Image Shape Classification Using Computational Intelligence and Object Orientation

Lukasz Antoni Machowski

A research report submitted to the

School of Electrical and Information Engineering
University of the Witwatersrand
Johannesburg
South Africa

in fulfilment of the requirements for the degree of

Master of Science in Engineering

Johannesburg
December 2004

Declaration

I declare that this is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg, South Africa. It has not been submitted before for any degree or examination at any other university.

Signed this _____ day of _____ 2004

_____

Lukasz Antoni Machowski

Executive Summary

With the increase in complexity of modern software systems, there is a great demand for software engineering techniques. Calculation processes are becoming more and more complex, especially in the field of machine vision and computational intelligence. A suitable object oriented calculation process framework is developed in order to address this problem. To demonstrate the effectiveness of the framework, a simple shape classification system is implemented in C#. A suitable method for representing shapes of images is developed and it is used for classification by a neural network. Sets of real-world images of hands and automobiles are used to test the system. The performance of the object oriented system in C# is compared to a functional paradigm system in Matlab and it is found that object orientation is well suited to the later stages of machine vision while the functional approach is well suited to low level image processing tasks.

Acknowledgements

I wish to thank the following people for their contributions to this project:

Prof. Tshilidzi Marwala, School of Electrical and Information Engineering, University of the Witwatersrand, for being such a motivating supervisor and an inspiring role model.

My Mother and Father, for all their hard work and support throughout the years. They have always pushed me to excel in what I do and to enjoy the privilege of studying. Their hard work and devotion to their fields has stood as an inspiration for me to persevere in this degree.
"If you do anything… do it with Passion!"

Robert and Edith, for always being interested in what I do and stimulating my mind with interesting discussions. Thank you for accommodating me in your lives.

Aleksandra and Rainer, for helping me through my years as a student.

Bradley van Aardt, for being available to discuss crucial aspects of this research.

Nicola, for all the love and support she has given me throughout this demanding time.

Foreword

This research report is presented to the University of the Witwatersrand, Johannesburg, South Africa for the degree of Master of Science in Engineering.

The research report is entitled “Image Shape Classification using Computational Intelligence and Object Orientation”. It is comprised of five chapters. The first introduces the research topic. The second describes a suitable method for representing shapes of images so that they can be classified by a neural network. The third chapter introduces the design of a calculation process framework for defining and performing complex calculations. The fourth chapter links the two concepts together and provides the essence of the research. The final chapter summarises the results from the research and combines the conclusions from the individual chapters. The appendices present aspects of the work that are not covered in great detail in the main document.

Appendix A presents additional work done on neural networks, including background information and possible uses.

Appendix B presents additional work done on image registration and optimisation techniques.

Appendix C contains the UML diagrams that detail the design of the system that has been developed.

Appendix D contains the actual source code and implementation of this research.

Table of Contents

List of Figures

Chapter 1: Introduction

The purpose of this study is to show that software object orientation is a suitable technique for designing and implementing a visual object recognition system using computational intelligence. In particular, an object oriented (OO) framework is developed for the later stages of the object recognition process where high-level information, such as shape, is used for the classification. The system uses pre-segmented images of objects and attempts to recognise which class the objects belong to. An object oriented design for the framework is well suited to the later stages of object recognition because they deal with relatively abstract concepts which are difficult to represent mathematically on a computer. Also, the complexity of the object recognition system at late stages of the recognition process becomes difficult to comprehend, manage, maintain and reuse without more advanced software engineering techniques.

Getting a machine vision system to recognize objects from images is a difficult and well studied problem and it makes use of many theoretical foundations. These range from highly mathematical techniques that rely on image processing, to neural-network-type learning systems. Being able to sense the environment using visual information has many applications in the commercial, industrial, medical and military fields.

The complexity of implementing such a system becomes a significant software engineering feat, especially when the later stages of object tracking require the use of high level (and possibly abstract) concepts which are not native to traditional computational methodologies.

This research investigates whether OO concepts are suitable for the later stages of object recognition, which include feature extraction and object classification. The system takes images of simple real-world objects as inputs and outputs the class of the object in the image. The sample data is generated by taking photographs of human hands showing different finger orientations. Side view images of motor vehicles are also used for seeing how the system performs on man-made objects. The system is implemented using an OO paradigm and is compared to the functional paradigm prototype. The performance between the two systems is compared in order to answer the research question of whether the OO approach is suitable for object recognition.

This research has significance for companies that are considering the use of OO for their future object recognition systems. It is typically the case that the complexity of their current system makes it extremely difficult to comprehend and manage the various aspects of their object recognition process. This research provides a starting point for migration from a functional paradigm to an OO paradigm.

The implementation language chosen for the implementation is C# which is a relatively new language and therefore has few computational intelligence toolboxes. It is also a high level language that has many constructs that support good software development techniques.

This report starts by describing a suitable method for shape representation. It then details the design of a calculation process framework that allows the creation of calculation objects. Both of these two concepts are combined in the last chapter to create an object oriented shape classification system.

From this research, the following papers have been published:

1. L. Machowski, T. Marwala. “Representing and Classifying 2D Shapes of Real-World Objects using Neural Networks”, Proceedings of the IEEE Conference on Systems, Man and Cybernetics, The Hague, Netherlands, pp. 6366-6372, 2004.

2. L. Machowski, T. Marwala. “An Object Oriented Calculation Process Framework”, Submitted for IEEE 3rd International Conference on Computational Cybernetics, Mauritius, 2005.

3. L. Machowski, T. Marwala. “Using an Object Oriented Calculation Process Framework and Neural Networks for Classification of Image Shapes”, Submitted for IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, Special Issue on "Advances in Heterogeneous and Complex System Integration", November, 2005.

Chapter 2: Representing and Classifying 2D Shapes of Real-World Objects using Neural Networks

Lukasz A. Machowski

Abstract *– A framework is presented which uses a polar representation of a segmented object for shape classification. This method produces a position, rotation and scale invariant representation of the shape. An efficient method for extracting multiple contours from the polar representation is used to handle the problem of many-to-one mappings in the radial and angular parameters. The contours are used to find interesting vertices of the shape. The shape information is mapped to spatial regions on a polar grid and fed into a multi-layer perceptron for classification. The framework is tested on manually segmented images of people's hands and on side views of automobiles. The results show that the network can achieve approximately 100% generalisation on test data even though the network is under trained.*

1. Introduction

Getting a machine vision system to recognize objects from images is a difficult and well studied problem [1]. The ability of humans to segment, classify and recognize objects with relative ease makes it a sought after ability for vision systems. Being able to sense the environment using visual information has many applications in the commercial, industrial, medical and military fields.

There are many different features of an image which can be used for classifying objects and these include colour, texture, motion, context and shape [1][2][3]. Shape information is used extensively for classification of objects in vision systems and is an important visual feature [4][5][6]. There are also several methods that can be used for performing the classification of the object and these include similarity distance measures, template matching, index calculation, hashing, maximum likelihood, clustering and neural network classification [2][7][8]. There are many advantages of using neural networks and these include being able to train the system using example images and having superior generalisation abilities when compared to other classification techniques.

This paper presents an improvement to the method described in [9] which uses a simple, yet effective method for shape representation. The contour points are converted into a normalized polar form with respect to the shape centroid [3][9]. This provides a rotation, scale and translation invariant description of the shape while still allowing the original shape contour to be retrieved from this representation. The problem of multi-value mappings must be addressed because they are common for real-world objects. This arises when the shape has concave sections as is shown in Figure 1. The corresponding polar representation is shown alongside.

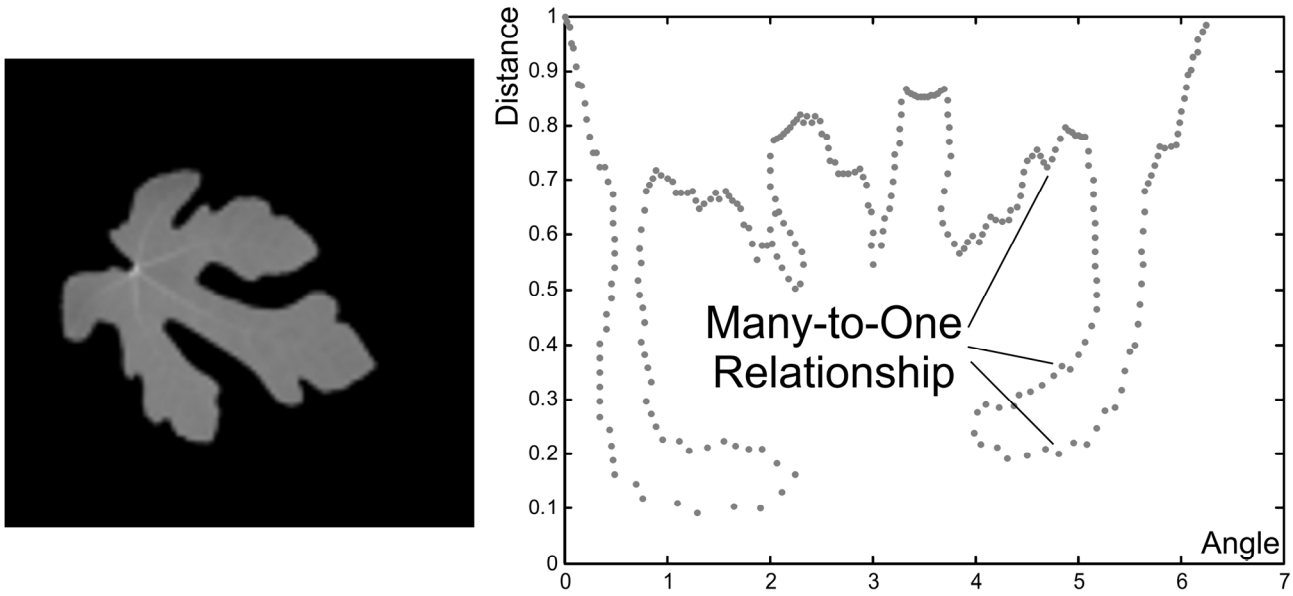


Figure 1 A leaf with concave sections and the problematic many-to-one polar representation.

1.1. Shape

Feature extraction is used to reduce the amount of information acquired from image data [3][7][8], while improving the ability of the vision system to classify the data [4][6][9]. Shape information is often the most characteristic visual feature of objects and is therefore used extensively for object recognition [1]. The shape of an object contains a large amount of information that is useful when trying to classify real-world objects.

There are several techniques for representing shape information [3][4][5][6][9][10][11] and many make use of the boundary of a segmented object to extract the shape information. The information can then be transformed to retrieve shape descriptors for classification of the shape or retrieval from a database [2][12]. It is desirable to have a representation that is easily computed and simple to interpret, especially if the system is to be implemented as an embedded system for applications such as automatic target recognition.

Acquiring the polar representation of a shape with respect to the centroid is an attractive approach because it is easily calculated and only requires the boundary information [10]. The major problem with this is that the centroid position is affected by noise and occlusion in the boundary information. However, the method described in [9] is relatively insensitive to typical variations which can be found in real-world objects. The advantage of using this method is that the shape information is position, scale and rotation invariant [3][9].

Another problem identified in [10] with the polar representation approach is that the function may be multi-valued for a certain class of objects. This is especially true for most real world objects, such as the leaf shown in Figure 1. The solution presented in [10] is ad hoc and an improvement is presented in this paper.

1.2. Neural Networks

Neural Networks are interconnections of artificial "neurons" that are greatly simplified versions of the biological neurons found in the human brain [13]. On a computer, the neural network is represented by a labelled acyclic directed graph with a clearly defined set of inputs and outputs [13][14]. The neural network may have any number of hidden units defined in an arbitrary amount of layers. There are several different architectures in existence, each one with its own advantages and disadvantages. The aim of the neural network is to be able to learn complex input-output relationships based on a source of training data, and then be able to make predictions for unseen data [13][14]. The neural network is used for its generalisation abilities and there is always a trade-off between generalisation ability and over-fitting [13]. It is possible to show that perfect generalisation with a neural network is not possible due to the fact that an infinite amount of neurons can produce output that is consistent with the training data [13].

There are three main regularization tools which can improve the performance of the network by ensuring that the network does not over fit the data and that it generalises well [15]. The first is early stopping where a sub-sample of the training data is kept separate to be used as test data. The progress of how well the network generalises for the test data is monitored while being trained. Typically a sweep is done and then the network weights that gave the lowest error for the test data are used for the network. This set of weights represents the network that has the best generalisation performance (based on the test data). The problem with early stopping is that we need to keep aside a subset of the training data as test data. For successful training, we need as many training samples as possible so there is a conflicting situation between the number of training inputs and the size of the test data that is used. Typically, a third of the training data is used as test data.

The second regularization tool is the weight decay factor in the network [16]. This allows the influence of old samples to affect the current weights much less than the current input. The old weight values decay over time. This allows the network to adapt to changes in the system that it is modelling. The problem with this is that the network could forget important information that it has learnt from the earlier samples.

The last tool is the use of noise injection [17]. The performance of the network can be improved for noisy and unseen data by training the network with a noise signal imposed on the training inputs or outputs. This makes the network more robust when it comes to predicting values for unseen data. The problem with this method is that we are artificially corrupting the data and therefore we have to ensure that the magnitude of the noise is not too great as to influence our network too much. And also it is not clear which probability distribution the noise must follow for a given application.
Neural networks can be applied to regression or classification problems. Regression involves fitting a curve to some sample data. Classification involves assigning labels to the sample data so that each point lies in a set of n classes [13][14][18][19].

Neural networks have been successfully applied to image classification [20]. The complexity of the classifier grows exponentially with the size of the object and with the number of dimensions [7]. It is therefore desirable to keep the number of input dimensions to the neural network as low as possible, reducing the curse of dimensionality [7]. Normally the image is down sampled or transformed before it is input to the neural network.

More information about neural networks is given in Appendix A.

2. Method

A framework for the shape classification system has been developed which follows the modular engineering approach for system design.

2.1. Overview

The method described is capable of working with multi-valued polar representations of the shape.
An overview of the shape classification system is shown in Figure 2.

