categorisation. Existing technologies and work practices were explored in the methodology chapter, with a view to pinpointing where and how current practices can be improved and reinvented through a new framework for testing. When formulating this framework, a systematic attempt was made to find out what tasks the client carried out using the existing image marking and algorithm testing tools. It was important to note the essential system components and time-frame constraints for the task of marking images and running image tests. Image storage, retrieval and the use of queries was also noted. One of the conclusions derived from exploration of current practices was that hand-marking hundreds of images can be troublesome if the tools used are limited; because the task becomes tiring, accuracy of markings may suffer. If the individual who marks images does not provide precise markings then the entire image algorithm testing process suffers because all tests rely on ground truth data for comparison – and ground truth data is achieved only through the image marking procedure. It was discovered that a new approach to the image marking and categorisation process would help to rejuvenate the testing of algorithms for the better. Resultantly, a set of detailed functional requirements were be drawn up and used for the development of the system.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

7.2 The Resultant Tool

The Image Marking Tool application fulfils many of the working requirements of the algorithm testing process. The framework has been tested and used in the partner company since the first fully working version was released towards the end of 2005, and has gradually – through various iterations – come to fulfil the demands served up by a commercial working environment. Now that the application is used daily, it has proven its usefulness, intuitiveness and robustness.

The aim of the thesis was not to drastically change or revolutionise the testing processes carried out in the image processing field, but looking at the application as it stands, this research may very well have discovered many efficient new ways of conducting and executing image processing tasks in the field of algorithm testing with regard to ground truth gathering, storage and manipulation. This work will hopefully highlight to algorithm testers in the industry the fundamental role of ground truth creation in the image processing field, as well as exhibiting the many advantages of a good image marking framework, good image marking tools, an intuitive and carefully developed interface and the vital link between onscreen interface components and the underlying data layer. Ultimately – when this work is combined with algorithm testing and reporting tools, as developed in the companion thesis - this work highlights the effectiveness of a marking tool to help speed up and streamline algorithm testing and ultimately algorithm improvement by the pinpointing of weaknesses at an early stage.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

7.3 Recommendations for Future Work in the Field

While conducting research for this thesis, the intriguing area of automated batch image marking and categorisation was stumbled upon. Put simply, this discipline aims to discover ways in which the process of hand marking images using a marking tool can be further sped up by letting detection algorithms aid in the marking process. In some cases, such technology has been used to attempt to track objects in a video stream – if there are 1000 frames in the video, then in each frame perhaps a person's face must be marked. In a situation like this automated marking is a must.

Regarding the use of similar automation techniques in the making tool developed for this thesis, this researcher sees such technology as a "helper" for the human image marker. For instance, if a person has to mark the faces in one hundred images, an automated marking tool would attempt to predict where all the faces should be marked on images by using a relevant detection algorithm. The user would then only have to quickly view each image to ensure the markings were placed in the correct positions. Depending on how advanced the algorithm is markings may need to be edited for accuracy. This kind of prediction tool would certainly be the next step for the marking tool if this researcher was to spend further time on this project. Incidentally, the detection algorithms used in the marking tool to help predict markings may very well be the same algorithms that are being tested later on – during the stage of algorithm testing and reporting.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

7.4 Conclusion

With the Image Marking Tool is in place, the partner company have reported higher productivity regarding the testing of algorithms using ground truth data (image markings). Ground truth data is easily obtainable online, editable, robust, accurate and modular. While further development, like the marking prediction tool (mentioned in 7.1) could be advantageous, the marking tool quite sufficiently carries out all roles that were specified in the requirements for this project.

APPENDIX A: XML Attributes

Dublin Core Attributes

The following DC attributes are used. Each attribute is encoded as an element under the image element.

- date -- date at which photo was taken
- title -- short caption
- description -- longer description
- creator -- photographer
- rights -- copyright and rights to distribute etc

In addition to DC attributes also have:

- created -- time at which image was added to the database (unix epoch time, UTC)
- lastmodified -- time at which image was last modified (unix epoch time, UTC)
- filename -- the original image file name if available

User Attributes

User defined. Examples: indoor/outdoor flag. User attributes should be prefixed with "user.".

