University of Pretoria etd – Peer, E S (2005)

A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence

by

Edwin S. Peer

Submitted in partial fulfilment of the requirements for the degree Magister Scientiae

in the Faculty of Engineering, Built Environment and Information Technology

University of Pretoria

Pretoria, South Africa

October 2004

University of Pretoria etd – Peer, E S (2005)

A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence

by

Edwin S. Peer

Abstract

A major flaw in the academic system, particularly pertaining to computer science, is that it rewards specialisation. The highly competitive quest for new scientific developments, or rather the quest for a better reputation and more funding, forces researchers to specialise in their own fields, leaving them little time to properly explore what others are doing, sometimes even within their own field of interest. Even the peer review process, which should provide the necessary balance, fails to achieve much diversity, since reviews are typically performed by persons who are again specialists in the particular field of the work. Further, software implementations are rarely reviewed, having as a consequence the publishing of untenable results. Unfortunately, these factors contribute to an environment which is not conducive to collaboration, a cornerstone of academia — building on the work of others.

This work takes a step back and examines the general landscape of computational intelligence from a broad perspective, drawing on multiple disciplines to formulate a collaborative software platform, which is flexible enough to support the needs of this diverse research community. Interestingly, this project did not set out with these goals in mind, rather it evolved, over time, from something more specialised into the general framework described in this dissertation. Design patterns are studied as a means to manage the complexity of the computational intelligence paradigm in a flexible software implementation. Further, this dissertation demonstrates that releasing research software under an open source license eliminates some of the deficiencies of the academic process, while preserving, and even improving, the ability to build a reputation and pursue funding.

Two software packages have been produced as products of this research: i) CILib, an open source library of computational intelligence algorithms; and ii) CiClops, which

University of Pretoria etd – Peer, E S (2005)

is a virtual laboratory for performing experiments that scale over multiple workstations. Together, these software packages are intended to improve the quality of research output and facilitate collaboration by sharing a repository of simulation data, statistical analysis tools and a single software implementation.

Keywords: Computational Intelligence, Design Patterns, Open Source, CILib, CiClops

Supervisor: Prof. A. P. Engelbrecht
Co-supervisor: Dr. F. van den Bergh

Submitted in partial fulfilment of the requirements for the degree Magister Scientiae.

University of Pretoria etd – Peer, E S (2005)

Acknowledgements

I would like to take this opportunity to thank the following:

- First and foremost, The Lord God Almighty for giving me the ability to do this work.
- My parents, Molly and Richard, for providing me with an ideal home environment.
- Hilary, my sister who fortuitously is an English studies graduate, for proof reading and checking up on my language usage. Not having a background in Computer Science must have made the work rather boring reading material, making her efforts all that much more appreciated.
- Andries, my supervisor, for providing me with guidance and putting up with all the changes in the direction that this work took to get completed.
- My good friend and co-supervisor Frans, who provided me with continuous motivation to get this work finished.
- Nathan, another good friend as well as my music teacher, for his patience during the music lessons in which I lacked the proper practice due to the time spent on research and writing for this dissertation.
- All those who have made contributions to CILib[1], particularly the early adopters of the software.
- SourceForge[2] for hosting the CILib project.
- The University of Pretoria, for providing me with an excellent undergraduate foundation to build upon and an environment with the necessary resources to conduct this research.
- The National Research Foundation, for providing financial support. The opinions expressed in this work and the conclusions arrived at are my own and should not necessarily be attributed to the National Research Foundation.

[1]http://cilib.sourceforge.net

[2]http://www.sourceforge.net

University of Pretoria etd – Peer, E S (2005)

Contents

1 Introduction **1**

1.1 Project History 2

1.2 Motivation 3

1.3 Scope 4

1.4 Contribution 4

1.5 Dissertation Layout 5

2 Computational Intelligence **6**

2.1 Problem Classes 7

2.1.1 Optimisation 8

2.1.2 NP-Complete Problems 10

2.1.3 Supervised Learning 13

2.1.4 Unsupervised Learning 14

2.2 Neural Networks 17

2.2.1 Feed Forward Neural Networks 17

2.2.2 Different Network Architectures 20

2.2.3 Learning Vector Quantiser 21

2.2.4 Self Organising Feature Maps 22

2.3 Evolutionary Computing 25

2.3.1 Genetic Algorithms 27

2.3.2 Genetic Programming 29

2.3.3 Evolutionary Programming 30

2.3.4 Evolutionary Strategies 31

2.3.5 Cultural Evolution 32

2.3.6 Coevolution 33

University of Pretoria etd – Peer, E S (2005)

2.4 Swarm Intelligence 34
2.4.1 Particle Swarm Optimisation 35
2.4.2 Ant Systems 37
2.5 Fuzzy Systems 42
2.5.1 Fuzzy Sets 42
2.5.2 Fuzzy Controllers 45
2.6 Other Paradigms 47
2.7 Hybrid Approaches 48
2.8 Software Requirements 50

3 Design Patterns **52**
3.1 Creational Patterns 54
3.1.1 Abstract Factory 54
3.1.2 Builder 55
3.1.3 Prototype 56
3.1.4 Singleton 57
3.2 Structural Patterns 58
3.2.1 Adapter 58
3.2.2 Composite 59
3.2.3 Decorator 60
3.2.4 Facade 61
3.2.5 Proxy 63
3.3 Behavioural Patterns 64
3.3.1 Interpreter 64
3.3.2 Iterator 65
3.3.3 Observer 67
3.3.4 Strategy 68
3.3.5 Template Method 69
3.3.6 Visitor 70
3.4 Discussion 71

4 Open Source Software (OSS) **73**
4.1 Licenses 74
4.1.1 Academic Free License (AFL) 75

4.1.2 Apache Software License (ASL) 76
4.1.3 Artistic License (AL) 77
4.1.4 BSD Licenses 77
4.1.5 Common Public License (CPL) 77
4.1.6 GNU General Public Licenses (GPL and LGPL) 78
4.1.7 MIT License 79
4.1.8 Mozilla Public License (MPL) 79
4.1.9 Open Software License (OSL) 80
4.2 The Open Source Ecosystem 80
4.3 Business Models 81
4.4 Open Source in a South African Context 83
4.5 University of Pretoria Intellectual Property 85
4.6 Credits 87

5 Languages and Tools 89
5.1 XML (eXtensible Mark-up Language) 89
5.1.1 Well Formed Documents 91
5.1.2 Document Types and Schemas 91
5.1.3 Document Object Model (DOM) 94
5.2 Java 94
5.3 Java 2 Enterprise Edition (J2EE) 99
5.3.1 Persistence Layer 100
5.3.2 Application Layer 101
5.3.3 Presentation Layer 102
5.3.4 Deployment 102
5.4 XDoclet 103
5.5 JUnit 104
5.6 Summary 106

6 CILib - Computational Intelligence Library 107
6.1 Coding Conventions 108
6.2 Implementation Details 110
6.2.1 Domains and Types 110
6.2.2 Problem Classes 116

University of Pretoria etd – Peer, E S (2005)

6.2.3 Algorithms . . . 119
6.2.4 Particle Swarm Optimisers . . . 123
6.2.5 Stopping Conditions . . . 129
6.2.6 Measurements . . . 131
6.2.7 Simulator . . . 133
6.3 Collaborations . . . 136
6.4 Limitations . . . 138

7 CiClops - Collaborative Laboratory 142
7.1 Architectural Overview . . . 143
7.2 Data Model . . . 145
7.3 Workers . . . 147
7.4 Client . . . 148
7.5 Status . . . 150

8 Conclusion 151
8.1 Summary . . . 151
8.2 Future work . . . 153

A List of Acronyms and Abbreviations 166

B Unified Modelling Language 170

C The Open Source Definition 172
C.1 Free Redistribution . . . 172
C.2 Source Code . . . 172
C.3 Derived Works . . . 173
C.4 Integrity of The Author's Source Code . . . 173
C.5 No Discrimination Against Persons or Groups . . . 173
C.6 No Discrimination Against Fields of Endeavor . . . 173
C.7 Distribution of License . . . 173
C.8 License Must Not Be Specific to a Product . . . 173
C.9 License Must Not Restrict Other Software . . . 174
C.10 License Must Be Technology-Neutral . . . 174

University of Pretoria etd – Peer, E S (2005)

D GPL Approval Letter **175**

E Popular Open Source Licenses **176**

E.1 Academic Free License (AFL) 176
E.2 Apache Software License (ASL) 180
E.3 Artistic License (AL) 184
E.4 BSD License 188
E.5 Common Public License (CPL) 189
E.6 GNU General Public License (GPL) 194
E.7 GNU Lesser General Public License (LGPL) 202
E.8 MIT License 212
E.9 Mozilla Public License (MPL) 213
E.10 Open Software License (OSL) 224