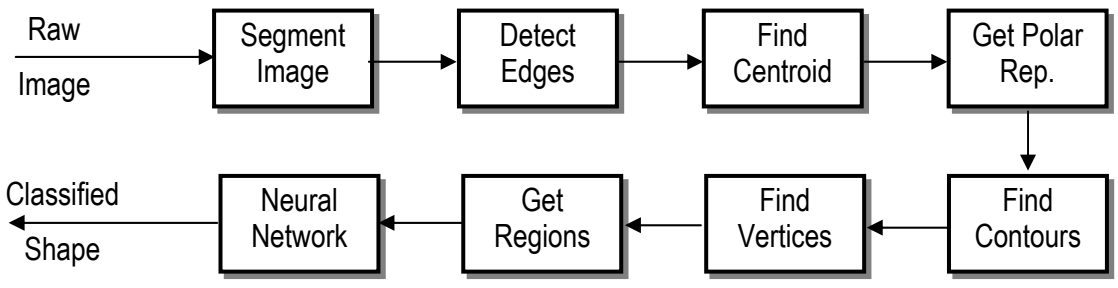


Figure 2 Overview of the shape classification system.

First the image is segmented to extract the object of interest. The boundary of the object is found by performing edge detection on the image mask. Next, the centroid of the shape is calculated and each point on the boundary of the object is used to get a polar representation of the shape. The polar representation is then converted into multiple contours so that important vertices can be found for the shape. This aids in the calculation of derivative information which is needed for finding interesting vertices. The polar representation is then sub-divided into a coarse grid and every region that contains boundary pixels or vertices is marked. The region grid is used as the input to the neural network where each segment of the grid is fed into a corresponding input of the network.

The framework is tested on two types of images. The first is a set of side-view images of automobiles taken from the first floor of a building. The system must classify the vehicles into three exclusive categories, namely: hatch-back, sedan or utility-vehicle. Examples of the three categories are shown in Figure 3.

Figure 3 Hatch-back, Sedan and Utility vehicle classes.

The second set of samples consists of hand images showing a different count of fingers. The system must output which fingers are being held up and this represents a non-exclusive set of outputs. The combinations of finger orientations are limited (for practical reasons) to the five shown in Figure 4.

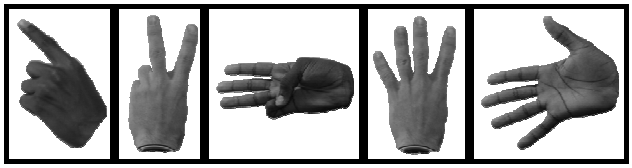


Figure 4 The finger orientations classified by the system.

This sample data represents a set of real world images that incorporate several problematic aspects of the polar representation method of shape classification.

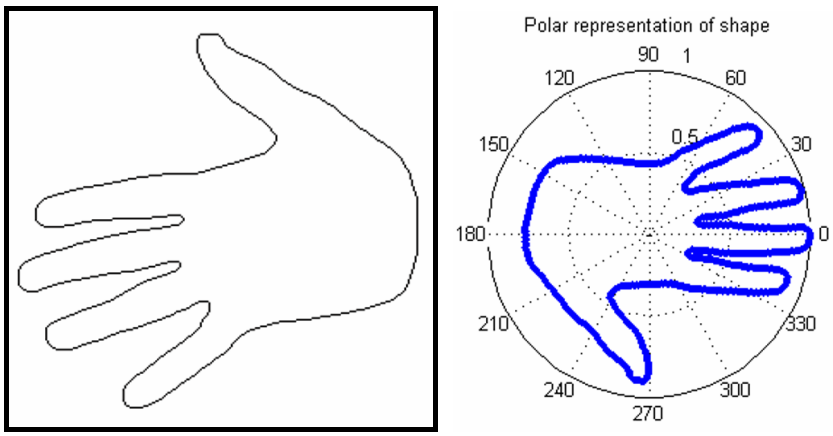
The method is limited to the representation and classification of the boundaries of 2D shapes or planar projections of 3D objects. Although this may seem restrictive, it is intended to be used in a more complex classification system where the shape information is just one parameter used to classify and track an object in an image.

2.2. *Segment Image*

The work assumes that the object has already been segmented from the background and that a mask represents the object to be classified. The actual segmentation process is beyond the scope of this paper but there are several techniques that can be used to do this [3]. The sample data in this paper was segmented by hand.

2.3. *Detect Edges*

In order to find the boundary of the object, one must perform edge detection on the mask image. The Canny method finds edges by looking for local maxima of the gradient of the intensity map [21]. The gradient is calculated using the derivative of a Gaussian filter. This method is robust and is more likely to detect true weak edges than other methods. A simpler (and therefore more computationally efficient) method for edge detection can be achieved by using morphological operators to erode the image mask. The pixels that are removed are the boundary pixels [3]. The result of the edge detection process is shown in Figure 5a.



(a) (b)

Figure 5 (a) Boundary information of the shape
(b) Polar Representation of the shape.

2.4. *Find Centroid*

The equation for calculating the centroid of an object [3][9] is given by:

$$(x,y)_{centre} = \left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n} \right) \qquad (1)$$

where n is the number of pixels on the boundary of the shape in the image; x_i and y_i are the horizontal and vertical pixel positions of the i^{th} boundary pixel. Every pixel in the boundary extracted from the

previous step is used to calculate the centroid. An alternate method to get the same centroid position is to use all the mask pixels of the object but this generally takes longer because the number of mask pixels is more than the boundary pixels. The centroid is what provides position invariance in the method.

2.5. Get Polar Representation

The polar representation is well suited to real world objects because they tend to be round and organic. The polar representation is able to capture this information conveniently without distorting the shape information significantly. The method described in [3] and [9] is used to get the polar representation of the shape boundary. The distance and angle of every boundary pixel is calculated with respect to the centroid position. This is shown in the equation below:

$$(\theta, R)_i = \left(\tan^{-1}\left(\frac{\Delta y_i}{\Delta x_i}\right), \sqrt{\Delta x_i^{\,2} + \Delta y_i^{\,2}} \right) \qquad (2)$$

where Δx_i and Δy_i are the differences between the i^{th} boundary pixel position and the centroid.

Scale invariance is achieved by normalizing the distance information for the extracted shape [9]. The phase of the boundary point with maximum radial distance is then used to rotate all subsequent points. This creates the rotation invariance in the method [9]. The polar representation of the example is given in Figure 5b. The polar representation of the shape is stored as a list of angle and distance pairs. See Appendix B for alternate methods for achieving position, rotation and scale invariance.

2.6. Find Contours

The polar representation is converted into a number of contours in order that we can find vertices which are representative of the shape. The contours themselves can contain information about sub-shapes which make up the overall shape. The advantage of the contour extraction is that each sub-contour can be treated as an independent shape and classified separately or used to aid classification of particular objects.

To find the contours efficiently, several methods were tried. The most effective method turns out to be the most computationally efficient one too. First the polar representation points are sorted with an increasing angle parameter. Because the angular information is cyclical, the polar representation is duplicated so that we have two revolutions of the shape. This allows us to easily find the contour trend across the $2n\pi$ angular position thresholds, for n being any integer.

The algorithm starts by adding the first (left most) point into the first contour. It then goes to the next point in the list and calculates the distance between the new point and the first point. If it is within some threshold (0.1 was used for this paper) then the point is added to the first contour. If not then the new point is added to a new contour. The distance of the next vertex is then calculated to the last points of each of the existing contours and the process continues until we have made a complete sweep of all the boundary points.

There will potentially be many contours that capture the multiple values of the polar representation as shown in Figure 6. After the various contours have been extracted a final sweep of the contours should be made to se if the ends of individual contours can be joined to make longer contours. This is done by checking the distances between starting and ending points of each of the contours. Contours that are within the distance threshold are then combined to make longer contours. If the end of one contour is near to the beginning of another then the two contours are merely concatenated. If the end of one contour is near the end of another then the order of points in the second contour is first reversed before concatenating the two contours. This ensures that the combined contour captures the behaviour that is illustrated in Figure 6.

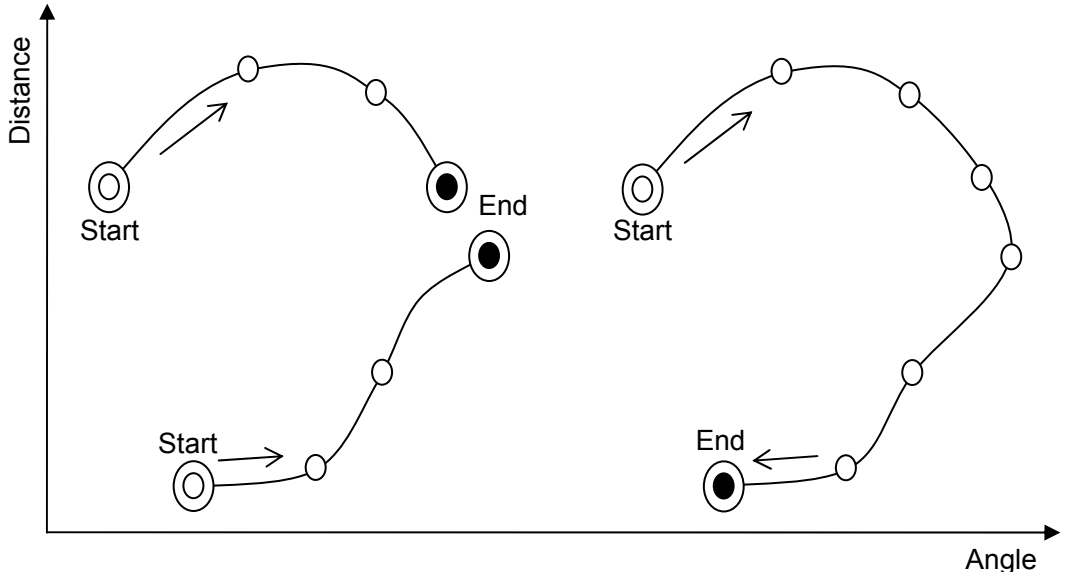


Figure 6 The ends of multiple contours should be combined if they are close to each other.

The method described above performs very well when compared to other more elaborate methods that explicitly look for the closest next boundary point. Those methods were tried but abandoned because they either take up too much memory or are too computationally expensive and time consuming to compute. Figure 7 shows the result of the contouring process on the polar representation.

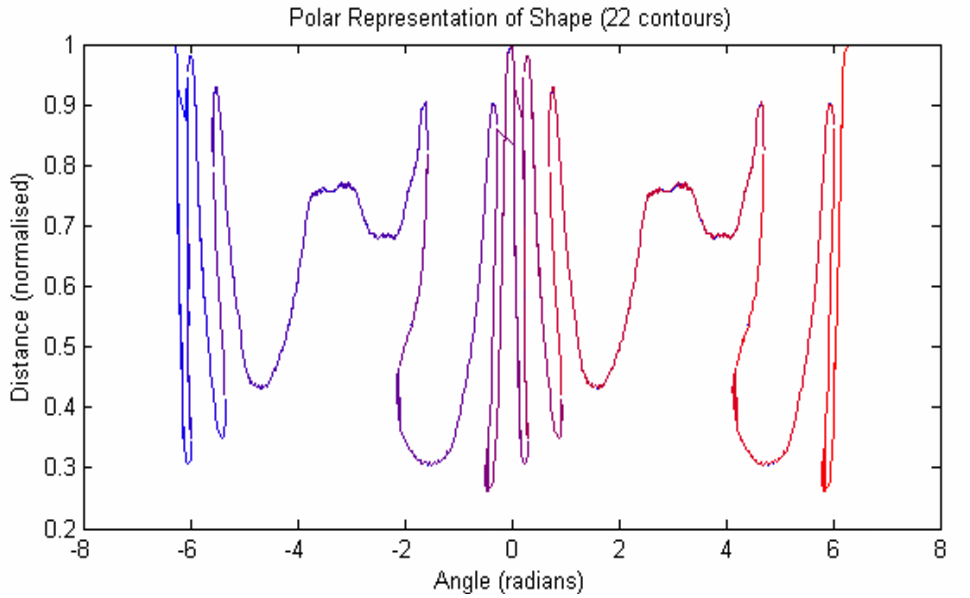


Figure 7 The polar representation after the contouring process.

The algorithm performs very well in regions where there is a low gradient (dR/dθ) such as 2 to 4 radians in Figure 7. This is because there is no ambiguity between which contours the points belong to as we move from left to right in the polar representation. When there is a high gradient, such as in the -1 to 1 radian region, there are typically incorrect contours being formed. By performing a similar sweep of the points but going from bottom to top (lowest to highest distances), we can build up a complimentary set of contours which would capture the behaviour of the contours more accurately in the regions of high gradient. They would however perform badly in the regions of low gradient. Further work must be done in combining the contours from both contour sweeps and only using the contours which perform well for the particular sweep direction.

2.7. Find Vertices

A method similar to [9] is used to find significant vertices of the shape. Zero crossings in the first derivative of the smoothed polar representation (See Eq. (3)) [9] are used to find these vertices.

$$\frac{d}{d\theta}R(\theta_p) = \frac{\sum_{i=1}^{n} R(\theta_{p-i}) - \sum_{i=1}^{n} R(\theta_{p+i})}{\sum_{i=1}^{n} (\theta_{p-i}) - \sum_{i=1}^{n} (\theta_{p+i})} \qquad (3)$$

where θ_p is the angle for point p; $R(\theta_p)$ is the distance of point p from the centroid; θ_{p-i} and θ_{p+i} are points before and after point p in the contour; n is the smoothing constant. The number of vertices extracted can be controlled my varying the smoothing constant. Higher values of n produce fewer vertices in general but this is not always the case. Each contour is padded with itself so that Eq. (3) is valid for the starting and ending points of the contour. The significant vertices for the example are shown in Figure 8.