EXIF Attributes

Standard EXIF data. Numeric values should be stored as numbers so that numeric comparitor operators can be used. EXIF attributes must be prefixed with "exif.".

Feature Markup

The actual markup for the image.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

APPENDIX B: Geometry Types Defined in the Schema

Point

A "Point" is a point in two dimensional pixel space defined by two integers (x and y).

- x
- y

Circle

A "Circle" is a circle defined by its center in two dimenaional pixel space (x and y) and a radius in integer pixels.

- x
- y
- radius

Computed attributes:

- area

Ellipse

An "Ellipse" is defined by its center in two dimenaional pixel space (x and y), a radius (of the major axis) in integer pixels and eccentricity.

- x
- y
- radius
- eccentricity

Computed attributes:

- area

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Rectangle

A "Rectangle" is defined by top left corner in two dimenaional pixel space (x and y), the width and height in integer pixels.

- x
- y
- width
- height

Computed attributes:

- area

LineString

A "LineString" is a piece-wire curve composed of line segments.

- 2 or more x,y coordinate pairs.

Computed attributes:

- length

LinearRing (aka Polygon)

- 4 or more x,y coordinate pairs. The first and last coordinate must be the same. A valid LinearRing must not self-intersect.

Computed attributes:

- area

Some non-geometrical attributes are as follows:

- name

IT Sligo An Institiúid Teicneolaíochta, Sligeach

- value

Attribute datatypes:

- integer
- double
- string
- complex (optional)
- selection (one or multi)

Computed attributes of features

These attributes apply to geometries. When added or modified to the database some computed attributes will be added which will aid in constructing queries. Example: length of LineString, area of Polygon, centroid of Polygon.

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

APPENDIX C: Usability Questionnaire

Rating scale to be used:

1 - strongly disagree

2 - disagree

3 - not sure

4 - agree

5 - strongly agree

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

I found it easy to pick up and use the system with little instruction required

The job of marking images was intuitive and fun to use

The system responded quickly and allowed me to mark many images in a reasonably short period

The steps involved in getting a new image, marking it and saving it were reasonably straightforward

The steps involved in going to a previously marked image to edit existing markings were reasonably straightforward

The steps involved in going to a previously marked image to delete an existing marking were reasonably straightforward

The image viewer area – where the image is displayed - provided enough detail for me to understand clearly and accurately where markings are located.

The information at the bottom of the image viewer was easy to read and understand.

The image attributes data, displayed below the image viewer, was easy to read and understand.

The tree viewer control helped me understand how I was marking the image as I was going along.

The tree viewer control was easy to read and understand

The tree viewer control made it easy to navigate through markings

It was always reasonably easy to tell which marking I was working on at a particular time – markings were sufficiently highlighted.

The quick mark marked tool was intuitive and relatively easy to use.

The quick mark all marked tool was intuitive and relatively easy to use.

Pop up controls for markings were readable and relatively easy to edit.

Zooming in and zooming out of the image works reasonably well

I used the zooming functionality a lot

Undo functionality was helpful

I would find such a system useful in my day to day job when I need to mark a large quantity of images.

What buttons, if any, did you find least useful? Most useful?

Are there any buttons (functions) that you would add to the system? Why?

If you ever deviated from the envisaged use of the system, what was the usual reason?

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Can you think of something the system could do to help prevent this?

What was the hardest thing to learn about using the system?

What feature did you particularly like?

Any additional comments about the system?

1. Does the new interface help to speed up the task of marking images?
2. Is it easy for the user to understand how to use the system based on it's visual appearance, or are some instructions required?
3. Does the system present any difficulties that prevent the user from carrying out tasks seamlessly?
4. Are parts of the system irrelevant or unnecessary?