University of Pretoria etd – Peer, E S (2005)

List of Figures

2.1 Computational Intelligence Paradigms . . . 7
2.2 Example TSP Network (not to scale) . . . 11
2.3 TSP Optimal Tour (length = 28) . . . 12
2.4 Hierarchical Clustering . . . 16
2.5 Three Layer Feed Forward Neural Network . . . 18
2.6 Two Layer Learning Vector Quantiser . . . 22
2.7 5x5 Self Organising Feature Map . . . 24
2.8 Example U-matrix plot . . . 25
2.9 Crossover Operators . . . 28
2.10 Genetic Program Tree Representation . . . 29
2.11 Cultural Algorithm . . . 32
2.12 Typical Neighbourhood Topologies . . . 36
2.13 Membership Functions for *Age* Linguistic Variable . . . 44
2.14 Fuzzy Controller Architecture . . . 45

3.1 Abstract Factory . . . 54
3.2 Builder . . . 55
3.3 Prototype . . . 56
3.4 Singleton . . . 57
3.5 Adapter . . . 59
3.6 Composite . . . 60
3.7 Decorator . . . 61
3.8 Facade . . . 62
3.9 Proxy . . . 63
3.10 Interpreter . . . 65

University of Pretoria etd – Peer, E S (2005)

3.11 Iterator . . . 66
3.12 Observer . . . 67
3.13 Strategy . . . 68
3.14 Template Method . . . 69
3.15 Visitor . . . 70

5.1 A Simple XML Phone Book Document . . . 90
5.2 Phone Book Document Type Definition (phonebook.dtd) . . . 92
5.3 Phone Book Schema (phonebook.xsd) . . . 93
5.4 EJB Entity Relationship . . . 101
5.5 JUnit Composite Test Framework . . . 105

6.1 Partial Domain Grammar . . . 111
6.2 Domain Composite/Interpreter . . . 111
6.3 Domain Visitor Interface . . . 113
6.4 Partial Type System . . . 114
6.5 Domain Builder . . . 115
6.6 Problem Interfaces . . . 116
6.7 Solution Classes . . . 117
6.8 Optimisation Problems . . . 118
6.9 Fitness Classes . . . 119
6.10 Algorithm, Stopping Conditions and Events . . . 119
6.11 Optimisation Algorithms . . . 121
6.12 Overview of PSO Architecture . . . 123
6.13 Particle Decorators . . . 125
6.14 Velocity Updates . . . 126
6.15 Particle Visitors . . . 128
6.16 Stopping Conditions . . . 130
6.17 Measurements . . . 132
6.18 XML Object Factory . . . 134
6.19 Simple Simulator Configuration . . . 135
6.20 More Complex Simulator Configuration . . . 137

7.1 CiClops Overview . . . 144

University of Pretoria etd – Peer, E S (2005)

7.2 CiClops Data Model 146
7.3 Configuring a CILib simulation using CiClops 148
7.4 CiClops monitoring CILib simulations 149

B.1 Example UML Class 170
B.2 UML Relationships 171

University of Pretoria etd – Peer, E S (2005)

List of Tables

2.1 Fuzzy Set Theoretic Operators 43

4.1 Open Source License Characteristics 76
4.2 Instrumental Open Source Software 87

5.1 NastyPSO Performance 97

6.1 Legal Algorithms for Stopping Conditions 131
6.2 CILib Contributors 138

University of Pretoria etd – Peer, E S (2005)

List of Algorithms

1 Neural Network Back-propagation Training 19
2 Learning Vector Quantiser Training 23
3 General Evolutionary Computing Framework 26
4 Particle Swarm Optimiser 37
5 Ant Colony Optimiser for TSP 39
6 Ant Colony Clustering 41

University of Pretoria etd – Peer, E S (2005)

Chapter 1

Introduction

> *"PLAN, v.t. To bother about the best method of accomplishing an accidental result." — Ambrose Bierce, Devil's Dictionary*

Some of the most significant discoveries are those stumbled upon unintentionally. History is scattered with examples of such discoveries that have apparently come about by accident [97]. Archimedes determined a method of calculating the volume of irregular shaped objects, using displacement, when he noticed the water level rising while getting into a bath. Another example is Newton's inspiration for the theory of gravity resulting from the falling of an apple. The inspiration for and the discovery of many inventions, ranging from velcro to penicillin, was due to the sagaciousness of inventors to recognise the value of something unexpected during another, usually unrelated, activity. The phenomenon of making discoveries in this manner has become known as the "Serendipity Effect" [48].

WordNet defines serendipity as "accidental sagacity; the faculty of making fortunate discoveries of things you were not looking for". Although this work may not be as significant to mankind as the discovery of penicillin, it definitely turned out to be more important to the Computational Intelligence Research Group at the University of Pretoria (CIRG@UP)[1] than its original focus.

The following section takes the reader through the history of this research detailing how the project serendipitously grew into something more ambitious than initially intended. Next, the importance of this research is covered in Section 1.2. Since this work

[1] http://cirg.cs.up.ac.za

University of Pretoria etd – Peer, E S (2005)

is only the first step in a collaborative effort, a careful scoping of what is and is not covered by this dissertation are discussed in Sections 1.3. This introduction concludes with the contribution of this research in Section 1.4 and a breakdown of the dissertation layout in Section 1.5.

1.1 Project History

This research set out with the very specific goal of creating a taxonomy of existing Particle Swarm Optimisers (PSOs, refer to Section 2.4.1) and performing an empirical analysis of their performance on various optimisation problems. To accomplish this, several PSOs and benchmark problems were implemented in C++, dubbed PSOLib, with the intention of having a flexible object oriented design to facilitate experimentation by making every aspect of the platform as configurable as possible. The lack of reflection features in C++, however, made it very difficult to configure properties of objects and their compositions at run time, leading to the investigation of Java as an implementation platform.

Java turned out to be a viable platform for multiple reasons. Most importantly, its reflection API (Application Programming Interface) enabled classes to be dynamically instantiated and composed according to a run time configuration file. Further, the built-in XML (eXtensible Mark-up Language, refer to Section 5.1) processing API was convenient, since XML was chosen as the representation for configurations. Finally, Java's performance was found to compare favourably to C++ for implementing PSOs (refer to Section 5.2).

Work began on porting the existing PSOLib code to Java, while at the same time generalising the platform to support the needs of a wider audience, at which point it became known as CILib (Computational Intelligence Library, refer to Chapter 6) and the focus of this work shifted away from PSOs. Initially, CILib was made available to other members of the CIRG@UP and it was later decided to release the software under an open source license (refer to Chapter 4) in an attempt to promote collaboration with third parties outside of the research group. Later, it became evident that there are strong merits for such a collaborative research platform, which ultimately became the subject of this research.

University of Pretoria etd – Peer, E S (2005)

1.2 Motivation

The following problems, which were identified during a survey of several PSO papers [89], serve as motivation for effective collaborative research tools:

- **Duplication of effort:** In the restricted context of a research group, duplication of effort equates to lost productivity. In general, the science is better served if researchers can expend their efforts on developing new algorithms instead of writing implementations for software that already exists elsewhere. A collaborative code base can save researchers from reinventing the wheel. Further, an awareness of what is happening in industry can reduce the likelihood of duplicating work in academia which is already in active use by industry players.

- **Failure to take latest developments into account:** A collaborative code base increases awareness of what others are doing, in effect providing all participants with a more generalised view even though they specialise on their own specific work.

- **Insufficient testing on problems:** The No Free Lunch (NFL) theorem [120, 121] implies that algorithms should be tested on many problems to determine which problems they are best suited for, since all algorithms are on average equivalent when all possible problems are considered. Thus, large amounts of empirical data will need to be generated, which may have value if shared, to draw conclusions about the relative merit of different algorithms.

- **Poor parameter choices:** Good parameter choices for algorithms can be communicated as default values in a shared implementation platform. Further, a shared repository of simulation results can make researchers aware of the best results obtained for a given algorithm by other researchers.

- **Conflicting results:** Ignoring the fact that results cannot be in conflict if everyone shares the same implementation, a collaborative platform will undergo more stringent peer review and is likely to be far more reliable than throw away research code.

- **Invalid statistical inference:** Shared statistical analysis tools, which provide decision support for the best analysis method to use in a given context, can reduce

University of Pretoria etd – Peer, E S (2005)

the risk of researchers making incorrect assumptions about the applicability of statistical tests.

1.3 Scope

Building a collaborative framework to support the needs of a large research community requires a broad view of the subject matter in order to make it general enough to suit all parties.