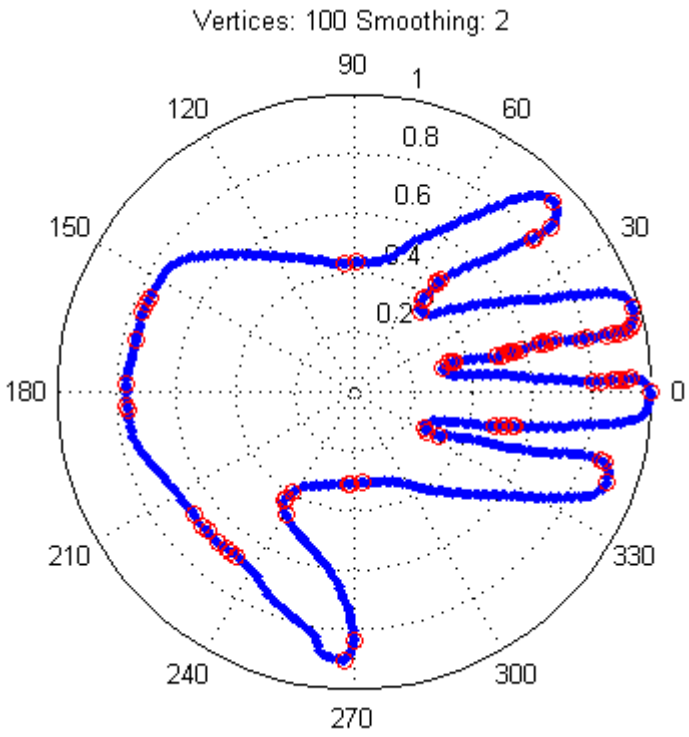


Figure 8 Significant vertices found using the zero crossings of the smoothed polar representation.

2.8. Get Regions

Once we have the polar representation of the shape with all the contours and vertices, we sub-divide the polar grid into a number of segments which represent spatial regions. The choice of region size should be tuned to the particular application but it was found that 20 divisions for the distance information and 108 divisions for the angular information (3.3° angular resolution) is more than sufficient for the two sample cases described in this paper. A matrix is used to represent the regions and if a boundary pixel lies in one of the regions then a value of 0.75 is written in the matrix at this position. If a vertex lies in a particular region then the region gets a value of 1.0. This allows us to give higher importance to the vertices of the shape.

The number of regions that are generated are relatively large when a fine resolution in the grid is desired. It is therefore important to reduce unnecessary regions in the grid. This is done by performing a logical OR operation on all the regions of the training data to get a mask for the sample data. The mask is morphologically dilated to remove any holes in the mask [2] and to allow for variation which we expect in test data. The overall region mask for the hand data is shown in Figure 9.

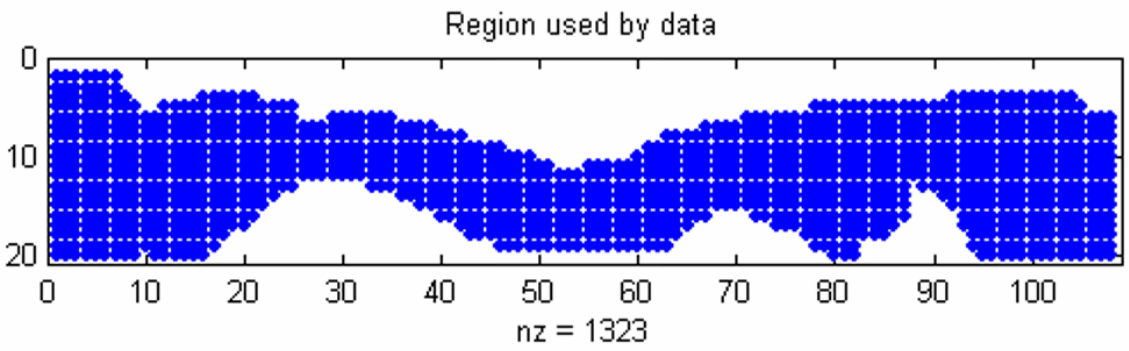


Figure 9 Overall region mask for all the hand training data after applying the dilation operator.

2.9. Neural Network

Neural networks are used to perform the shape classification as opposed to the traditional template (or distance) matching [7][22]. The polar representation regions that fall within the mask are used as the inputs to the neural network classifier. This ensures that the neural network learns the spatial relationship between the shape information and the shape classes.

An alternative method was tried where the normalized coordinates of the shape vertices are fed into the neural network but this scheme gave very poor results. This can be attributed to the difficulty of getting the same vertices into the corresponding inputs of the neural network for each shape. Another difficult way is to order the vertices in such a way that each input has the same meaning irrespective of what shape we are classifying. Principal Component Analysis (PCA) [16] was also used to reduce the dimensionality of the region-inputs. This gave poor results because the network is unable to distinguish between the slight changes in the reduced inputs for the various shapes. After performing the PCA, the actual values being fed into the network are no longer near-binary values but rather have a spread of values from 0.0 to 1.0. The network performs badly when trying to classify using the PCA data.

The major limitation with using neural networks to classify the shapes is that one has to know the number of shape classes that exist beforehand. This means that it is difficult to add an additional class once the network has already been trained.

The network is trained using the Scaled Conjugate Gradient (SCG) method because it converges quickly [16]. The generalisation ability of the network for unseen shapes is highly dependent on the values of the network weights at initialization. A subset of the training data is therefore used as (unseen) test data to gauge the generalisation ability of the learned network. The network is retrained several times until acceptable classification performance is achieved on the test data. This idea is similar to early stopping.

The network outputs are rounded off to the nearest integer when comparing the network outputs with the actual outputs. The actual values from the output of the neural network can be used as confidence limits on the classification of the shape. The closer the value is to either 0.0 or 1.0, the more confident the network is of its classification. Values approaching 0.5 mean that the network is uncertain of which class the shape is in.

3. Results

3.1. Automobiles

An MLP with 5 neurons, 894 inputs and 5 outputs was trained on the automobile data. A total of 26 training inputs were provided covering the various automobile classes. A total of 13 test shapes were used to measure the generalisation ability of the network.

3.2. Hands

An MLP with 5 neurons, 1323 inputs and 5 outputs was trained on the hand data. A total of 23 training inputs were provided covering the finger orientations. A total of 12 test shapes were used to measure the generalisation ability of the network.

3.3. General

A classification rate of 100% can be achieved for both data sets (automobiles and hands) if the network is retrained sufficiently. On average (without retraining), the network achieves about 90% classification depending on the random starting weights. The SCG requires fewer than 100 iterations to converge on a stable solution which gives zero error on the outputs after rounding off.

4. Analysis

A surprising result from the system performance is that the network achieves near 100% classification even though it is severely under trained. A general rule of thumb is to have at least twice as many training samples as the number of weights in the network. This is clearly not the case in the above training data and yet the network performs the classification very well on unseen data. This behaviour can be attributed to the number of inputs that are fed into the network and the near-binary values which are used. This representation of shape therefore makes the separation of the various classes for the given sample problem very clear. The converse was noticed when using PCA to reduce the number of inputs to the network. The values fed into these inputs were no longer near-binary values and the separation of the classes was significantly more difficult (only 30% classification was achieved, which approaches the performance of random guessing).

The above classification rates are misleading and it is expected that the network will have worse performance with noisy data. The disadvantage of the shape normalization method is that it is highly dependant on the boundary point with maximum radial distance from the centroid. Noise is able to confuse the entire shape representation if it alters the maximum radial distance point substantially. This is not a problem for small variations because the neural network is able to generalize sufficiently well for slight changes in the shape information. The discrete size of the region grid allows a significant amount of variation to actual boundary points without changing the values that are fed into the neural network. There is a trade-off between region segment size and the amount of detail that the neural network can distinguish. The results from the tests show very promising behaviour from the shape classification system. More work must be done to investigate how the system performs with larger data sets and noisier data.

5. Conclusions

In this paper, we have presented a shape classification framework which gives promising results for images of real world objects. The framework converts the boundary of the object into a polar representation which is suited to the representation of natural or organic objects. An efficient method is presented for finding and processing the multiple contours of the shape. These contours are used to extract interesting features from the shape and they are fed, along with the boundary information, into a neural network to be classified. The entire framework makes use of only simple arithmetic procedures and basic operations. This makes the framework relatively light-weight and suitable for embedded systems. Results from tests show impressive classification ability but more work must be done using larger datasets with more noise. Overall, the framework provides satisfactory results and should be considered for addition into a much larger object recognition system.

6. References

[1] J. Zhang, X. Zhang, H. Krim, and G. Walter, "Object representation and recognition in shape spaces", Pattern Recognition 36, pp 1143 – 1154, 2003.

[2] S. Antani, R. Kasturi, and R. Jain, "A survey on the use of pattern recognition methods for abstraction, indexing and retrieval of images and video", Pattern Recognition 35, 945–965, 2002.

[3] B. Jähne, "Digital Image Processing: Concepts, Algorithms and Scientific Applications, 4th Ed.", Springer, Berlin, 1997.

[4] S. Loncaric, "A survey of Shape Analysis Techniques", Pattern Recognition 31, 983-1001, 1998.

[5] T. Sebastian, P. Klein, and B. Kimia, "Recognition of Shapes by Editing Shock Graphs", ICCV, 755-762, 2001.

[6] D. Zhang, and G. Lu, "Review of Shape Representation and Description Techniques", Pattern Recognition 37, 1-19, 2004

[7] M. Egmont-Petersen, D. de Ridder, and H. Handels. "Image processing with neural networks—a review", Pattern Recognition 35, 2279–2301, 2002.

[8] M. James, "Pattern Recognition", BSP Professional Books, Oxford, 1987.

[9] T. Bernier, and J. Landry, "A new method for representing and matching shapes of natural objects", Pattern Recognition 36, 1711-1723, 2003.

[10] E. Davies, "Machine Vision: Theory, Algorithms, Practicalities, 2nd Ed.", Academic Press, 1997.

[11] R. Jain, R. Kasturi, and B. Schunck, "Machine Vision", McGraw-Hill, 1995.

[12] J. Martínez. "MPEG-7 Overview (version 9)" International Organisation for Standardisation; March 2003; http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm Last Accessed: 27/03/2004.

[13] M. Vidyasagar, "A Theory of Learning and Generalization", Springer-Verlag, London, 1997.

[14] M. Jordan, and C. M. Bishop, "Neural Networks", MIT Artificial Intelligence Laboratory, 1996.

[15] R. Duin, and D. de Ridder, "Neural network experiences between perceptrons and support vectors", Papers from the BMVC97 Proceedings, 1997. Last Accessed 20/05/2004. http://www.bmva.ac.uk/bmvc/ 1997/papers/duin/duin.html

[16] I. Nabney, "NETLAB: Algorithms for Pattern Recognition", Springer, London, 2002.

[17] M. Skurichina, S. Raudys, and R. P. W. Duin, "K-Nearest Neighbors Directed Noise Injection in Multilayer Perceptron Training", IEEE Transactions on Neural Networks, Vol. 11, No. 2, March 2000.

[18] R. Hecht-Nielson, "Neurocomputing", Addison-Wesley Publishing Company, USA, 1990.

[19] G. Luger, and W. Stubblefield, "Artificial Intelligence and the design of Expert Systems", The Benjamin/Cummings Publishing Company; California, 1989.

[20] H. Rowley, S. Baluja, and T. Kanade, "Neural Network-Based Face Detection", IEEE, PAMI, 1998.

[21] J. Canny, "A Computational Approach to Edge Detection" IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-8, No. 6, pp. 679-698, 1986.

[22] N. Rishikesh, and Y. Venkatesh, "Shape Recognition using an invariant pulse code and a hierarchical, competitive neural network", Pattern Recognition 34, 841-853, 2001.

Chapter 3: An Object Oriented Calculation Process Framework

Lukasz A. Machowski

Abstract *– Modern software applications are relying on increasingly complex calculation processes to perform tasks that are required. An object oriented calculation process framework is presented which brings calculations into the object oriented world. It does this by making each calculation process into a separate object with specific input and output variables. The benefit of this framework over existing ones is that the data types for the input or output variables do not have to descend from a framework specific data class. The framework still provides data type error checking by using reflection which is an advanced feature of .NET. The framework also discovers all sub-processes within the user defined process class using the same mechanism of reflection. The framework allows the process developer to concentrate on writing clean calculation code instead of wasting time writing error handling and process modelling code.*

1. Introduction

The complexity of modern software projects is perpetually increasing because of the growth in computational power available. Modern software engineering techniques allow us to comprehend these large software systems [1] and to manage the complex requirements of a user's application [2]. There is a tendency for users to demand more intelligence from their software and this typically requires the complexity of the calculation process to be increased. Up to now, software engineering techniques have been assisting us in designing, reusing and maintaining the large software systems. What is lacking is the focus on developing methods for writing complex calculation processes which are necessary for increasing the overall intelligence of the system.

This paper presents a framework which allows engineers and scientists to define complex calculation processes which are reusable and maintainable by making them into calculation objects. This aids in the effective design and sharing of calculation processes between teams of experts because of the type checking and other error-eliminating features of modern compilers. The class definitions required to make the calculation objects also ensure that they have precisely defined interfaces and their behaviour is predictable.

The remainder of this paper introduces the background of modelling and simulation, followed by the object oriented approach to process modelling. The design of the process calculation framework is

presented along with the decisions that were important for its development. A brief discussion of the framework is given along with recommendations for further work. Final comments about the framework are made in the conclusion.

2. Background

2.1. Modelling and Simulation

Engineers and Scientists rely on system modelling to understand the characteristics of physical objects and predict their responses in the real world [3][4]. Models allow the output of a system to be predicted without having to construct the physical system. Simulation allows scenarios to be tested that would otherwise be difficult to generate.

Engineers are familiar with solving complex problems by breaking them down into smaller ones and thus applying the principle of "Divide and Conquer" [5][6]. The majority of complex problems can be broken down into the interconnection of simpler modules which together form a process [7]. The large problem is solved by passing the data through the various stages of the process until the solution emerges at the end. The fields of Process-and-Control Engineering are built upon this foundation.

Scientists take a mathematical approach to solving problems. They tend to model their systems as functions and differential equations. This allows them to apply their vast body of theory to the system which they are modelling and fully characterize the output that it is expected to produce.