Please rate the system from 1 to 5 based on the following measures. Feel free to leave a comment for any of the measures.

a. Efficiency ______________________________
b. User friendliness ______________________________
c. Pleasant to use ______________________________
d. Easy to remember ______________________________
e. Overall satisfaction______________________________
f. Potential future usage ______________________________

IT Sligo An Institiúid Teicneolaíochta, Sligeach

References

1998, "Evaluation and Validation of Computer Vision Algorithms", in *9th Workshop "Theoretical Foundations of Computer Vision"*.

The XML DB Initiative. http://xmldb-org.sourceforge.net/faqs.html . 1-2-2003.
Ref Type: Electronic Citation

A.C.Loui, C. N. J. S. L. "An Image Database for Benchmarking of Automatic Face Detection and Recognition Algorithms".

Alexander M.Bronstein, Michael M.Bronstein, & Ron Kimmel. 3D Face. Geometric Image Processing Laboratory, Institute of Technology Israel . 10-3-2003.
Ref Type: Electronic Citation

Cem Kaner, J.Falk, & Hung Quoc 1993, *Testing Computer Software* International Thompson Publishing.

Chandra Narayanaswami & M.T.Raghunath 2004, "Expanding the Digital Camera's Reach", *IEEE Computer Magazine*, vol. 37, no. 12.

Christopher Jaynes, Amit Kale, Nathaniel Sanders, & Etienne Grossman "A Scripted Multi-Camera Indoor Video Surveillance Dataset with Ground Truth", in *IEEE Workshop on Visual Surveillannce and Performance Analysis for Tracking and Surveillance*, Ctr. for Visualization and Virtual Environments and Dept. of Computer Science, University of Kentucky.

IT Sligo An Institiúid Teicneolaíochta, Sligeach

CNN. Twins crack face recognition puzzle. CNN . 10-3-2003.
Ref Type: Electronic Citation

Daniel A.Carp. Market Convergence:Advances in Imaging are Reshaping World Markets. Kodak Press Centre . 25-9-2002.
Ref Type: Electronic Citation

Dov Dori & Lio WenYin "Principles of Constructing A Performance Evaluation Protocol for Graphics Recognition Algorithms", Kluwer Academic, pp. 97-106.

Erich Gamma, R. H. R. J. J. V. 1995, *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional.

Francois Fleuret & Donald Geman 2002, *Fast Face Detection and Pose Estimation.*

H.Kang, T.F.Cootes, & C.J.Taylor "A Comparison of Face Verification Algorithms using Appearance Models".

Hongjun Xu. Digital Image Processing. 6-6-2003.
Ref Type: Slide

Ian Sommerville 2000, *Software Engineering.*

IDC Press Release. IDC Survey: IDC's European Consumer Digital Imaging Survey Highlights Key Growth Opportunity in the Home Printing Market. 19-1-2005.
Ref Type: Generic

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

Ilker Atalay. Face Recognition Vendor Test 2002 Review. 16-3-2002.
Ref Type: Generic
Keywords: FERET/recognition techniques/report/techniques
Notes: Report on Testing and Evaluation of image recognition techniques

Jakob Nielsen. Why You Only Need to Test With 5 Users.
http://www.useit.com/alertbox/20000319.html . 19-3-2000.
Ref Type: Electronic Citation

Jakob Nielsen. Success Rate: The Simplest Usability Metric.
http://www.useit.com/alertbox/20010218.html . 18-2-2001.
Ref Type: Electronic Citation

Jakob Nielsen. Usability 101: Introduction to Usability.
http://www.useit.com/alertbox/20030825.html . 25-8-2003.
Ref Type: Electronic Citation

Jef Raskin 2000, *The Humane Interface: New Directions for Designing Interactive Systems* Addison-Wesley Professional.

Jin-Hyuk Hong, Eun-Kyung Yun, & Sung-Bae Cho. A Review of Performance Evaluation for Biometrics Systems. International Journal of Image and Graphics . 24-3-2004. World Scientific Publishing Company.
Ref Type: Magazine Article

John A.Black, M.Gargesha, K.Kahol, P.Kuchi, & Sethuraman Panchanathan "A Framework for Performance Evaluation
of Face Recognition Algorithms".