For this reason, the computational intelligence field is examined in detail. Design patterns are examined as a means to manage the complexity of this broad field, ensuring a flexible software design capable of supporting the subject matter. Open source licensing is studied for the benefits it brings to a collaborative software development process. Further, this work draws on other software, tools and best practices from industry, which are unlikely to be found in scientific circles, but provide significant benefits for the software implementation.

The software implementation, however, is only discussed within the context of particle swarms, which was the original focus of this work, as a specific example demonstrating the more general framework. Further, an in depth knowledge of Object Oriented (OO) [21, 49] programming is assumed.

1.4 Contribution

The primary contribution of this work is two software components:

- **CILib** (Computational Intelligence Library), which is a shared collaborative framework for implementing computational intelligence software. Publishing it under an open source license maximises its visibility and its availability to potential collaborators.

- **CiClops** (Computational Intelligence Collaborative Laboratory Of Pantological Software), which is intended to further the collaborative goal by providing a scalable simulation environment, a shared repository of empirical data and statistical support tools.

University of Pretoria etd – Peer, E S (2005)

Finally, this dissertation shows how the above mentioned frameworks facilitate collaboration in computational intelligence, while serving the dual purpose of providing documentation, introducing the framework to potential collaborators.

1.5 Dissertation Layout

The remainder of this dissertation is organised as follows:

- **Chapter 2:** The computational intelligence field is examined, illustrating its complexity and highlighting requirements for a flexible software framework capable of handling this complexity.
- **Chapter 3:** Patterns are explored as a mechanism for implementing good software design by drawing on the experience of experts.
- **Chapter 4:** Open source licensing is investigated as a means to facilitate collaboration while exposing software developers to reputation rewards and profit opportunities.
- **Chapter 5:** The languages and tools which are prerequisites for working with the software developed for this research are discussed.
- **Chapter 6:** The implementation of CILib is discussed with particular reference to the platform's use of patterns.
- **Chapter 7:** CiClops is introduced as a mechanism to address some implementation specific limitations of CILib while improving its viability as a collaborative platform.
- **Chapter 8:** This dissertation is concluded and ideas for future work are discussed.

University of Pretoria etd – Peer, E S (2005)

Chapter 2

Computational Intelligence

> *"If computers get too powerful, we can organize [sic] them into a committee – that will do them in." — Bradley's Bromide*

The formulation of a precise definition for Computational Intelligence (CI) and how it relates to the broader Artificial Intelligence (AI) field is a challenging task. Arguably, CI comprises of those paradigms in AI that relate to some kind of biological or naturally occurring system. General consensus suggests that these paradigms are neural networks, evolutionary computing, swarm intelligence and fuzzy systems [29, 31, 88, 130]. Neural networks are based on their biological counterparts in the human nervous system. Similarly, evolutionary computing draws heavily on the principles of Darwinian evolution observed in nature. Swarm intelligence, in turn, is modelled on the social behaviour of insects and the choreography of birds flocking. Finally, human reasoning using imprecise, or fuzzy, linguistic terms is approximated by fuzzy systems.

Figure 2.1 shows these four primary branches of CI and illustrates that hybrids between the various paradigms are possible. Another, more precise, definition describes CI as the study of adaptive mechanisms to enable or facilitate intelligent behaviour in complex and changing environments [31]. Yet there are other AI approaches, that satisfy both this definition as well as the requirement of modelling some naturally occurring phenomenon, that do not fall neatly into one of the paradigms mentioned thus far. Could it be argued that the definition for CI is in itself complex, changing and fuzzy? A more pragmatic approach might be to specify the classes of problems that are of interest without being too concerned about whether or not the solutions to these problems satisfy any constraints implied by a particular definition for CI.

University of Pretoria etd – Peer, E S (2005)

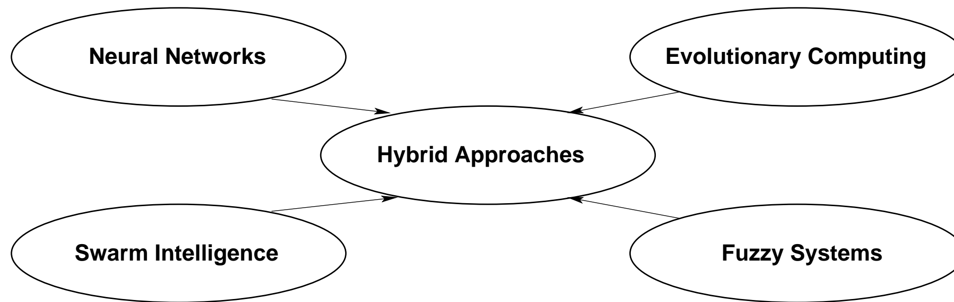


Figure 2.1: Computational Intelligence Paradigms

The following section identifies and describes four primary problem classes for CI techniques. A compendious overview of the main concepts behind each of the widely recognised CI paradigms is presented in Sections 2.2 through 2.5. Further, paradigms that are not generally recognised as CI, but that arguably also classify as such are mentioned in Section 2.6. Examples of hybrid approaches are given in Section 2.7. Finally, a discussion, in Section 2.8, concludes with some software implementation requirements made apparent by the contents of this chapter.

2.1 Problem Classes

Optimisation, defined in Section 2.1.1, is undoubtedly the most important class of problem in CI research, since virtually any other class of problem can be re-framed as an optimisation problem. This transformation, particularly in a software context, may lead to a loss of information inherent to the intrinsic form of the problem. The discussion in Section 2.8 illustrates how these intrinsic features can be exploited in software.

Section 2.1.2 discusses the well known travelling salesman problem as a model representative for the NP-Complete class of problems that are generally thought to be intractable. Function learning and classification, which are characteristic of supervised learning, are presented in Section 2.1.3. Finally, unsupervised learning is represented by clustering in Section 2.1.4.

University of Pretoria etd – Peer, E S (2005)

2.1.1 Optimisation

The process of seeking out values for variables that either minimise or maximise some objective function is known as optimisation [12]. Stated formally, for the case of minimisation:

$$\textit{Given}: \quad f: \mathbb{S} \to \mathbb{R},\ \text{find } \mathbf{x}^* \in \mathbb{S} \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{S} \tag{2.1}$$

where $\mathbb{S}$ represents the search domain which is typically, but but not necessarily, $\mathbb{R}^n$. The minimiser, $\mathbf{x}^*$, is the solution to the minimisation problem defined by the objective function f. The dual problem does not require separate discussion, since, in general, finding the maximiser for an objective function $g : \mathbb{S} \to \mathbb{R}$ is exactly the same as finding the minimiser for $f : \mathbb{S} \to \mathbb{R}$ with $f(\mathbf{x}) = -g(\mathbf{x})$.

When the objective function is defined for a search domain of $\mathbb{R}^n$, further equality and inequality constraints may be defined to restrict the feasible region in which solutions are considered. The constrained optimisation problem is defined formally as follows:

$$\textit{Given}: \quad f: \mathbb{R}^n \to \mathbb{R},\ \text{find } \mathbf{x}^* \in \mathbb{R}^n \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{R}^n \tag{2.2}$$

$$\text{subject to} \quad p_i(\mathbf{x}) = 0,\ i \in \{\mathbb{Z} \mid 1 \leq i \leq r\} \tag{2.3}$$

$$q_j(\mathbf{x}) \geq 0,\ j \in \{\mathbb{Z} \mid 1 \leq j \leq s\} \tag{2.4}$$

where $p_i(\mathbf{x})$ and $q_j(\mathbf{x})$ are respectively, r equality and s inequality constraint functions on the components of the vector $\mathbf{x} \in \mathbb{R}^n$. Constraints of the form $a \leq x_k \leq b$ for $k \in \{\mathbb{Z} \mid 1 \leq k \leq n\}$ can be rewritten as two instances of the single sided inequality constraint of Equation (2.4), namely $q_a(\mathbf{x}) = x_k - a$ and $q_b(\mathbf{x}) = -x_k + b$.

Many algorithms for performing optimisation are designed to be applied to unconstrained optimisation problems, so it is desirable to be able to convert a constrained problem into the form of Equation (2.1) with $\mathbb{S} = \mathbb{R}^n$. A simple method to achieve this is to add to the objective function a suitable penalty term encapsulating the constraints. Thus, the function under optimisation becomes $f(\mathbf{x}) = g(\mathbf{x}) + P(\mathbf{x})$ where $P(\mathbf{x})$ is the penalty term.