2.2. Process Models

A process can be defined as a structured and measured set of events occurring over time to produce a specified output [8][9]. From the literature, it becomes evident that the process deals with *how* the output is derived and not *what* output is derived. This has several implications. The first is that the process is made up of smaller units of work that have clearly defined inputs, outputs and behaviours. The second is that the order in which each work unit is done is just as important as what work unit is done. The relationships that relate and constrain these sub-processes are most important to defining the process [8][9].

Reuse is a key issue for handling complexity [1]. Process reuse involves identification, selection, understanding and awareness of suitable model-fragments [9] or partial-models [1] to reuse as templates. Identifying process templates is usually a tedious and difficult task and often leads to redevelopment of the entire process or using the same template for most of the processes [9].

2.3. Software and Modelling Paradigms

Software development is a process which takes as much skill to perfect as it does patience to endure [2]. Part of the problem lies in the fact that the software developer has to solve real world problems with a limited set of instructions that the computer understands. The different types of problems have led to various software paradigms that assist in solving the problems. A software paradigm is a pattern of how to design and develop software. Originally, software developers were mathematicians and scientists which were most comfortable using functional or procedural paradigms. Over time, the software systems became far too complex for individuals to comprehend and this led to the development of the Object Oriented (OO) paradigm. OO models real-world objects by capturing their attributes and behaviours and introduces effective mechanisms for ensuring maintainability and reusability within large software systems and amongst large groups of people. The functional paradigm was used because scientists were familiar with complex mathematical functions. With OO it is more difficult to get this ease of use for complex calculation processes because they have to be modelled as methods of objects or static functions. The problem with object methods is that they tend to be counter-intuitive when modelling mathematical or scientific calculation processes. The problem with static methods is that there is no mechanism that verifies when and how the calculations get performed, leaving a substantial burden on the programmer and ultimately introducing many possible sources of error.

As with the software paradigms mentioned above, there already exist many modelling paradigms and commercial simulation packages [7]. They can roughly be characterized as follows:

- Graph-based or language-based.
- Multi-domain or single-domain.
- Declarative or procedural modelling.

A modelling paradigm that is particularly useful for software development is the port-based modelling paradigm because of the types of calculation processes that are typically performed in software. A port defines an intended interaction (in the form of a signal or energy) between a sub-process and its environment [7]. A process is made up by connecting the ports of smaller sub-processes together. Signals are used for the interactions between processes in a block diagram. The drawback with this is that the user is responsible for determining the causality of the model [7].

Paredis *et. al.* [7] developed the concept of reconfigurable models which accommodate simulations at different levels of detail without the need for remodelling the complete system, thus allowing the process models to evolve.

2.4. Object Oriented Process Modelling

The Object Oriented Modelling Paradigm was invented by Hilding Elmqvist as part of his PhD dissertation [5][10]. This paradigm shares many of the advantages of object oriented programming while allowing one to model a-causal systems.

Representing processes as classes enables the modelling of process structures with work breakdown (decomposition) and control-flow dependencies (associations) [9]. Associations express how classes can be related in building the global system [8].

The Unified Modelling Language (UML) has officially become the standard for object oriented analysis and design [9]. Often, classes and associations are stereotyped as process model primitives. The lack of first class process modelling primitives is a major limitation of UML, particularly with associations, which have to be implemented by references [8]. The Object Management Group (OMG), who are responsible for defining and maintaining UML, are focusing on model driven architectures and are showing an interest in model-driven process support [9].

Object Oriented process models decrease the conceptual gap between process modelling and software implementation [9] while demonstrating that OO can be used successfully for modelling software processes [8].

2.5. Existing Languages:

Most of the general-purpose simulation software on the market such as ACSL, SIMULINK and SystemBuild assume that a system can be decomposed into block diagram structures with causal interactions [3].

There have been several attempts to define object oriented languages for physical modelling [1]. Jaccheri et. al. [8] focus on associations in their E^3 process modelling language [9]. In E^3 associations are first class primitives that can be defined and specialized just like classes [9]. Using inheritance, the modeller is able to create classes and associations at a high level of abstraction without preventing a later refinement by specialization [8]. Similarly, using aggregation, the modeller is able to hide the structure of the classes, and defer their detailed representation to the moment when the necessary information becomes available [8].

The port-based modelling paradigm builds on object oriented modelling languages such as VHDL-AMS and Modelica ([1][4]) [7].

2.6. Reflection

Reflection is an advanced feature of the .NET framework that gives runtime access to the metadata of a program so that it can gather information about its types. This includes the ability to get information about all the members (fields, properties, methods and custom attributes) of a class. Reflection also allows one to build new types at runtime. Reflection is typically used to create typed browsers; for finding fields to persist in serialization and for building symbol tables in compilers [11]. Because fields and properties are treated differently (properties being more powerful than fields), reflection is also useful for finding out which members of a class are fields and which are properties.

2.7. Current Problems with Process Modelling

The language for defining the process model is typically different to the rest of the application language. This means that the developer has to learn a completely new language with typically unrelated concepts. The process may also have to be implemented on a separate application system which would require interfacing to the user's application. This introduces compatibility problems and often involves licensing issues.

What is needed is a framework which allows the developer to define and implement the process in their native language of development. This reduces the time to learn how to use the process framework because the developer reuses their skills that they know from object oriented programming. What is also needed is a way of treating calculation processes as objects, which is not typically the case because calculations tend to be object methods or static functions.

The method presented in this paper aims to satisfy both requirements by allowing the definition, creation and execution of complex processes by using native C# and the advanced features of the .NET framework.

3. Method

This section describes the design and implementation of the process calculation framework developed for .NET. Firstly a brief overview is given of the entire framework and its usage. Then details of the most important classes are given to elaborate on the design. More UML diagrams are given in Appendix C.

3.1. Overview and Usage

The calculation process framework is based on an object oriented process modelling paradigm as described in the background section. Currently, the framework uses a simplified version of the port-based paradigm by using signals to connect sub-processes together to make a larger process. The current design does not use ports, which may model energy flow, but there is nothing stopping it from doing so conceptually. The consequence of this is that process developers are required to determine the causality of their models themselves. In a port-based paradigm this would not be the case. The reason for this design decision is that block-diagram-type processes are sufficient for a large number of calculation processes, including the ones that this framework has been designed for. A diagram showing the major concepts in the framework is shown in Figure 10.

The signals that flow between sub-processes are transmitted from output-variables and are received by input-variables. The different variable categories ensure that data flows in the correct direction between sub-processes.

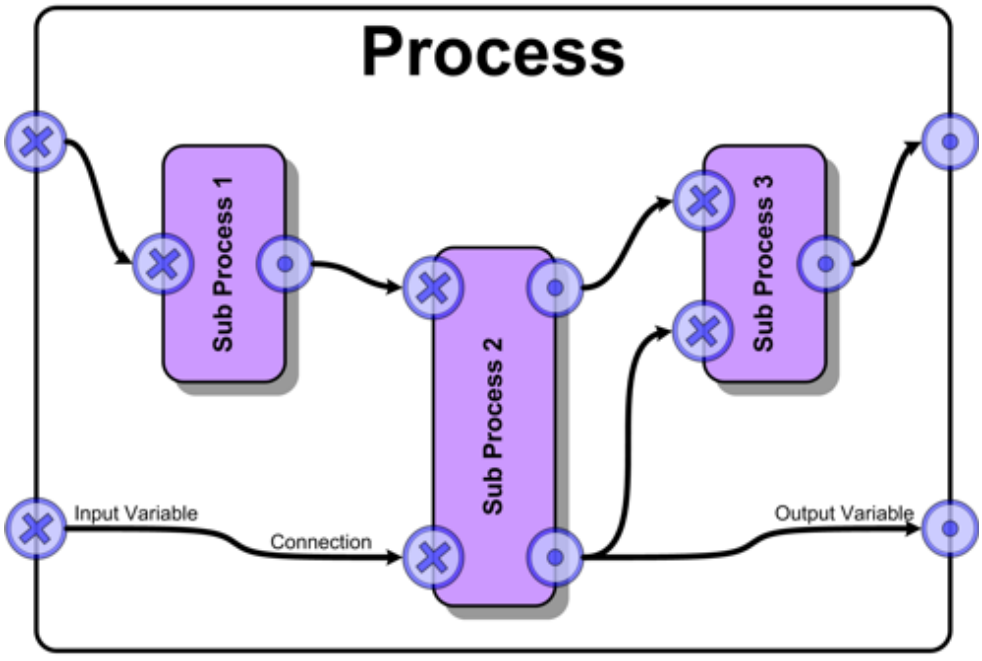


Figure 10 Calculation Process made up by connecting variables of sub-processes together.

A complete class diagram of the framework is given in Figure 11. The developer defines new processes by creating a descendant of the Process class. They then create attributes as they would normally do and define them as variables for the process by overriding *DefineVariables(...)* and using the *DefineVariable(...)* method. They then create any sub-processes by overriding *CreateSubProcesses(...)*; the framework automatically discovers all sub-processes that it contains. These sub-processes are then connected together by overriding *CreateConnections(...)* and using the *MakeConnection(...)* method. The developer then performs any pre-processing on the variables by overriding *PerformCalculationPreSub(...)* before the variables get passed to the sub-processes. After the sub-processes are complete, the developer can do post-processing by overriding *PerformCalculationPostSub(...)*. The actual calculation for the process gets performed in this method. An atomic process will have no sub-processes and the code for the calculation will be located in the *PerformCalculationPostSub(...)* method.

To use the process, the user creates the process object and assigns all required input values to the actual data attributes. They then flag that these process variables are ready by using the *Input["VariableName"]* collection to set the *IsReady* property to true. Once all the input variables are ready, the user may call the *Calculate(...)* method to actually perform the calculation process. They may query the success of this operation by checking the *CalculationSucceeded* property. If it is true then the output attributes of the process will contain the calculated results and the *Output["VariableName"].IsReady* property for each output variable will be true. The output variables resulting from the calculation process can be queried and used just as any other fields or properties would be used on an object.

More details about how specific aspects of the framework have been designed are given next.

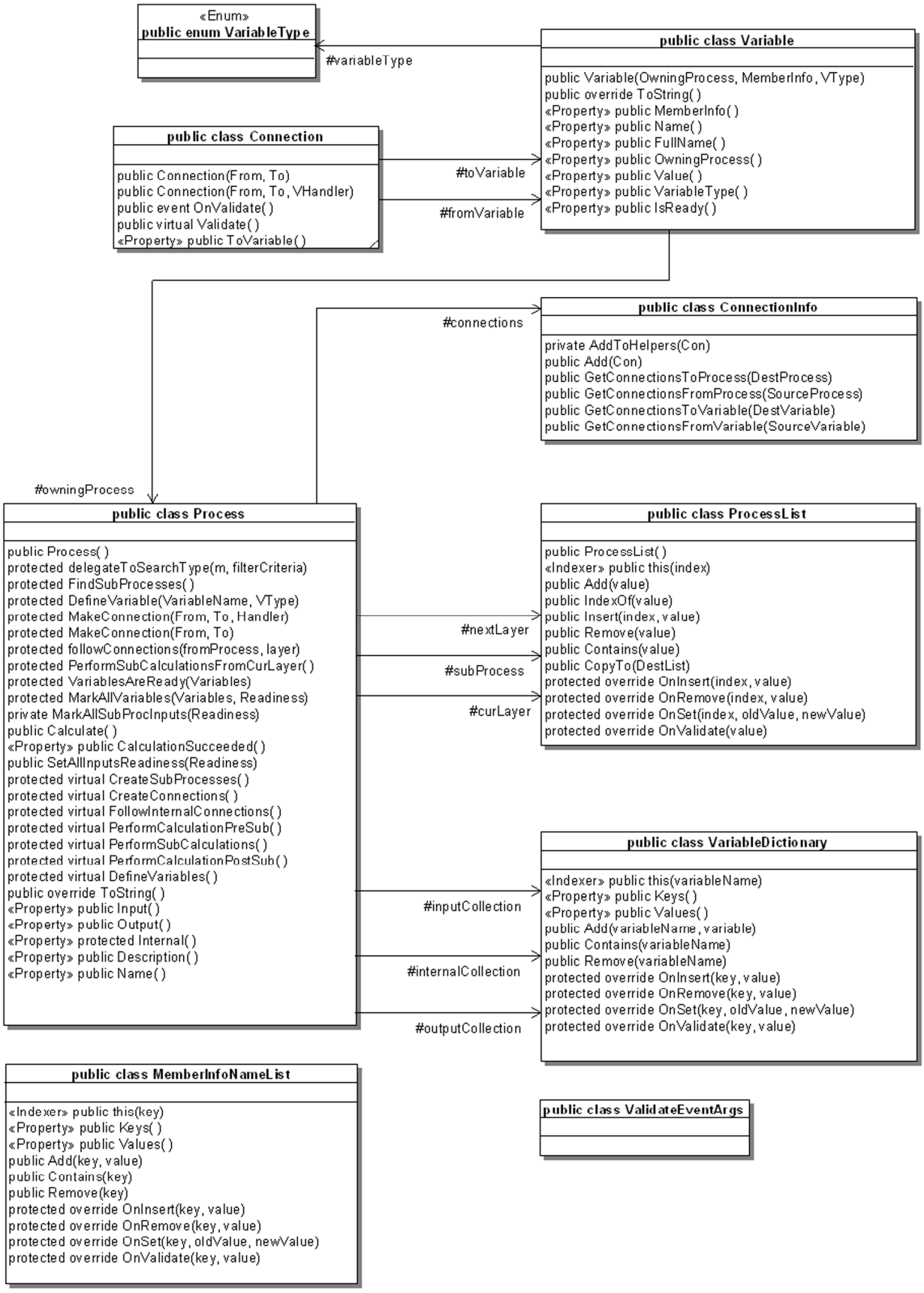


Figure 11 Complete class diagram of the Calculation Process Framework.

3.2. Variables

Typical objects in the OO paradigm contain their data in attributes (fields or properties). Variables are used to mark specific attributes as signal ports for the processes. The major advantage of this framework is that variables wrap arbitrary data types automatically as opposed to having the user manually wrap the data with a descendant of a framework-defined parent class. This means that any field or property of an object can be used as a variable in the framework without losing the ability to have compile-time and runtime type-checking performed on the variables. This flexibility is very attractive because with other frameworks, like E^3, the process developer has to descend their data class from the framework data class.