IT Sligo An Institiúid Teicneolaíochta, Sligeach

K-K.Sung & T.Poggio. Example-Based Learning for View-Based Human Face Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 20[1], 39-51. 10-1-1998.
Ref Type: Magazine Article

Lei Zhang, Yuxiao Hu, Mingjing Li, Weiying, Ma, & Hongjiang Zhang "Efficient Propagation for Face Annotation in Family Albums", in *ACM Multimedia, New York, US, 2004*.

M.Roper 1994, *Software Testing* McGraw-Hill.

Margaret L.Johnson 2004, "Biometrics and the Threat to Civil Liberties", *IEEE Computer Magazine*, vol. 37, no. 4, pp. 92-91.

Ming-Hsuan Yang, D. J. K. N. A. "Detecting Faces in Images: A Survey", in *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 1 edn.

Mislav Grgic, K.Delac, & S.Grgic 2005, *Face Recognition: Hypothesis testing across all ranks* University of Zagreb.

MIT. MIT Database. 2-1-2006.
Ref Type: Data File

National Science Foundation. Advanced Technological Education Project. National Science Foundation . 22-8-2000.
Ref Type: Electronic Citation

IT Sligo
An Institiúid Teicneolaíochta, Sligeach

P.Courtney & N.Thacker 2001, "Performance Characterisation in Computer Vision," in *Imaging and Vision Systems: Theory, Assessment and AplicationsImaging and Vision Systems: Theory, Assessment and Applications*, NOVA Science Books.

P.J.Phillips, P.J.Flynn, T.Scruggs, K.W.Bowyer, J.Chang, K.Hoffman, J.Marques, J.Min, & W.Worek "Overview of the Face Recognition Grand Challenge", in *IEEE Conference on Computer Vision and Pattern Recognition, 2005.*

P.Jonathan Phillips, H. M. S. A. R. P. J. R. 2000, "The FERET Evaluation Methodology for Face-Recognition Algorithms", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 10.
Keywords: recognition techniques/techniques
Notes: Evaluation of image recognition techniques

P.Jonathon Phillips, A. M. C. L. W. M. 2000, "An Introduction to Evaluating Biometric Systems", *IEEE Computer Magazine*, vol. 33, no. 2.
Notes: Evaluation of the performance of emerging biometric technologies

Patrick Grother, Ross Micheals, Duane M.Blackburn, Elham Tabassi, Mike Bone, & P.Jonathon Phillips 2003, *Face Recognition Vendor Test 2002 - Evaluation Report.*

Patrick Grother & Ross Michealsand, P. J. P. Face Recognition Vendor Test 2002 Performance Metrics. 1-1-2005.
Ref Type: Generic
Keywords: - Identification,verification,validation,Techniques to evaluate algorithms
Notes: Detailed explanation of methodology and recognition performance characteristics used in the Face Recognition Vendor Test 2002

IT Sligo An Institiúid Teicneolaíochta, Sligeach

Patrick Grother, Ross Micheals, & P.Jonathon Phillips. Face Recognition Vendor Test 2002 Review. 2005.
Ref Type: Generic

Peter N.Belhumeur "Ongoing Challenges in Face Recognition", in *2005 U.S.Frontiers of Engineering Symposium Presentations*, National Academy of Engineering.

Prag Sharma & P.Reilly 2003, "A colour face image database for benchmarking of automatic face detection algorithms", in *Video/Image Processing and Multimedia Communications, 2003. 4th EURASIP Conference, 2-5 July 2003*.

Rogerio Schmidt Feris, Jim Gemmell, Kentaro Toyama, & Volker Kruger 2002, *Facial Feature Detection Using a Hierarchial Wavelet Face Database*, Microsoft Research.

Syed A.Rizvi, P.Jonathan Phillips, & Hyeonjoon Moon 1998, *The FERET Verification Testing Protocol for Face Recognition Algorithms*.
Keywords: FERET

W3C. XML Path Language. 16-11-1999.
Ref Type: Data File

IT Sligo An Institiúid Teicneolaíochta, Sligeach