Another technique, known as Lagrange's method [69], can be used to convert a constrained problem with equality constraints of the form in Equation (2.3) to an unconstrained problem. The Lagrange function is defined as:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{i=1}^{r} \lambda_i p_i(\mathbf{x}) \tag{2.5}$$

University of Pretoria etd – Peer, E S (2005)

where $f(\mathbf{x})$ and $p_i(\mathbf{x})$ are the same as in Equation (2.2) and (2.3) respectively and the λ_i constants are known as Lagrange multipliers. At the optimal point of intersection the constraint and the objective functions are tangent to each other and so $\nabla f(\mathbf{x}) = \lambda_i \nabla p_i(\mathbf{x})$ provided that $\nabla p_i(\mathbf{x}) \neq 0$. Given Equation (2.5), this is true if and only if $\nabla \mathcal{L}(\mathbf{x}, \lambda) = 0$ so solving the following yields a solution to the original constrained problem:

$$\frac{\delta \mathcal{L}}{\delta x_k} = \frac{\delta \mathcal{L}}{\delta \lambda_i} = 0, \; i \in \{\mathbb{Z} \mid 1 \leq i \leq r\}, \; k \in \{\mathbb{Z} \mid 1 \leq k \leq n\} \tag{2.6}$$

which defines a system of $r + n$ equations that can be cast into an unconstrained optimisation problem by minimising the SSE (Sum Squared Error) defined by:

$$f(\mathbf{x}, \lambda) = \sum_{k=1}^{n} \left(\frac{\delta \mathcal{L}}{\delta x_k} \right)^2 + \sum_{i=1}^{r} \left(\frac{\delta \mathcal{L}}{\delta \lambda_i} \right)^2 \tag{2.7}$$

where the point $(\mathbf{x}, \lambda)$ can be considered as a single vector argument to a function of the form $f(\mathbf{x})$ in Equation (2.1) with $\mathbb{S} = \mathbb{R}^{r+n}$. Inequality constraints can be handled in a similar fashion by introducing slack variables into a modified Lagrangian:

$$\mathcal{L}(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) - \sum_{i=1}^{r} \lambda_i p_i(\mathbf{x}) - \sum_{j=1}^{s} \mu_j (q_j(\mathbf{x}) - e_j) \tag{2.8}$$

where $q_j(\mathbf{x})$ is the same as in Equation (2.4), the μ_j constants are additional Lagrange multipliers and e_j is the slack variable corresponding to the jth inequality constraint.

Optimisation can be further extended into the multi-objective case where the task is to satisfy multiple, possibly conflicting, objectives simultaneously [73]. For example, it may be required that cost be minimised while at the same time benefit is maximised. Some kind of trade off is required when objectives such as these clash, since optimising one necessarily causes deterioration of another. Generally, the goal is to find representative points belonging to the, possibly infinite, pareto optimum set of minimisers given a set of objective functions. A pareto [38], or non-dominated, point is a minimiser for which none of the objectives can be further improved without adversely affecting another. Each of these pareto minimisers represents a different trade off between objectives.

Multi-objective minimisation is formally stated as:

$$\begin{aligned} \textit{Given}: \quad & F(\mathbf{x}) = \{f_k(\mathbf{x}) \mid f_k : \mathbb{S} \rightarrow \mathbb{R}\}, \; k \in \{\mathbb{Z} \mid 1 \leq k \leq m\} \\ \text{find} \quad & X^* = \{\mathbf{x}^* \in \mathbb{S} \mid F(\mathbf{x}^*) \preccurlyeq F(\mathbf{x}), \; \forall \mathbf{x} \in \mathbb{S}\} \\ \text{where} \quad & F(\mathbf{x}) \preccurlyeq F(\mathbf{y}) \iff (\forall_i)(f_i(\mathbf{x}) \leq f_i(\mathbf{y})) \wedge (\exists_i)(f_i(\mathbf{x}) < f_i(\mathbf{y})) \end{aligned} \tag{2.9}$$

University of Pretoria etd – Peer, E S (2005)

where X^* is a representative set of non-dominated minimisers and $F(\mathbf{x})$ is the set of m objective functions. The expression $F(\mathbf{x}^*) \preccurlyeq F(\mathbf{x})$ denotes that $\mathbf{x}^*$, a pareto minimiser, dominates the point $\mathbf{x}$ which is not an element of the pareto set. Once again, the search domain $\mathbb{S}$ may be $\mathbb{R}^n$ and further constrained by Equations (2.3) and (2.4).

If only a single solution in the pareto set is required then multi-objective optimisation can be converted into a single objective optimisation problem of the form in Equation (2.1) by defining the objective as:

$$f(\mathbf{x}) = \sum_{k=1}^{m} w_k f_k(\mathbf{x}) \tag{2.10}$$

which is simply a weighted sum over the set of objective functions that comprise $F(\mathbf{x})$. By varying the weights w_k and performing sequential optimisation passes multiple solutions in the pareto set may be obtained.

2.1.2 NP-Complete Problems

The Travelling Salesman Problem (TSP) [52], a well known problem in computer science, belongs to the NP-Complete class of problems and has been chosen for discussion as a representative for its class. The best known deterministic algorithms able to solve problems of this class execute in exponential-time, or worse, in proportion to the amount of input data.

However, they all have Non-deterministic Polynomial-time (NP) solutions that, in order to yield correct results, require guessing correctly at every decision point during execution by means of some magical non-deterministic process. While such a magical algorithm does not have much practical use, this property does at least guarantee the existence of a short certificate that can be used to validate whether a given solution is correct or not. No polynomial-time deterministic algorithms are known to exist for these problems and as such they are considered to be intractable.

Furthermore, a subset of these problems known as NP-Complete are all polynomial-time reducible amongst themselves, meaning that finding an effective solution to one problem in NP-Complete implies having an effective solution to all those in NP-Complete. Certain CI algorithms, which are by their nature non-deterministic, can be applied in an attempt to yield approximate solutions, given large data sets, in a reasonable amount of time.

University of Pretoria etd – Peer, E S (2005)

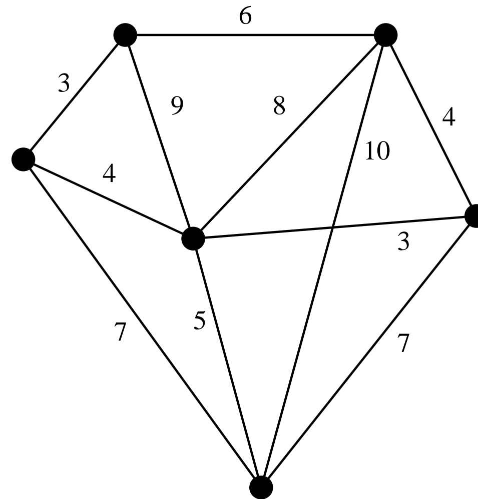


Figure 2.2: Example TSP Network (not to scale)

Problems in NP-Complete include knapsack packing, scheduling, graph colouring and testing the satisfiability of propositional calculus formulae amongst many other distinct problems. Some of these appear to be toy problems, such as the monkey puzzle problem [52], while others have important real world applicability. However, due to their polynomial-time inter-reducibility, all of them are actually of relatively equivalent importance.

In particular, the TSP has real world application in route optimisation, circuit design and the programming of industrial robots [52]. Moreover, the TSP is an ideal candidate for discussion, because it admits an interesting ant system solution (refer to Section 2.4.2) and, as described shortly, can also be cast into a constrained optimisation problem, as defined in the previous section.

The TSP concerns a salesman that must travel from city to city selling his wares before returning back to his city of origin. Each city must be visited exactly once and the distance travelled must be minimised. The problem can be characterised by a graph where each vertex represents a city while the edges correspond to the possible routes between cities and their associated costs. The goal is to determine the shortest closed tour that passes through each of the nodes in the graph for a given network. Figure 2.2 shows a possible network of cities while Figure 2.3 illustrates the optimal tour for that network which is of length 28.