A variable is the abstraction that the framework makes for user defined fields and properties. It allows the framework to pass signals across sub-processes while still doing type-checking and validating the data. The variable class has a *Value* property that allows the user to access and modify the underlying attributes directly. The variable automatically returns the correct value of the underlying attribute (typed as an *object*, which is the parent of all classes in C#) no matter if it is a field or a property of the process class. Likewise with setting the value of a variable, the framework automatically checks what type of underlying attribute the variable represents and tries to set the value if the data types are compatible. If this is not the case then an exception is thrown.

A variable object is always linked to a specific instance of a process and therefore implements the idea of a port. In order to assist in finding the causality of a process calculation, each variable has a specific directionality assigned to it. Input variables receive data from outside the process. Output variables transmit data to other processes. Internal variables are generated within the process (possibly from calculations performed on the inputs) and may be used by sub-processes. The process developer is responsible for ensuring the causality of processes that depend on internal variables, while the framework automatically ensures the correct calculation order for processes that use Input and Output variables.

A variable object is instantiated by passing the *Owning Process*, the *Member Info* for the attribute and the *Directionality* of the variable (*Input*, *Internal* or *Output*). The *Owning Process* links the variable to a specific process object. The *Member Info* is a class defined in the *Reflection* namespace of the .NET framework and it allows the process framework to do type checking on the actual class attributes that are marked as variables.

When the process developer calls the *DefineVariable(...)* method, they pass the attribute name that they want to define as a process variable. The method then uses reflection to find any fields or properties in the process class with that name. If it is found, then the *Member Info* for that attribute is extracted and a

new variable object is created. This variable is then added to one of three internal lists of the process class, depending on the directionality of the variable. This makes retrieving *Inputs*, *Internals* and *Outputs* easy for the user because they can get to the variables by accessing collections with the same names.

Another important aspect of the variable class is the property that flags whether the variable's data is ready or not. The user sets the *IsReady* flag of Input variables that they assign data to. The framework automatically checks that all inputs are ready before the calculation process commences. This flag is used to determine if the output from the calculation is successful or not. A variable with the *IsReady* flag set to false should be treated as having invalid data stored in it.

3.3. Connections

Connections are used to create the associations between sub-processes. These allow one to add control-flow dependencies between these sub-processes. The signals flowing along the connections can be validated and the behaviour of the connections can be changed depending on the value of the data. This allows one to develop dynamic-processes which are very powerful and effective if designed carefully.

A connection is simply an object that allows signalling of information from one variable to another. When instantiating a connection object, one merely passes the source and destination variables to the constructor. The framework then checks to see that destination variable is assignment compatible with the source variable. Incorrectly using these connection objects is a common source of errors in most process frameworks. Therefore this functionality is abstracted away from the process developer by making them use the *MakeConnection(…)* method to define connections. The process developer merely informs the framework of which variables to connect by overriding the *CreateConnections(…)* method. The framework manages and creates the correct connection objects and adds them into the process so that the sub-processes can be calculated in sequence. The user may assign a validation handler to the connection which allows them to verify the data being passed between sub-processes. If the data is not valid (out of range or erroneous) then the data does not get propagated and the destination variable's *IsReady* flag gets set to false.

The connection objects do not provide a great deal of functionality themselves and are not really intended to be accessed by the user. Instead, most of the useful functionality comes from a helper class called *ConnectionInfo*. Each process contains an object of type *ConnectionInfo* which stores all the structural information for connecting sub-processes. This *ConnectionInfo* class allows one to easily get all the connections leaving or going into a particular sub-process or a particular variable. This is used by the framework to retrieve and perform the sub-process calculations in the correct order.

3.4. Processes

The *Process* class is the parent of all user defined processes. A process object provides the ability to convert input variables into output variables. The process class contains all the functionality to define sub-processes and to actually perform the calculations for the process. This includes the error-checking on the variables and inter-process communication. This frees the process developer from having to worry about many of the nuances of performing complex calculation processes and allows them to concentrate on writing clean calculation code. The biggest paradigm shift in the way of thinking is that the calculation developer assumes that all the input variables are valid when they write the calculation code. This makes the code significantly cleaner and easier to follow because it is not littered with confusing error checking code. All of the error checking is done elsewhere in the framework and it provides space for the process developer to write clean error-checking code. This decoupling of error-checking code and calculation code is one of the most valuable aspects of this framework.

The framework automatically detects and finds all sub-process objects contained in a process by using reflection to add them into an internal process list. This eliminates the need for the process developer to register the sub-processes with the framework and therefore makes defining new processes much simpler. In Section V, a discussion on how a similar result can be achieved for registering variables is conducted.

The order in which sub-processes are calculated is determined at runtime so that the framework can support the concept of dynamic processes. To get the first layer of sub-processes to calculate, the framework follows all connections leaving the input and internal variables of the parent process. It then attempts to perform the calculations on these processes. If a sub-process is successfully calculated then it follows the connections leaving its outputs and marks the connected sub-processes for calculation in the next layer. It may happen that sub-processes keep failing because of variable dependencies or other reasons but the overall calculation only fails if all sub-processes in one layer fail at the same time. This means that there is a deadlock situation in the process definition and typically means that the causality of sub-processes was badly thought out or there is an error in the sub-process calculation.

4. Analysis

Our framework improves on systems like E^3 because it does not require that the data classes be derived from a specified parent class [8]. Any data types can be used as variables and reflection allows the type checking to be performed by the framework.

Error logging is crucial to the successful development of processes. This is because the process developer is seldom able to implement error-free calculation code on the first attempt. Experience from using the framework has shown that logical errors in the calculation code can easily be found and corrected if the framework reports which sub-process calculations failed, and which variables are not valid. This focuses the process developer's attention directly on the source of the errors and reduces debugging time.

The use of reflection is very important because it ensures a tight coupling between the variable names used for processes and the actual attribute names used in the process classes. The framework automatically detects if the developer uses variable names that do not exist as attributes in the actual class being developed and reports an error. The automatic detection of sub-processes by using reflection also means that the user does not have to worry about registering the sub-processes with the framework. These are good features because they bring the two domains of process development and process implementation closer together.

The major advantage of using a framework like this one is that it removes the calculation specific code from the inter-process communication code. This allows process developers to concentrate on writing calculation process code, while still giving them extensive error checking and reporting capabilities by reusing the framework classes.

5. Further Work

A very powerful feature of the .NET framework is the ability to provide descriptive metadata about code elements in the form of custom attributes [11]. This means that the user can mark fields and properties as variables by using a framework defined custom attribute (eg: [*Variable("Input")]*). This would allow the framework to automatically find all the variables for a process by using reflection to find all the members with the *VariableAttribute*. This would eliminate the need for the process developer to override *DefineVariables(...)* and register the variables manually. The framework would find all the variables automatically just like it does with the sub-processes.

Another improvement to the framework would be to have the concept of parameters [1][3]. These behave like inputs to sub-processes except that they stay constant during a calculation. The values can be changed between calls to the *Calculate(...)* method.

6. Conclusions

There is a big push in .NET to make everything an object. With this framework, we bring calculations into the object oriented paradigm by implementing an object oriented process modelling framework. Reflection, which is an advanced feature of .NET, allows a closer coupling between process development and process implementation. The framework defines a *Process* class which is sub-classed to create a new process. All of the functionality that is necessary for connecting up sub-processes and calculating them is captured within the *Process* class. This allows the process developer to concentrate on writing clean calculation code while separating the data validation code elsewhere. This framework is well suited to implementing complex calculation processes while incorporating the main advantages of object orientation which are maintainability and reusability. This framework makes it significantly easier for large groups of people to develop and maintain complex calculation processes and therefore does the same for process development as object orientation did for software development.

7. References

[1] H. Elmqvist and S. Mattsson, M. Otter. "Modelica: The new object oriented modeling language" Presented at the 12th European Simulation Multiconference; Manchester; UK; 1998.

[2] H. van Vliet. "*Software Engineering - Principles and Practice. 2nd ed*." John Wiley & Sons, New York, USA, 2000.

[3] H. Elmqvist and S. Mattsson. "Modelica — The Next Generation Modeling Language An International Design Effort" in *Proceedings of the 1st World Congress on System Simulation* (WCSS'97); Singapore; 1997.

[4] S. Mattsson and H. Elmqvist. "An Overview of the Modeling Language Modelica", in *Eurosim'98 Simulation Congress*, Helsinki, Finland, 1998.

[5] F. Cellier. "Object Oriented Modeling: Means for Dealing with Software Complexity"; in *Proceedings of 15th Benelux Systems and Control Conference*, Mierlo, March 1996.

[6] S. Aitken and J. Curtis. "A Process Ontology" in *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 02),* Springer Verlag; 2002;

[7] C. Paredis, A. Diaz-Calderon, R. Sinha and P. Khosla. "Composable Models for SimulationBased Design" *Engineering with Computers*, vol. 17, pp. 112-128.

[8] M. Jaccheri, G. Picco and P. Lago. "Eliciting Software Process Models with the E3 Language", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 4, 1998.

[9] H. Jørgensen. "Interactive Process Models"; PhD Thesis; Department of Computer and Information Science; Faculty of Information Technology; Mathematics and Electrical Engineering; Norwegian University of Science and Technology, Trondheim, Norway, 2004.

[10] H. Elmqvist. "A Structured Model Language for Large Continuous Systems", Ph.D. Dissertation, Report CODEN: LUTFD2/(TFRT-1015), Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden, 1978.

[11] MSDN Library. "Reflection Overview", *.NET Framework Developer's Guide*, Visual Studio .NET 2003, Microsoft, 2003.

Chapter 4: Using an Object Oriented Calculation Process Framework and Neural Networks for Classification of Image Shapes

Lukasz A. Machowski

Abstract - *Modern software systems are perpetually becoming more complex. Machine vision is a field that demonstrates this concept clearly. Shape classification, which is a part of machine vision, has both high and low level image processes in it. This paper presents a framework that brings the definition and implementation of complex calculation processes into the object oriented paradigm. Defining processes in this manner assists in concurrent development by groups of people. The paper describes how the shape classification process is implemented in the framework. Results of a C# implementation are compared to a procedural implementation in Matlab. The timing requirements for the various stages of the shape classification process are evaluated. Conclusions from this are that the object oriented framework is well suited to high level machine vision concepts while the procedural paradigm is well suited to low level tasks such as edge detection. Poor classification performance is reported for the C# implementation but it is attributed to the poor training ability of the neural network toolbox used. The C# implementation of the shape classifier demonstrates the maintainability and reusability of the framework. It has 26 out of 33 process classes using inheritance for specialization or composition for defining sub-processes. This paper shows that the framework is well suited to machine vision processes.*

1. Introduction

The complexity of modern software projects is perpetually increasing because of the growth in computational power available. Modern software engineering techniques allow us to comprehend these large software systems and to manage the complex requirements of a user's application [1]. There is a tendency for users to demand more intelligence from their software and this typically requires the complexity of the calculation process to be increased. Up to now, software engineering techniques have been assisting us in designing, reusing and maintaining the large software systems. What is lacking is the focus on developing methods for writing complex calculation processes which are necessary for increasing the overall intelligence of the system. An example where this is the case is in the field of machine vision. There are many different features of an image which can be used for classifying objects and these include colour, texture, motion, context and shape [2][3][4]. Shape information is used extensively for classification of objects in vision systems and is an important visual feature [5][6]. There are also several methods that can be used for performing the classification of the object and these include similarity distance measures, template matching, index calculation, hashing, maximum likelihood,

clustering and neural network classification [2][7]. There are many advantages of using neural networks and these include being able to train the system using example images and having superior generalization abilities when compared to other classification techniques.

This paper presents the design and implementation of a shape classification system in an object oriented (OO) calculation framework. It allows engineers and scientists to define complex calculations by making them into calculation objects. This aids in the effective design and sharing of calculation processes between teams of experts because of the type checking and other error-eliminating features of modern compilers. The class definitions required to make the calculation objects also ensure that they have precisely defined interfaces and their behaviour is predictable. This framework is then evaluated by implementing a shape classification process in an object oriented manner. The implementation is given in Appendix D.

2. Background

2.1. Process Frameworks

A process can be defined as a structured and measured set of events occurring over time to produce a specified output [8][9][10]. From the literature, it becomes evident that the process deals with *how* the output is derived and not *what* output is derived. This has several implications. The first is that the process is made up of smaller units of work that have clearly defined inputs, outputs and behaviours. The second is that the order in which each work unit is done is just as important as what work unit is done. The relationships that relate and constrain these sub-processes are most important to defining the process [8][9]. The role of a process is to convert given inputs into useful outputs [11].

Reuse is a key issue for handling complexity [1][10]. Process reuse involves identification, selection, understanding and awareness of suitable model-fragments [9] or partial-models [1] to reuse as templates. Identifying process templates is usually a tedious and difficult task and often leads to redevelopment of the entire process or using the same template for most of the processes [9].

2.2. Object Oriented Modelling and Simulation

There already exist many modelling paradigms and commercial simulation packages [12][13]. A modelling paradigm that is particularly useful for software development is the port-based modelling paradigm because of the types of calculation processes that are typically performed in software. A port defines an intended interaction (in the form of a signal or energy) between a sub-process and its environment [12]. A process is made up by connecting the ports of smaller sub-processes together.

Signals are used for the interactions between processes in a block diagram. The drawback with this is that the user is responsible for determining the causality of the model [11][12].
The Object Oriented Modelling Paradigm was invented by Hilding Elmqvist as part of his PhD dissertation [14][15]. This paradigm shares many of the advantages of object oriented programming while allowing one to model a-causal systems. Representing processes as classes enables the modelling of process structures with work breakdown (decomposition) and control-flow dependencies (associations) [9]. Associations express how classes can be related in building the global system [8]. The OO modelling and simulation concept has great intuitive appeal because the notion of interacting objects is similar to that of real-world experiences [10].