University of Pretoria etd – Peer, E S (2005)

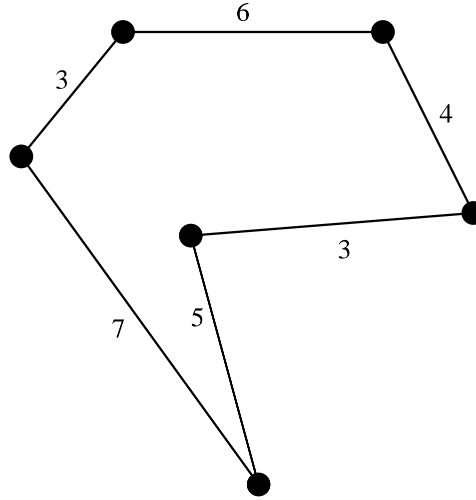


Figure 2.3: TSP Optimal Tour (length = 28)

By imposing an arbitrary ordering from 1 to n on the cities the problem can be redefined as determining the permutation π of visits that yield a minimal length tour. The problem is then reduced to the following constrained optimisation problem [83]:

$$\begin{aligned}
\textit{Given}: \quad & f(\mathbf{x}) = \sum_{i,j} c_{i,j} x_{i,j}, \ i,j \in \{\mathbb{Z} \mid 1 \leq i,j \leq n\} && (2.11) \\
\text{find} \quad & \mathbf{x}^* \in \mathbb{Z}^{n \times n} \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}), \ \forall \mathbf{x} \in \mathbb{Z}^{n \times n} \\
\text{subject to} \quad & \sum_{k=1}^{n} x_{k,i} - 1 = 0 \text{ and } \sum_{k=1}^{n} x_{i,k} - 1 = 0 && (2.12) \\
& x_{i,j} - 1 \leq 0 \text{ and } - x_{i,j} \leq 0 && (2.13) \\
& u_i - u_j + n x_{i,j} - n + 1 \leq 0 \text{ for } j \neq 1 && (2.14)
\end{aligned}$$

where $c_{i,j}$ is the cost of travelling from city i to j. In general, $c_{i,j} = c_{j,i}$ is not necessarily true, $c_{i,j} = \infty$ if no route from i to j exists, and $c_{i,j} = 0$ whenever $i = j$. Equation (2.13) restricts the $x_{i,j}$ to the boolean values 0 and 1 so that $x_{i,j} = 1$ can be taken to mean that city j is visited immediately after i and Equation (2.12) expresses that exactly one city just before and exactly one city just after the ith city is visited. By defining $\pi(u_i) = i$, so that $u_i = j$ implies that i is the jth city visited, a single closed tour is guaranteed by Equation (2.14). Together these constraints ensure that $x_{i,j} = 1 \iff \pi(i) = j$ and $x_{i,j} = 0 \iff \pi(i) \neq j$ when Equation (2.11) is minimised.

University of Pretoria etd – Peer, E S (2005)

2.1.3 Supervised Learning

Supervised learning is the process of determining the intrinsic characteristics of a system using only examples of its operation [84]. The most generic form of supervised learning is function approximation, stated formally:

$$\begin{aligned} \textit{Given}: \quad & P = \{(\mathbf{x},\ \mathbf{t}) \mid \mathbf{x} \in \mathbb{S},\ \mathbf{t} \in \mathbb{T}\} \\ \text{find} \quad & f : \mathbb{S} \rightarrow \mathbb{T} \text{ such that } f(\mathbf{x}) \approx \mathbf{t},\ \forall (\mathbf{x},\ \mathbf{t}) \in P \end{aligned} \tag{2.15}$$

where P is a set of example patterns that demonstrate the operation of the system described by the function f. The pair $(\mathbf{x},\ \mathbf{t})$ is known as a training pattern where $\mathbf{x}$ is an input to the system under learning and $\mathbf{t}$ is the target output. $\mathbb{S}$ and $\mathbb{T}$ may be any domains. The process is called supervised learning because target values are provided for given inputs by some external "teacher" that understands the working of the system.

Care must be taken to ensure that the learning process does not over-fit the data [42]. Over-fitting may occur when the target function is afforded more degrees of freedom or less example patterns than are necessary to describe the system under learning. Under these circumstances the function may fit noise inherent in the data set or other very specific features that have no causal relation to the intrinsic characteristics of the system. Conversely, under-fitting occurs when the target function is not afforded enough degrees of freedom to properly model the underlying data.

The goal is to find a function that has good generalisation ability. This is measured by the ability of the learned function to correctly approximate the target output for inputs that the learning process was not exposed to. For this reason, the example patterns are typically partitioned into separate training and validation sets. Learning is performed using the training set while the validation set is used to test for over-fitting and generalisation ability. An over-fitted function will correctly model the training set while performing poorly on the validation set. On the other hand, a function with the ability to generalise well properly describes the intrinsic characteristics of the system under learning.

Supervised learning manifests itself in many forms including classification, pattern recognition and control problems. For classification problems, the function f in Equation (2.15) is a labelling function that assigns a class to an input pattern where $\mathbb{T}$ is some set of classes. Pattern recognition is just a special case of classification problem.

University of Pretoria etd – Peer, E S (2005)

For example, in handwriting recognition, input patterns might correspond to bitmaps of hand written characters and the set of classes consists of alphanumeric assignments to those bitmaps. In control problems the function relates the sensory input of a system under control to the required output actions.

By defining a suitable parameterisation τ that describes the composition of the function f in Equation (2.15), supervised learning can be reduced to a minimisation problem of the form in Equation (2.1) as follows:

$$g(\tau) = \sum_{i=1}^{n} (\mathbf{t}_i - f(\mathbf{x}_i))^2, \text{ where } \tau \Rightarrow f \tag{2.16}$$

so that $g(\tau)$ is the SSE over the n training patterns, with $t \in \mathbb{R}$, for a function $f : \mathbb{S} \to \mathbb{R}$ implied by the parameterisation τ. Any suitable distance based metric can be used to support targets having arbitrary domains.

There are many ways to define the parameterisation τ. Supervised learning neural networks define very specific functions that are parameterised by weights (refer to Section 2.2). As another example, under the assumption that $\mathbf{x} \in \mathbb{R}^m$ and that the function can be approximated by a polynomial of degree n in each dimension, the following is a suitable definition:

$$f(\mathbf{x}, \tau) = \sum_{i=1}^{m} \sum_{j=0}^{n} \tau_{ij} x_i^j \tag{2.17}$$

where $\tau \in \mathbb{R}^{m \times (n+1)}$ is a matrix of coefficients that parameterise f. Thus, by optimising $g(\tau)$ in Equation (2.16) a function that models the underlying data is constructed.

2.1.4 Unsupervised Learning

Unsupervised learning, also known as self-organisation, requires that a suitable model be fitted to observed patterns without *a priori* knowledge about target outputs for those patterns.

A common unsupervised learning problem is clustering [60] where the goal is to partition observations into homogeneous groupings. The patterns in a given group should be most similar to each other while simultaneously being least similar to observations in other groups, stated formally:

$$\textit{Given}: \quad P = \{\mathbf{p}_t \mid \mathbf{p}_t \in \mathbb{S}\},\ t \in \{\mathbb{Z} \mid 1 \leq t \leq m\}$$

University of Pretoria etd – Peer, E S (2005)

$$\begin{aligned} \text{find} \quad & C_i \subset P,\ \bigcup C_i = P,\ C_i \cap C_j = \emptyset,\ i,j \in \{\mathbb{Z} \mid 1 \leq i,j \leq k,\ i \neq j\} \\ \text{such that} \quad & \mathbf{p}_t \in C_i \Longleftrightarrow \sum_{\mathbf{p}_\alpha \in C_i} d(\mathbf{p}_t,\ \mathbf{p}_\alpha) \leq \sum_{\mathbf{p}_\beta \in C_j} d(\mathbf{p}_t,\ \mathbf{p}_\beta) \end{aligned} \tag{2.18}$$

where $d(\mathbf{x}, \mathbf{y})$ is a suitable distance metric that measures the dissimilarity between $\mathbf{x}$ and $\mathbf{y}$. The k clusters, C_i, are subsets of the set of patterns, P, such that the observations in a given cluster are related by having similar characteristics. If the clusters are pairwise disjoint then the clustering is a true partition. Equation (2.18) only permits such partitions, however, in general it is possible for a given pattern to belong to multiple clusters, with some degree of membership (refer to Section 2.5.1), yielding a fuzzy clustering. The domain, $\mathbb{S}$, of the m input patterns in P can be anything for which a distance metric can be constructed. If $\mathbb{S} = \mathbb{R}^n$ then a suitable Minkowski metric [7] may be used:

$$d_p(\mathbf{x}, \mathbf{y}) = (\sum_{k=1}^{n} |x_k - y_k|^p)^{\frac{1}{p}} \tag{2.19}$$

for some specified value for p where d_1 and d_2 are the well known Manhattan and Euclidean distances respectively.

The number of clusters inherent to a given data set is generally not known. Choosing a value for k that is either too large or too small is analogous, respectively, to over-fitting and under-fitting in supervised learning.

Missing attributes for patterns can be predicted based on related observations in the same cluster. Appropriate clusters for these patterns are determined using the remaining attributes. An over-fitted model which groups related patterns into separate clusters will be unable to accurately predict missing attributes. Similarly, an under-fitted partitioning that groups unrelated patterns into the same cluster will also have poor prediction ability.