The Unified Modelling Language (UML) has officially become the standard for object oriented analysis and design [9][16]. Often, classes and associations are stereotyped as process model primitives. The lack of first class process modelling primitives is a major limitation of UML, particularly with associations, which have to be implemented by references [8]. The Object Management Group (OMG), who are responsible for defining and maintaining UML, are focusing on model driven architectures and are showing an interest in model-driven process support [9].

Object oriented process models decrease the conceptual gap between process modelling and software implementation [9] while demonstrating that OO can be used successfully for modelling software processes [8].

2.3. Reflection

Reflection is an advanced feature of the .NET framework that gives runtime access to the metadata of a program so that it can gather information about its types. This includes the ability to get information about all the members (fields, properties, methods and custom attributes) of a class. Reflection also allows one to build new types at runtime. Reflection is typically used to create typed browsers; for finding fields to persist in serialization and for building symbol tables in compilers [17]. Because fields and properties are treated differently (properties being more powerful than fields), reflection is also useful for finding out which members of a class are fields and which are properties.

2.4. Machine Vision and Image Processing

Getting a machine vision system to recognize objects from images is a difficult and well studied problem [4]. The ability of humans to segment, classify and recognize objects with relative ease makes it a sought after ability for vision systems. Being able to sense the environment using visual information has many applications in the commercial, industrial, medical and military fields.

Machine Vision is used to understand the content of images while image processing enhances the ability to achieve machine vision [18]. Image processing can be partitioned into three levels and each level has a specific output [18]:

- Low-level: an improved image, such as one with reduced noise.
- Mid-level: a two dimensional data structure, such as the result of edge detection.
- High-level: a data structure describing the content of the scene.

Machine vision is the computer analysis of images with the intent to discover what information the images contain [18]. Recognition of features is the core of machine vision and it relies on finding suitable representations in order to find these features. Once an adequate representation has been achieved, pattern and feature analysis are employed to analyze the image for information content [18].
According to Erman and Lesser [19], it is impossible to construct a successful machine vision system without careful attention to issues of system engineering. These issues include maintainability and configuration control, human engineering, performance analysis, and efficiency. If we include reusability in this list then we notice that it resembles the problems faced by software developers before the advent of the object oriented software paradigm.

2.5. Shape Representation

Representation is the process of determining a description for an object or region [18]. There are several techniques for representing shape information [3][5][6][18][20][21][22] and many make use of the boundary of a segmented object to extract the shape information. The information can then be transformed to retrieve shape descriptors for classification of the shape or retrieval from a database [2][23]. It is desirable to have a representation that is easily computed and simple to interpret, especially if the system is to be implemented as an embedded system.

One method of representing shape is the Freeman Chain Coding which encodes the directions taken while moving from pixel to pixel around the shape boundary [18]. Problems with this method occur when the boundaries are not smooth or when they are fragmented [18]. The advantage is that chain coding is an efficient method for storing the shape information. Other methods include fitting line segments, B-Spline representation, Fourier descriptors and Autoregressive models [24].

If the pixels of a 2D image representing a shape are processed into a single dimension then we have what is known as a signature [18]. The polar representation is an example of a signature [20]. Acquiring the polar representation of a shape with respect to the centroid is an attractive approach because it is easily

calculated and only requires the boundary information [21]. The major problem with this is that the centroid position is affected by noise and occlusion in the boundary information. However, the method described in [20] is relatively insensitive to typical variations which can be found in real-world objects. The advantage of using this method is that the shape information is position, scale and rotation invariant [3][20].

Another problem identified in [21] with the polar representation approach is that the function may be multi-valued for a certain class of objects. This is especially true for most real world objects, such as the leaf shown in Figure 1. The solution presented in [21] is ad hoc and an improvement is presented in [25].

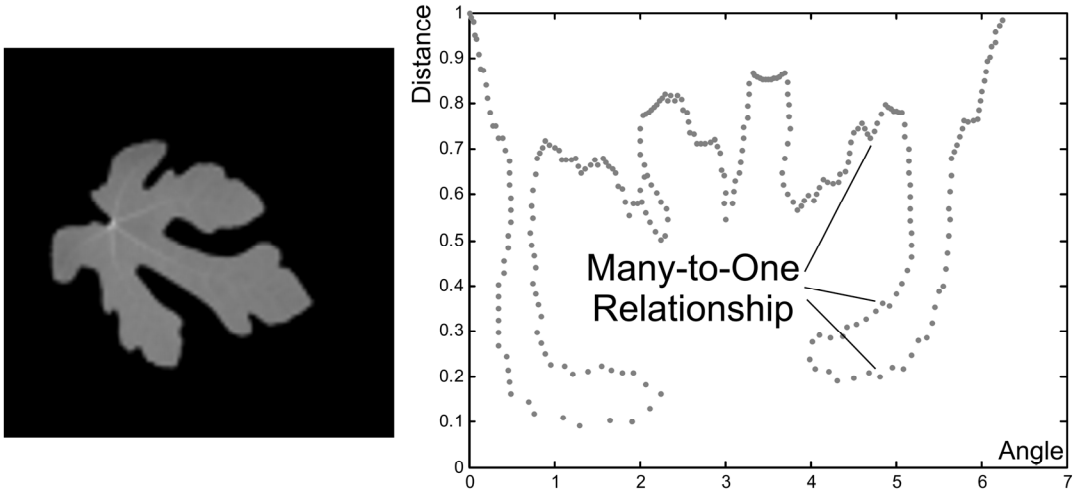


Figure 12 A leaf with concave sections and the problematic many-to-one polar representation.

2.6. Neural Networks

Neural Networks are interconnections of artificial 'neurons' that are greatly simplified versions of the biological neurons found in the human brain [26][27]. On a computer, the neural network is represented by a labelled acyclic directed graph with a clearly defined set of inputs and outputs [26][28]. The neural network may have any number of hidden units defined in an arbitrary amount of layers. There are several different architectures in existence, each one with its own advantages and disadvantages. The aim of the neural network is to be able to learn complex input-output relationships based on a source of training data, and then be able to make predictions for unseen data [26][28]. The neural network is used for its generalization abilities and there is always a trade-off between generalization ability and over-fitting [26][27]. It is possible to show that perfect generalization with a neural network is not possible due to the fact that an infinite amount of neurons can produce output that is consistent with the training data [26].

Neural networks can be applied to regression or classification problems. Regression involves fitting a curve to some sample data. Classification involves assigning labels to the sample data so that each point lies in a set of n classes [26][28][29][30].

Neural networks have been successfully applied to image classification [27][31]. The complexity of the classifier grows exponentially with the size of the object and with the number of dimensions [7]. It is therefore desirable to keep the number of input dimensions to the neural network as low as possible, reducing the curse of dimensionality [7]. Normally the image is down sampled or transformed before it is input to the neural network.

The neural network used for the classification stage in this paper is the Multi Layer Perceptron (MLP) [27]. This architecture is simple yet sufficient for classifying shapes of real-world objects [25].

3. Method

3.1. Calculation Process Framework

The calculation process framework is based on an object oriented process modelling paradigm as described in [13]. Currently, the framework uses a simplified version of the port-based paradigm by using signals to connect sub-processes together to make a larger process. The current design does not use ports, which may model energy flow, but there is nothing stopping it from doing so conceptually. The consequence of this is that process developers are required to determine the causality of their models themselves. In a port-based paradigm this would not be the case. The reason for this design decision is that block-diagram-type processes are sufficient for a large number of calculation processes, including the ones that this framework has been designed for. A diagram showing the major concepts in the framework is shown in Figure 13.

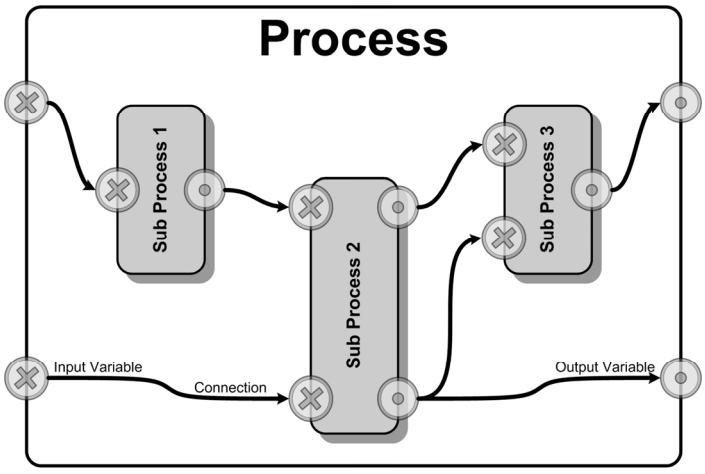


Figure 13 Calculation process made up by connecting variables of sub-processes together [13].

The signals that flow between sub-processes are transmitted from output-variables and are received by input-variables. The different variable categories ensure that data flows in the correct direction between sub-processes.

A complete class diagram of the framework is given in Figure 14. The developer defines new processes by creating a descendant of the *Process* class. They then create attributes as they would normally do and define them as variables for the process by overriding *DefineVariables(...)* and using the *DefineVariable(...)* method. They then create any sub-processes by overriding *CreateSubProcesses(...)*; the framework automatically discovers all sub-processes that it contains using reflection (which is an advanced feature of the .NET framework [13]). These sub-processes are then connected together by overriding *CreateConnections(...)* and using the *MakeConnection(...)* method. The developer then performs any pre-processing on the variables by overriding *PerformCalculationPreSub(...)* before the variables get passed to the sub-processes. After the sub-processes are complete, the developer can do post-processing by overriding *PerformCalculationPostSub(...)*. The actual calculation for the process gets performed in this method. An atomic process will have no sub-processes and the code for the calculation will be located in the *PerformCalculationPostSub(...)* method.

To use the process, the user creates the process object and assigns all required input values to the actual data attributes. They then flag that these process variables are ready by using the *Input["VariableName"]* collection to set the *IsReady* property to true. Once all the input variables are ready, the user may call the *Calculate(...)* method to actually perform the calculation process. They may query the success of this operation by checking the *CalculationSucceeded* property. If it is true then the output attributes of the process will contain the calculated results and the *Output["VariableName"].IsReady* property for each output variable will be true. The output variables resulting from the calculation process can be queried and used just as any other fields or properties would be used on an object.

Figure 14 Complete class diagram of the Calculation Process Framework [13].

3.2. Image Shape Classification

The method described in [25] is capable of classifying multi-valued polar representations of shapes. An overview of the shape classification system is shown in Figure 2.

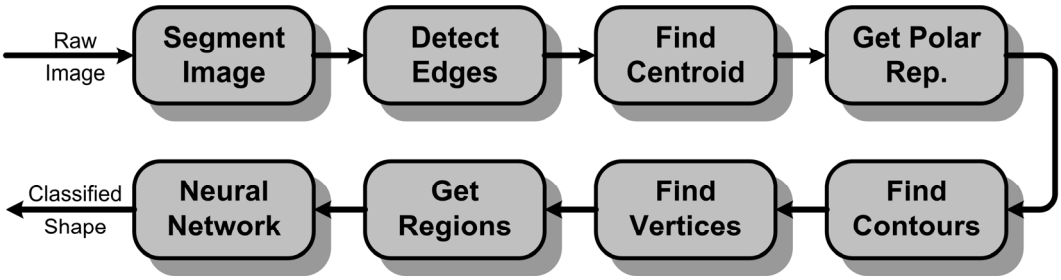


Figure 15 Overview of the shape classification system.

First the image is segmented to extract the object of interest. The boundary of the object is found by performing edge detection on the image mask. Next, the centroid of the object is calculated by using each pixel point of the object and this is used to get a polar representation of the shape. The polar representation is then converted into multiple contours so that important vertices can be found for the shape. This aids in the calculation of derivative information which is needed for finding interesting vertices. The polar representation is then sub-divided into a coarse grid and every region that contains boundary pixels or vertices is marked. The region grid is used as the input to the neural network where each segment of the grid is fed into a corresponding input of the network. A training set of example images can be used to create a region mask which reduces the number of inputs to the neural network. Examples of the various stages in the shape classification are shown in Figure 16. More details are found in [25].

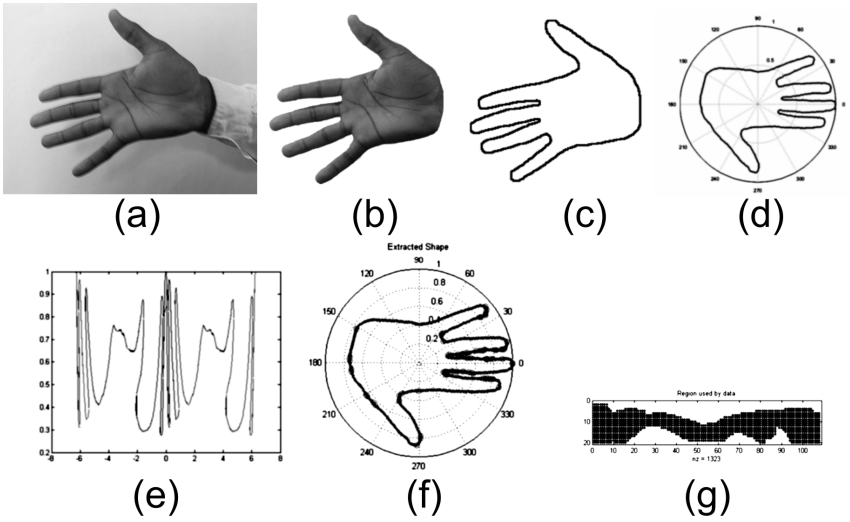


Figure 16 Shape classification stages [25]. (a) Raw Image. (b) Segmented Object. (c) Edges. (d) Polar Representation. (e) Contours. (f) Polar Representation with Vertices (g) Region Mask.

3.3. Object Oriented Approach

3.3.1. Overview

In order to create an object orientated implementation of the shape classification system described in Figure 16, one has to restructure the various stages of the process as in Figure 17.