Hierarchical clustering, depicted in Figure 2.4, provides a selection of clusterings where each level in the hierarchy roughly corresponds to a different choice for the value of k. Agglomerative clustering is a bottom up approach where each observation is initially assigned to its own cluster. The closest two clusters are then repeatedly merged until all the observations fall into the same cluster at the root of the tree.

Various strategies exist for determining the merging criteria for clusters. Complete linkage clustering utilises the maximum distance between observations in each cluster. If the minimum distance is used instead then the strategy is known as single linkage clustering. An average linkage clustering results when the mean distance between ob-

University of Pretoria etd – Peer, E S (2005)

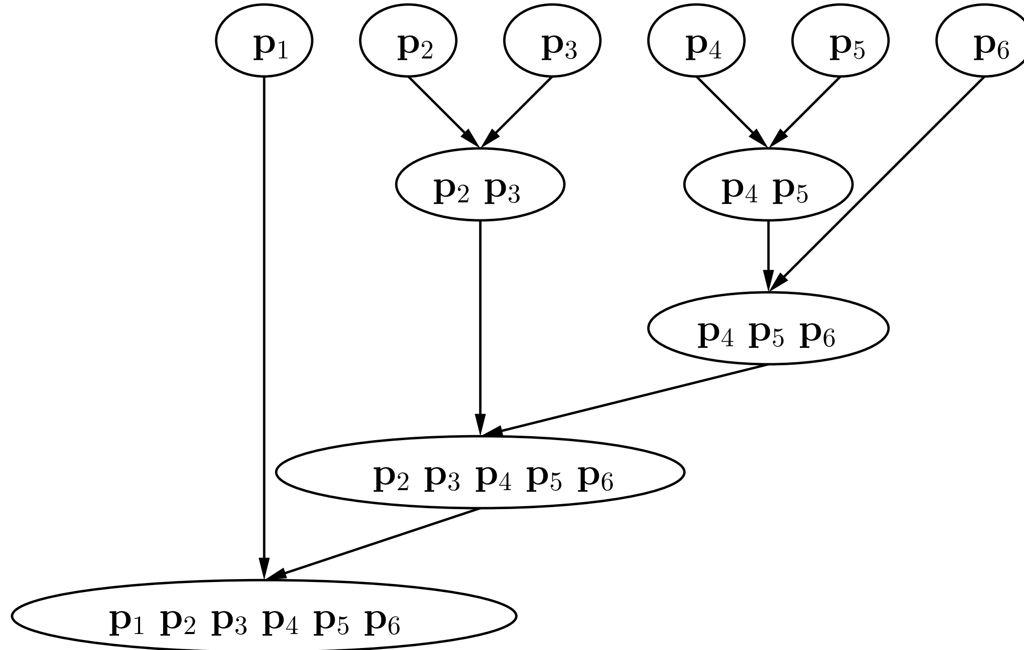


Figure 2.4: Hierarchical Clustering

servations of each cluster is used as a criterion. The average linkage distance between cluster $\mathcal{A}$ and cluster $\mathcal{B}$ is defined as:

$$D(\mathcal{A},\mathcal{B}) = \frac{1}{card(\mathcal{A})card(\mathcal{B})} \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} d(x,y) \tag{2.20}$$

where $card(X)$ is the cardinality of cluster X. Metrics based on intra cluster variance or change in variance (Ward's criterion) are also possible [5].

The clustering problem can be represented by a constrained optimisation problem for a given value of k by determining the optimal assignment vector that maps observations to cluster indexes. One such strategy minimises the distance between observations and the centroids of their clusters, stated formally:

$$\begin{aligned} Given: &\quad f(\mathbf{x}) = \sum_{t=1}^{m} d(\mathbf{p}_t, \mathbf{c}_{x_t}),\ i \in \{\mathbb{Z} \mid 1 \le i \le m\} \\ \text{find} &\quad \mathbf{x}^* \in \mathbb{Z}^m \text{ for which } f(\mathbf{x}^*) \le f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{Z}^m \\ \text{subject to} &\quad -x_i + 1 \le 0 \text{ and } x_i - k \le 0 \end{aligned} \tag{2.21}$$

where $\mathbf{c}_j$ is the centroid of cluster C_j and $\mathbf{x} \in \{\mathbb{Z}^n \mid 1 \le x_i \le k\}$ is the assignment vector such that $x_t = j \iff \mathbf{p}_t \in C_j$.

University of Pretoria etd – Peer, E S (2005)

Clusters defined by a single centroid vector permit only round cluster boundaries. Arbitrarily shaped boundaries can be constructed using a technique known as mixture modelling where each cluster is defined by a weighted density model of different distributions [14].

2.2 Neural Networks

The human brain and nervous system are comprised of billions of nerve cells known as neurons. Each biological neuron is a single cell with receptors called dendrites and an effector called an axon. Neurons are arranged into networks so that the axon of any given neuron can stimulate dendrites of other neurons. When a neuron receives sufficient input stimulus via its dendrites, it fires a signal along its axon which in turn further stimulates the dendrites of other neurons. The arrangement of these relatively simple cells into complex networks generally enables intelligent behaviour in people.

In a similar fashion, the fundamental building block of neural networks in CI is the artificial neuron. By combining these neurons into more complex structures both supervised and unsupervised learning problems can be solved. The canonical feed forward neural network, used for supervised learning, is presented in Section 2.2.1. Other supervised network architectures are mentioned in Section 2.2.2. Unsupervised neural networks such as the learning vector quantiser and self organising feature maps are discussed in Sections 2.2.3 and 2.2.4 respectively.

2.2.1 Feed Forward Neural Networks

Feed forward neural networks can be used to represent nonlinear multivariate relationships [31, 88]. Figure 2.5 illustrates a fully connected three layer network. The layers consist of neurons which compute a function of their inputs and pass the result to the neurons in the following layer. In this manner, the input signal is fed forward from left to right through the network.

The output of a given neuron is characterised by a nonlinear activation function, a weighted combination of the incoming signals, and a threshold value. The threshold can be replaced by augmenting the weight vector to include the input from a constant bias unit. By varying the weight values of the links, the overall function which the network

University of Pretoria etd – Peer, E S (2005)

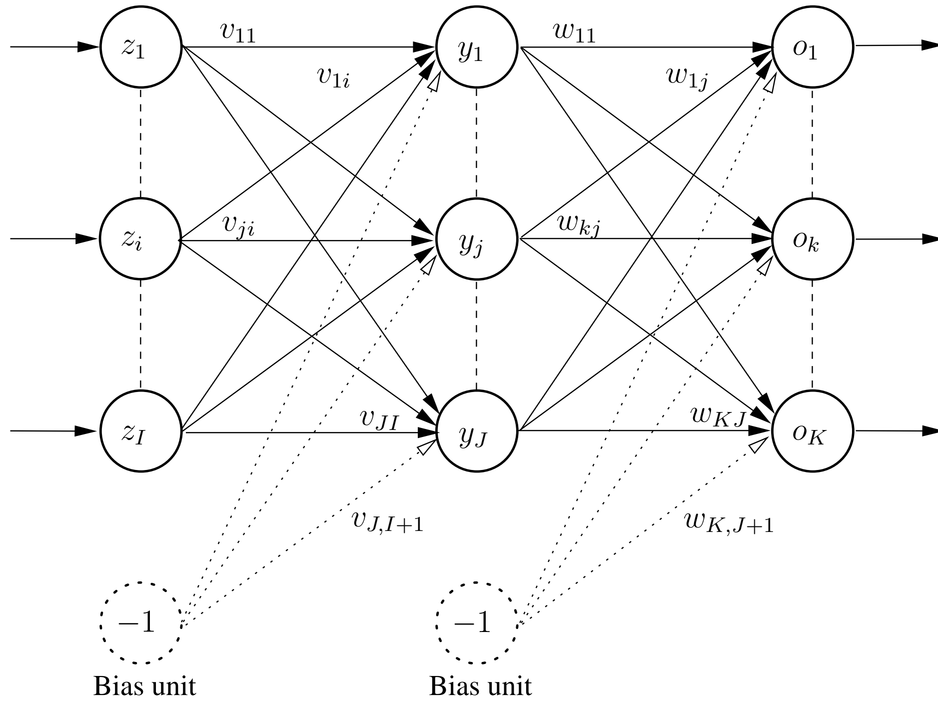


Figure 2.5: Three Layer Feed Forward Neural Network

realises is altered.

The activation signal, o_k for the kth output neuron, for a network with I input, J hidden and K output neurons is given by:

$$o_k = f_{o_k}(\sum_{j=1}^{J+1} w_{kj} y_j) \tag{2.22}$$

$$= f_{o_k}\Big(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1} v_{ji} z_i)\Big) \tag{2.23}$$

where v_{ji} and w_{kj} are weights connecting neurons in their respective layers, y_j is the activation signal of the jth hidden neuron, and z_i is the ith input signal. The activation functions f_{y_j} and f_{o_k} are typically the sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.24}$$

which forces outputs into the range $(0, 1)$. Thus, a feed forward network having I inputs, K outputs and sigmoid activation functions realises a nonlinear mapping of the form

University of Pretoria etd – Peer, E S (2005)

$\mathbb{R}^I \rightarrow (0,1)^K$ which is parameterised by the weights v_{ji} and w_{kj}. Alternative activation functions are mentioned in Section 2.2.2.

Training involves finding values for the weights so that the network best approximates the function for a given supervised learning problem (refer to Equation (2.15)). Since the network can only realise values in the range $(0,1)$, target values must be scaled appropriately. In addition, inputs should also be scaled to fall within the active region of the activation functions which, in the case of sigmoid activations, is roughly $[-\sqrt{3}, \sqrt{3}]$. Classification problems are encoded by dedicating a separate output to each label, so that each output represents the posterior probability that an observation belongs to the class associated with that output.

Algorithm 1 Neural Network Back-propagation Training

1: Initialise $v_{ji}, w_{kj} \sim U(-1,1)$
2: $t \leftarrow 0$
3: **repeat**
4:    **for all** training patterns **do**
5:       $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}(t) + \alpha \Delta w_{kj}(t-1)$ (refer to Equation (2.25))
6:       $v_{ji} \leftarrow v_{ji} + \Delta v_{ji}(t) + \alpha \Delta v_{ji}(t-1)$ (refer to Equation (2.26))
7:    **end for**
8:    $t \leftarrow t+1$
9: **until** stopping condition

Pseudocode for back-propagation learning using gradient descent is presented as Algorithm 1 [116]. Weights are uniformly initialised to small random values and are iteratively updated for each pattern until some stopping criterion is met. The change in output layer weights, derived from the derivative of the SSE over the network, is given by:

$$\Delta w_{kj} = \eta(t_k - o_k)(1 - o_k)o_k y_j \tag{2.25}$$

and the change in hidden layer weights is propagated back using:

$$\Delta v_{kj} = z_i \sum_{k=1}^{K} (1 - y_j) w_{kj} \Delta w_{kj} \tag{2.26}$$

where t_k is the target for the kth output neuron and η is the learning rate. A momentum term which preserves the velocity of weight updates is specified by α.

University of Pretoria etd – Peer, E S (2005)

Instead of simple gradient descent, scaled conjugate gradient techniques [10] or indeed almost any optimisation process could be used to determine appropriate weight values.

2.2.2 Different Network Architectures

There are many ways in which supervised neural network architectures can be customised. Although the number of input and output neurons is defined by the problem, the number of hidden neurons can be varied. At the individual neuron level, different activation functions and methods by which input signals are combined can be utilised. Finally, the network topology can be altered implicitly through dynamic growing, pruning and regularisation; or explicitly at design time as is the case for recurrent and time delay neural networks [31].

Varying the number of hidden neurons affects the complexity of mappings that can be realised by a given neural network. A network with more weights and neurons has more expressive power than one having fewer degrees of freedom. Increasing the number of hidden neurons, however, may lead to over-fitting, since the network would be able to fit inherent noise more easily. Training time is also increased, since more weight updates are required.

In order to fit arbitrary data without over-fitting, the simplest network possible is desired. Regularisation [46, 118] involves driving network weights to zero, in effect removing links to alter the topology, by adding a penalty term to the network error surface that penalises network complexity. Other approaches involve growing or pruning the network by adding or removing neurons respectively when certain triggering criteria are met [31].

Product unit networks [27] utilise higher order combinations of inputs and as such can realise more complex functions with fewer neurons than ordinary summation unit networks. The drawback of a product unit network is that many local minima exist in the error surface causing gradient descent based training algorithms to become trapped at suboptimal solutions more easily. Functional link networks [43] make higher order functions of the inputs available to the hidden layer in an attempt to realise more complex functions with standard summation units.

Sigmoid activation functions are the most common, however, other functions may be used instead. The type of problems for which supervised networks are used typically

University of Pretoria etd – Peer, E S (2005)

exhibit nonlinear behaviour. Linear activation functions may be better suited for linearly related data, but will perform poorly for nonlinear relationships. Step functions model binary characteristics in data while ramp functions can realise a mixture between binary and linear relationships. The hyperbolic tangent has a range of $(-1, 1)$, making it suitable for use in hidden layers, since its output nominally falls within the active input region of typical activation functions. The training process should, however, cause weights to be chosen such that inputs lie in the active region irrespective of the output from the previous layer. Although any conceivable activation function may be used, including Gaussians, there is by definition of supervised learning no *a priori* knowledge about the relationship between inputs and targets. As long as their is no good reason to favour one activation function over another, the relative simplicity of the sigmoid makes it most suitable. A combination of sigmoids in the hidden layer and linear output units has also proven to be a good choice [14].

Various network topologies that attempt to model temporal characteristics in data are also possible [54]. Recurrent neural networks attempt to model these temporal characteristics by storing the signal from the hidden or output layers and feeding it back as additional inputs for subsequent training patterns. In a similar fashion, time delay networks maintain the inputs from previous passes as additional inputs to the network.

2.2.3 Learning Vector Quantiser

The Learning Vector Quantiser (LVQ), shown in Figure 2.6, is a two layer unsupervised learning neural network [66]. The input layer has direct connections to the output neurons and there are no bias units. Unlike supervised networks, the weights in an LVQ network have a special meaning. The kth output neuron, o_k, represents a cluster with an I-dimensional centroid comprising the incoming weights, v_{ki}.

Algorithm 2 outlines the training procedure for an LVQ network. As is the case for supervised networks, the weights are initialised to small uniform random values and training patterns are repeatedly presented to the network causing changes to the weight values.

The weights of the nearest output neurons to a given pattern are updated according to the following equation:

$$\Delta v_{ki}(t) = \eta(t)[z_i - v_{ki}(t-1)] \tag{2.27}$$

University of Pretoria etd – Peer, E S (2005)

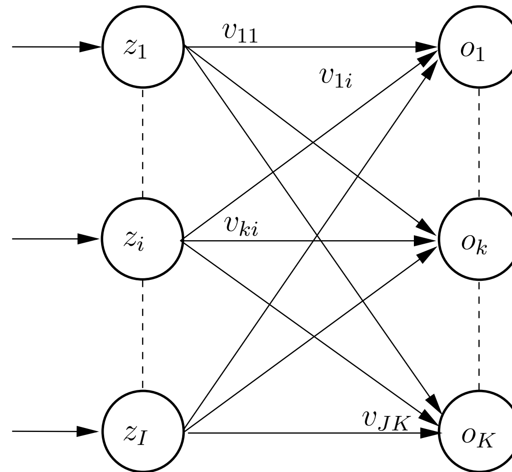


Figure 2.6: Two Layer Learning Vector Quantiser

where $\eta(t)$ is a decaying learning rate so that $\eta(t) \rightarrow 0$ as $t \rightarrow \infty$. The closest output neuron is determined using the Euclidean distance between the training pattern, $\mathbf{z} \in \mathbb{R}^I$ and the weight vector, $\mathbf{v}_k$, that corresponds to o_k. The set $\kappa_k(t)$ consists of output neuron indices considered to be in the neighbourhood of o_k at time t. The neighbourhood, like the learning rate, is also reduced over time so that $\kappa_j(t) \rightarrow \{j\}$ as $t \rightarrow \infty$. In addition to the absolute winner j, in terms of closest output neuron, the weights of all the neurons in $\kappa_j(t)$ are typically also updated. A conscience factor can be incorporated into the distance metric in line 5 to penalise output neurons that overly dominate during training [31]. The result is that cluster centroids, represented by the weights of their respective output neurons, are moved towards the most appropriate input patterns.

2.2.4 Self Organising Feature Maps

Conceptually, a Self-Organising Feature Map (SOFM) [66] functions similarly to an LVQ. In fact, the training algorithm is virtually identical. The most notable difference is that the output layer is a two-dimensional map as shown in Figure 2.7. One of the key benefits of SOFMs over LVQ is that the topology of the input space is preserved in the map. That is, if two patterns are closely related in the input space then they usually map to output neurons that are close to each other in terms of coordinate indices in the map. Thus, SOFMs project an I-dimensional input space onto a two-dimensional map space making them a useful data visualisation tool [31].