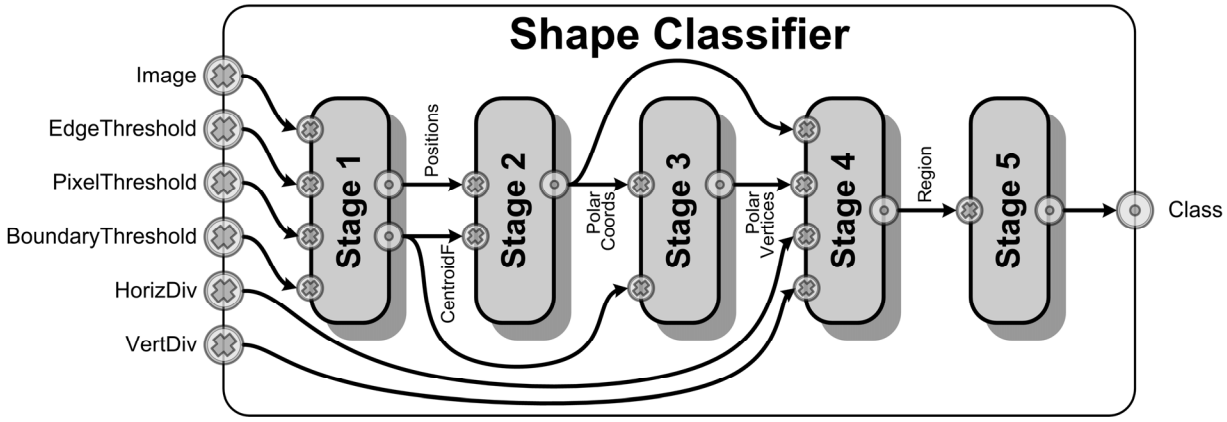


Figure 17 Various stages in OO shape classification.

Stage 1 is responsible for finding the positions of the boundary pixels and also the centroid of the shape. Stage 2 uses this information to find the polar representation of the shape. Interesting features in the form of vertices are then extracted from the polar representation in stage 3. The polar coordinates and vertices are then converted into a form that is understood by a neural network in stage 4. Stage 5 performs the actual shape classification. The class diagram of how these processes were implemented in the framework is given in Figure 18. More details about the various stages are given next.

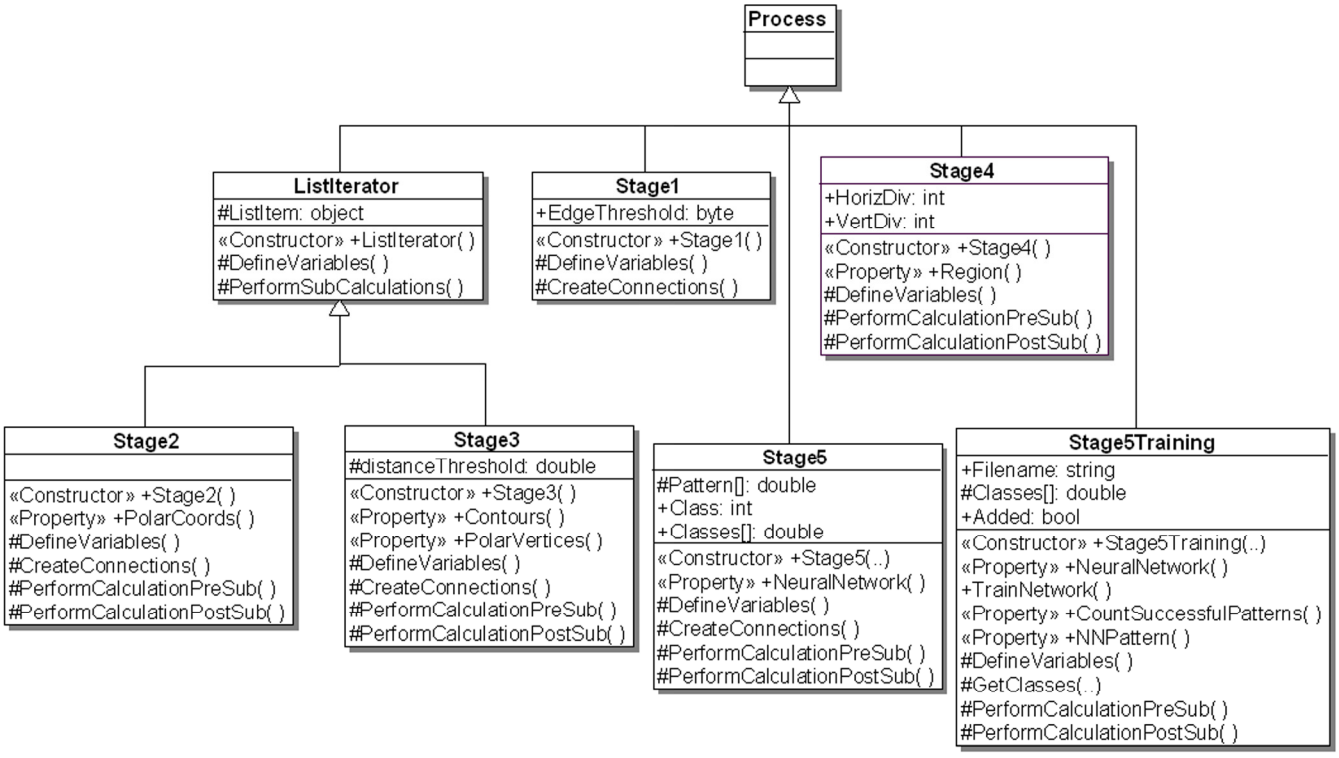


Figure 18 Class diagram of stage processes.

3.3.2. Stage 1: Boundary and Centroid Finder

Segmentation of the object is beyond the scope of this paper so it is assumed that stage 1 receives a segmented image as its input. This stage is actually made up of two processes running in parallel. The first process is a *BoundaryFinder* which performs edge detection (*EdgeDetector* sub-process) on the segmented image and then it returns the positions of all the boundary pixels (*BoundaryFromEdge* sub-process). This process takes two additional input parameters. The first is the *EdgeThreshold*, which filters out any pixels that are below a specified colour in the segmented image. This is useful for removing unwanted noise and other artefacts. The second parameter is the *BoundaryThreshold* which will not return the positions of edge pixels that are below that threshold. In order to implement the *BoundaryFromEdge* sub-process, we extend the calculation framework by creating a process called a *PixelIterator*.

3.3.3. PixelIterator Class

This process takes an image as an input and then it runs its sub-processes on each and every one of the pixels in the image. The *PixelIterator* defines two variables, *PixelPosition* and *PixelColor*. The process automatically connects up the variables to sub-processes that descend from the *PixelProcess* class, which define and register corresponding variables. In this way, we are able to easily define processes that operate on all the pixels of an image and have the framework worry about running the processes in the correct order. The *BoundaryFromEdge* class descends from the *PixelIterator* class and it has two sub-processes as is shown in Figure 19.

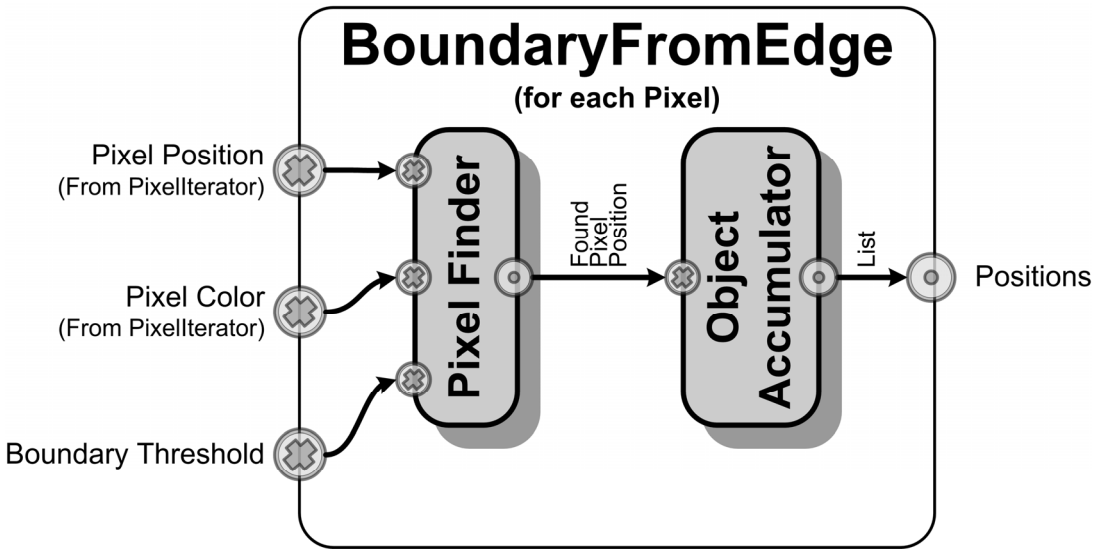


Figure 19 Sub-processes operating on all pixels in the *BoundaryFromEdge* class.

The *PixelFinder* process is a simple filter that compares the *PixelColor* to the *BoundaryThreshold*. If it exceeds the threshold then the *PixelPosition* is passed on to the *ObjectAccumulator* which maintains a list of all the inputs that it received. The output from the overall process is a list of pixel positions that

represent the boundary. The *BoundaryFromEdge* process illustrates how the framework allows dynamic processing because if a valid pixel is not found (in the filter) then the output for the sub-process is not valid and the framework will not execute the second sub-process.

The second parallel sub-process of stage 1 is the *CentroidFinder*. This class also descends from the *PixelIterator* class and it is implemented in a similar fashion to the *BoundaryFromEdge* process. It contains a pixel filter which feeds valid pixels into a sub-process that accumulates and calculates the centroid of the list of pixel positions.

3.3.4. Stage 2: Polar Representation Converter

This process takes the boundary pixel positions and the centroid found in stage 1 and creates a polar representation of the shape. Stage 2 is a descendant of the *ListIterator* class which is another extension to the process framework. It works exactly the same as the *PixelIterator* class except that it takes any arbitrary list (which implements the *IList* interface) as an input. Every single boundary pixel position is fed into a *PolarConverter* sub-process. This makes use of the centroid position and the algorithm described in [25] to find the polar representation. The output of this process is another list containing the boundary pixel locations on the polar grid.

3.3.5. Stage 3: Vertex Finder

In order to find vertices in the shape we first need to convert the shape representation into contours. The algorithm described in [25] is implemented in the *ContourMaker* process. Stage 3 is a descendant of the *ListIterator* class and it goes through each polar coordinate and feeds it into the *ContourMaker* sub-process. This sub-process slowly builds up a list of contour objects as it gets all the polar coordinates. A suitable object oriented design would dictate that the vertices should be properties of the contour objects. This means that the implementation for finding the vertices is done as a method of the *Contour* class. It should be noted that if calculating the vertices was a very complex process in itself, it would be appropriate to use process objects in the implementation of this method. At the end of stage 3, a list of all the polar vertices is extracted from the generated contours and this represents all the vertices in the shape.

3.3.6. Stage 4: Region Maker

All of the calculated information is useless unless it can be fed into the neural network in a form that it would understand. The role of stage 4 is to convert the shape and its features into *Regions* which allow a simple MLP to be able to classify the shape [25]. This stage merely creates a 2-dimensional polar grid with the given input dimensions (*HorizDiv* for the number of angular divisions, and *VertDiv* for the number of radial divisions). Stage 4 fills in values of 0.75 for segments of the polar grid that contain boundary pixels and it fills in values of 1.0 for segments that contain vertices. This stage also passes the

data through the region mask which eliminates polar-regions that are not expected to contain data for the range of shapes that are possible. This has the effect of reducing the number of inputs to the neural network.

3.3.7. Stage 5: Classifier

This stage performs the actual classification of the shape using an MLP. Each segment of the *Region* that is not filtered out by the region mask is used as an input to the neural network. The output classes represent particular shapes. We make use of *Neurobox* [32] for implementing the MLP and its built-in support for pattern recognition. The *Region* class contains all the functionality required to represent the data in a form that is compatible with the *Neurobox* network. As can be seen in Figure 18, there is a process called *Stage5Training*. This is a modification to the stage 5 process which assists in training and saving networks with example data. *Stage5* is designed to be more efficient for classification once a network has been trained.

3.3.8. Shape Classifier

The calculation framework is designed for reuse, and it makes defining new processes very quick and effective. The commonality between processes can easily be extracted and just segments that need to change can be slotted into an existing process. Once all 5 stages have been designed and implemented, all that remains is to connect up the various stages as is shown in Figure 17. The resulting process is what makes the *ShapeClassifier*. It demonstrates the ability of the framework to model complex processes at varying levels of abstraction because they are made up of a recursive set of sub-processes. The framework allows the actual implementation of the *ShapeClassifier* class to be both concise and easy to understand because it merely defines how the sub-processes are connected together. This eliminates many sources of error and confusion when debugging the code. It also allows various people to work on the design and implementation of the complex process independently, thus facilitating multi-developer collaboration.

4. Results

The shape classifier is implemented in C# as a collection of process classes. Using the classifier is as simple as using any other process of the framework. The sample data consists of a set of hand images showing different counts of fingers as in [25]. The performance of the implementation is evaluated in the next section.

4.1. Performance

4.1.1. Time Required

The time required to perform each stage of the shape classification is analyzed and compared between a C# (object oriented) implementation and a Matlab (functional) implementation. The amount of processing required depends on the image data being analyzed. In order to account for this, we evaluated the times for all the samples in the data set and present the average results in Figure 20.

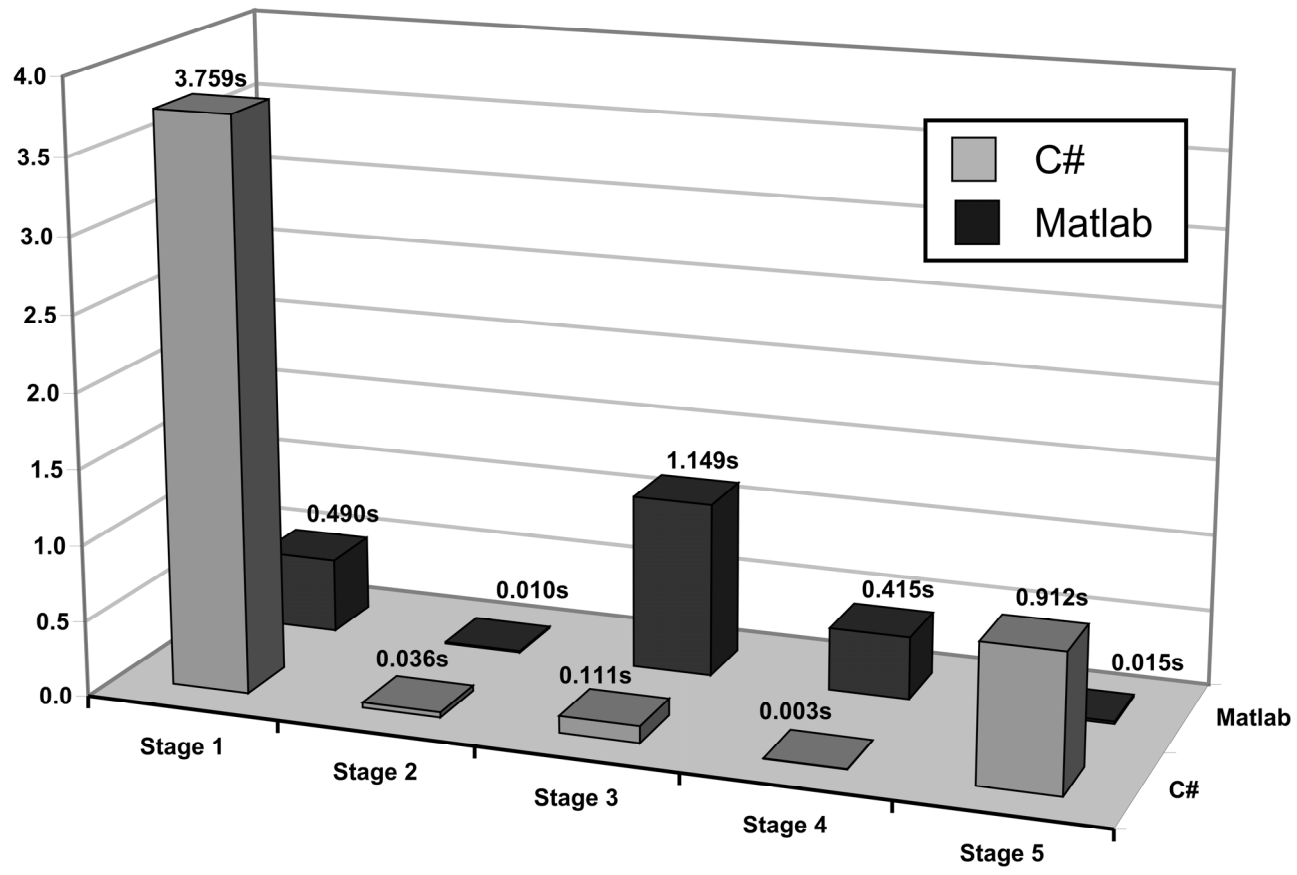


Figure 20 Normalized timing results for each stage of the shape classification.

The results clearly show that the stage 1 is implemented in a very inefficient manner in the object oriented process framework. This can be attributed to the fact that edge detection is a very low level image-process and is well suited to the functional paradigm. Stage 3, on the other hand is more efficient in the C# implementation because we are finding contours and vertices, which are easily modelled as objects, from the boundary information. This suggests that the process framework is efficient for the higher-level processes of machine vision. The poor performance of stage 5 (classification) can be attributed to the particular neural network toolbox used [32] and is discussed in the next section.

4.1.2. Classification Rate

The neural network toolbox used for classification in the C# implementation is designed in a completely object oriented manner [32]. This has the disadvantage of being slow when training the neural network on large data sets. The Neurobox toolbox makes use of simple back propagation and this takes a substantially long time when the training set exceeds about 20 image shapes. It is also very difficult to train the network so that it manages to classify all training shapes correctly. On the other hand, the Netlab [33] toolbox for Matlab is more advanced and manages to train quickly and accurately [25]. This can be attributed to the more advanced training algorithms (such as Scaled Conjugate Gradient methods) implemented in the toolbox. For this reason, it would not be a fair indication of the performance of the framework if we looked at the classification rates alone. In order to demonstrate that the framework does work, a small data set of 10 sample images was used to train the network. The classification rate is evaluated using the 'leave-out-one' method [27] which has the effect of eliminating variance due to different network weights and it also gives statistically relevant classification results. The network architecture had 1356 units in the input layer, 5 neurons in the hidden layer and 5 units in the output layer. A classification rate of 68% was achieved for the C# implementation, whereas the Matlab implementation (using the Netlab toolbox) achieved 90%. This proves that the overall shape classification process works and the timing performance shows that the object oriented process framework is feasible for implementing machine vision processes. The poor classification rate is attributed to the training mechanism of the Neurobox toolbox and it should not be used as a measure to judge the process framework. The outputs from stage 4 in both the C# and Matlab implementations are identical, which means that classification rates would be identical if the same neural network toolboxes were used.

5. Analysis

The calculation framework is a very effective means for implementing complicated processes. When implementing the shape classification system, a total of 33 classes were developed. Twenty six classes demonstrated code reuse by either using inheritance to specialize processes or composition to define new sub-processes from existing ones. The calculation framework is not meant to force the developer into creating software that resembles the procedural way of programming. Instead, they are encouraged to design the system in an object orientated manner and use the framework to convert complicated calculations (that would typically be implemented in long methods) into sets of inter-connected calculation objects. This capability also makes it possible for multiple software developers to work on complex calculation processes concurrently. This would have been more difficult to achieve in a functional programming paradigm because the smallest units of code that can be checked out in most concurrent source-code versioning systems are individual methods. With this framework, the large complex process can be broken down into a number of simpler classes while still giving the benefits of object orientation to the developers. The timing results described earlier demonstrate that framework is well suited to high level machine vision tasks, particularly when they model things that can be represented as objects. The edge detection process (which is very low level) should be reworked. One obvious improvement would be to parallelize the processes on each pixel so that the *PixelIterator* only goes through the image once. The reason why this was not done is because edge detection requires neighbouring pixels, and these cannot be accessed in the current implementation. The poor classification rate of the C# implementation suggests that a better neural network toolbox should be found. A port of the Netlab toolbox into C# may be one option.

6. Conclusions

This paper has demonstrated how a shape classification system, which has both low and high level machine vision processes, can be implemented in an object oriented manner. It also introduces a suitable framework which allows developers to define object oriented calculation processes. The framework allows existing processes to be specialized by using inheritance, and complex processes can be made up from simpler sub-processes by using composition. The framework promotes all the concepts behind object orientation for calculations that would typically have been implemented in object-methods or static-methods. The framework directly supports reusability and maintainability and it brings complex calculations into the object oriented world. This paper has shown that a practical shape classification system can easily be implemented in the proposed framework.

7. References

[1] H. Elmqvist and S. Mattsson, M. Otter. “Modelica: The new object oriented modeling language” Presented at the 12th European Simulation Multiconference, Manchester, UK, 1998.

[2] S. Antani, R. Kasturi, and R. Jain, “A survey on the use of pattern recognition methods for abstraction, indexing and retrieval of images and video”, Pattern Recognition 35, pp. 945–965, 2002.

[3] B. Jähne, “Digital Image Processing: Concepts, Algorithms and Scientific Applications”, 4th Ed., Springer, Berlin, 1997.

[4] J. Zhang, X. Zhang, H. Krim, G. Walter. “Object representation and recognition in shape spaces”, Pattern Recognition 36, pp. 1143 – 1154, 2003.

[5] S. Loncaric, “A survey of shape analysis techniques”, Pattern Recognition 31, 983-1001, 1998.

[6] D. Zhang, G. Lu. “Review of shape representation and description techniques”, Pattern Recognition 37, 1-19, 2004.

[7] M. Egmont-Petersen, D. de Ridder, H. Handels. “Image processing with neural networks—a review”, Pattern Recognition 35, 2279–2301, 2002.

[8] M. Jaccheri, G. Picco, P. Lago. “Eliciting software process models with the E3 language”, ACM Transactions on Software Engineering and Methodology, vol. 7, no. 4, 1998.

[9] H. Jørgensen. “Interactive process models”, PhD Thesis, Department of Computer and Information Science, Faculty of Information Technology, Mathematics and Electrical Engineering, Norwegian University of Science and Technology, Trondheim, Norway, 2004.

[10] J. Joines, S. Roberts. “Object-Oriented Simulation” in J. Banks. “Handbook of Simulation”, Wiley & Sons, New York, 1998.

[11] P. Gawthrop, L. Smith. “Metamodelling: Bond Graphs and Dynamic Systems”, Prentice Hall, London, 1996.

[12] C. Paredis, A. Diaz-Calderon, R. Sinha, P. Khosla. “Composable Models for Simulation Based Design”, Engineering with Computers, vol. 17, pp. 112-128, 2001.

[13] L. Machowski, T. Marwala. “An Object Oriented Calculation Process Framework”, Submitted for IEEE 3rd International Conference on Computational Cybernetics, Mauritius, 2005.

[14] F. Cellier. “Object Oriented Modeling: Means for Dealing with Software Complexity”, in Proceedings of 15th Benelux Systems and Control Conference, Mierlo, March 1996.

[15] H. Elmqvist. “A Structured Model Language for Large Continuous Systems”, Ph.D. Dissertation, Report CODEN: LUTFD2/(TFRT-1015), Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden, 1978.

[16] J. Holt. “UML for Systems Engineering”, Institution of Electrical Engineers, London, 2001.

[17] MSDN Library. “Reflection Overview”, .NET Framework Developer’s Guide, Visual Studio .NET 2003, Microsoft, 2003.

[18] H. Myler. “Fundementals of Machine Vision”, SPIE – The International Society for Optical Engineering, Washington, 1999.

[19] L. Erman, V. Lesser. “System Engineering Techniques for Artificial Intelligence Systems” in A. Hanson, E. Riseman. “Computer Vision Systems”, Academic Press, New York, 1978.

[20] T. Bernier, J. Landry. “A new method for representing and matching shapes of natural objects”, Pattern Recognition 36, 1711-1723, 2003.

[21] E. Davies. “Machine Vision: Theory, Algorithms, Practicalities”, 2nd Ed., Academic Press, 1997.

[22] R. Jain, R. Kasturi, B. Schunck. “Machine Vision”, McGraw-Hill, 1995.

[23] J. Martínez. “MPEG-7 Overview (version 9)”, International Organisation for Standardisation, March 2003, Last Accessed: 27/03/2004. http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm

[24] A. Jain. “Fundamentals of Digital Image Processing”, Prentice Hall, New Jersey, 1989.

[25] L. Machowski, T. Marwala. “Representing and Classifying 2D Shapes of Real-World Objects using Neural Networks”, Proceedings of the IEEE Conference on Systems, Man and Cybernetics, The Hague, Netherlands, pp. 6366-6372, 2004.

[26] M. Vidyasagar, “A Theory of Learning and Generalization”, Springer-Verlag, London, 1997.

[27] C. Bishop. “Neural Networks for Pattern Recognition”, Oxford University Press, Oxford, UK, 1995.

[28] M. Jordan, C. M. Bishop. “Neural Networks”, MIT Artificial Intelligence Laboratory, 1996.

[29] R. Hecht-Nielson. “Neurocomputing”, Addison-Wesley Publishing Company, USA, 1990.

[30] G. Luger, W. Stubblefield. “Artificial Intelligence and the design of Expert Systems”, The Benjamin/Cummings Publishing Company, California, 1989.

[31] H. Rowley, S. Baluja, T. Kanade. “Neural Network-Based Face Detection”, IEEE, PAMI, 1998.

[32] C. Ruegg. “NeuroBox Neural Network Library”, GPL Licence, Last Accessed: 18/11/2004. http://cdrnet.ch/projects/neuro/

[33] I. Nabney. “NETLAB: Algorithms for Pattern Recognition”, Springer, London, 2002.

Chapter 5: Conclusions

From the discussion in this report, it becomes evident that the shape representation method and calculation framework are well suited to the shape classification process. The results show that the OO paradigm for designing and implementing processes is well suited to the later stages of shape recognition. These results are expected to be valid for other stages of object recognition since they are all primarily based on abstract concepts that are not always easily represented using mathematics. The results also show that the OO paradigm is not well suited to low level image processing.

The research shows that the calculation process framework is an invaluable extension to how complex calculations can be implemented. It also demonstrates a very powerful use of the advanced features of the .NET framework by using reflection to narrow the divide between process definition and process implementation.

By implementing a simple shape classification system, this research has shown that a much larger object recognition system can easily be implemented by a number of people while still providing maintainability and reusability. The overhead of the OO framework is easily justified by the benefits of the conceptual abstractions that can be made while modelling the processes. Once a suitable calculation process has been defined, a new and optimised implementation can be generated which executes efficiently at runtime. It is therefore worth considering this framework for systems that have a significantly complex calculation process or for refactoring existing systems that have become too complex to understand.

Appendix A: Additional Work on Neural Networks

This appendix contains one paper that gives more information about neural networks and their uses.

Appendix B: Additional Work on Image Registration

This appendix contains two papers that give more information about image registration.

The second paper was accepted as a Work In Progress for the Fifteenth Annual Symposium of the Pattern Recognition Association of South Africa (PRASA 2004).

Appendix C: UML Diagrams

This appendix contains various UML diagrams detailing the design of the process framework.

Appendix D: Source Code and Applications

The source code and applications developed during the course of this Masters can be found on the accompanying CD-ROM.

A brief overview of what can be found on the CD is given below:

+---C#	The implementation of the system in C#.
| +---Processes	The calculation process framework.
| +---ShapeClassifier	The shape classification system and all supporting applications.
| | +---PatternDataMaker	Application used for making training sets.
| | +---ShapeClassifier	The actual shape classification application.
| | +---ShapeProcesses	The various process classes for the system using the framework.
| \---NeuroBox.src-2.5-LAM	The modified Neurobox Toolbox used for shape classification.
+---Images	The sample images used.
+---Matlab	The Matlab implementation of the shape classification system.
| +---Segmentation	Code for automatically segmenting the hand images.
| \---Shape	The shape classification system.
+---Thesis	The electronic version of this document.
+---Info	Papers and other resources used as information for this research.
\---DotNet	The Microsoft .NET framework runtime.