University of Pretoria etd – Peer, E S (2005)

Algorithm 2 Learning Vector Quantiser Training

1: Initialise $v_{ki} \sim U(-1, 1)$
2: $t \leftarrow 0$
3: **repeat**
4: **for all** training patterns **do**
5: Find j for which $d_2(\mathbf{z}, \mathbf{v}_j)$ is minimised (refer to Equation (2.19))
6: **for all** $k \in \kappa_j(t)$ **do**
7: $v_{ki} \leftarrow v_{ki} + \Delta v_{ki}(t)$ (refer to Equation (2.27))
8: **end for**
9: **end for**
10: $t \leftarrow t + 1$
11: **until** stopping condition

Although SOFM weights may also be initialised to small uniformly distributed random values, there is a better method of performing initialisation that may improve the quality of the mapping [107]. The weights corresponding to the four corners of the map are initialised to the respective four most extreme patterns in the training set. The remaining weights, $\mathbf{v}_{kj}$, are interpolated as follows:

$$\mathbf{v}_{1j} = \frac{\mathbf{v}_{1J} - \mathbf{v}_{11}}{J-1}(j-1) + \mathbf{v}_{11} \tag{2.28}$$

$$\mathbf{v}_{Kj} = \frac{\mathbf{v}_{KJ} - \mathbf{v}_{K1}}{K-1}(j-1) + \mathbf{v}_{K1} \tag{2.29}$$

$$\mathbf{v}_{k1} = \frac{\mathbf{v}_{K1} - \mathbf{v}_{11}}{K-1}(k-1) + \mathbf{v}_{11} \tag{2.30}$$

$$\mathbf{v}_{kJ} = \frac{\mathbf{v}_{KJ} - \mathbf{v}_{1J}}{J-1}(k-1) + \mathbf{v}_{1J} \tag{2.31}$$

$$\mathbf{v}_{kj} = \frac{\mathbf{v}_{kJ} - \mathbf{v}_{k1}}{J-1}(j-1) + \mathbf{v}_{k1} \tag{2.32}$$

for a JxK map with $j \in \{\mathbb{Z} \mid 2 \leq j \leq J-1\}$ and $k \in \{\mathbb{Z} \mid 2 \leq k \leq K-1\}$.

The standard SOFM training algorithm is identical to LVQ except that the weight update for each neuron is now given by:

$$\mathbf{v}_{kj}(t+1) = \mathbf{v}_{kj}(t) + \eta(t)\Phi_{c_{xy},c_{jk}}(t)[\mathbf{z} - \mathbf{v}_{kj}] \tag{2.33}$$

where $\eta(t)$ is once again a decaying learning rate. The coordinates c_{xy} and c_{jk} are the locations of the winning and current neurons respectively on the map. Again, the winning

University of Pretoria etd – Peer, E S (2005)

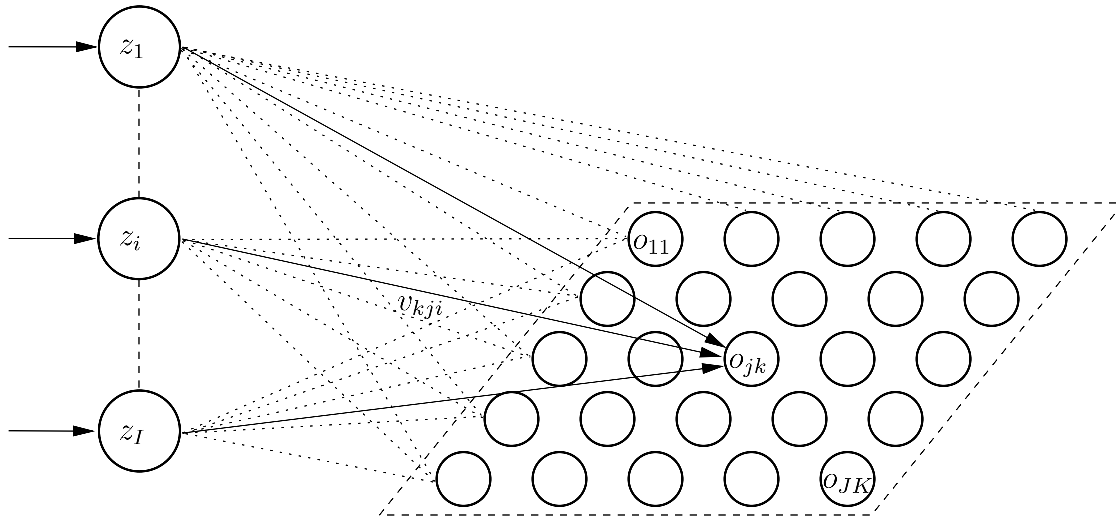


Figure 2.7: 5x5 Self Organising Feature Map

neuron is the one having the closest weight vector, in terms of Euclidean distance, to the current training pattern $\mathbf{z} \in \mathbb{R}^I$. Unlike LVQ, every neuron is typically updated for each training pattern instead of only updating those neurons in an explicit neighbourhood set. The neighbourhood function, $\Phi_{c_{xy},c_{kj}}(t)$, determines the extent which a training pattern has influence over the weights surrounding the winning neuron. Thus, neurons further away from the winning neuron, in map coordinate space, are affected less by a given training pattern. The following Gaussian neighbourhood function is typically used:

$$\Phi_{c_{xy},c_{jk}}(t) = e^{-\frac{||c_{xy}-c_{jk}||_2^2}{2\sigma^2(t)}} \tag{2.34}$$

where $\sigma(t)$ gives the width of the kernel and $\sigma(t) \to 1$ as $t \to \infty$.

A typical SOFM has more output neurons than there are clusters inherent in the training data. Thus, a single output neuron will not, in general, correspond to a single cluster centroid. A unified distance matrix (U-matrix) can be constructed to determine the actual cluster boundaries [31]. The U-matrix is constructed by calculating the distances between each neuron's weight vector and its immediate neighbours in map coordinate space. Large values in the U-matrix are indicative of cluster boundaries while small values indicate groups of neurons belonging to the same cluster. If the map has a high enough resolution then the U-matrix can be plotted as a two-dimensional image that is useful for data visualisation. Figure 2.8 is an example of such a plot with clus-

University of Pretoria etd – Peer, E S (2005)

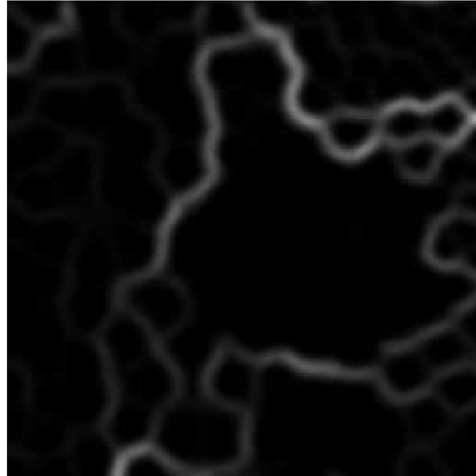


Figure 2.8: Example U-matrix plot

ter boundaries illustrated by white contours that correspond to large U-matrix values. These high resolution maps allow for arbitrary shaped cluster boundaries.

2.3 Evolutionary Computing

All living organisms, ranging from the single celled Amoeba to complex multi-cellular human beings, have a genetic blueprint that describes their physical and behavioural characteristics. This genetic blueprint is made up of DNA (Deoxyribonucleic Acid) arranged into chains of nucleotides called chromosomes. The precise arrangement of the different nucleotides, or genes, defines the characteristics of an organism. The information encapsulated by the DNA is known as the genotype of an organism, while the phenotype is the physical expression of that information. The relationship between genotype and phenotype is typically complex, owing to the influence of pleiotropy and polygeny [77].

Small changes in the genetic material of a population are realised through random mutations and recombination during reproduction between individuals. These changes to the genotype of individuals affect their phenotype and consequently their ability to survive in a given environment. Darwinian theory states that the evolution of a species is guided by competition and natural selection [82]. That is, useful changes in genetic material are preserved from generation to generation, since individuals with better char-

University of Pretoria etd – Peer, E S (2005)

acteristics are the most likely to survive and reproduce.

Algorithm 3 General Evolutionary Computing Framework

1: $t \leftarrow 0$
2: $P(t) \leftarrow \text{initialise}(\mu)$
3: $F(t) \leftarrow \text{evaluate}(P(t), \mu)$
4: **repeat**
5: $\quad P'(t) \leftarrow \text{recombine}(P(t), \Theta_r)$
6: $\quad P''(t) \leftarrow \text{mutate}(P'(t), \Theta_m)$
7: $\quad F(t) \leftarrow \text{evaluate}(P''(t), \lambda)$
8: $\quad P(t+1) \leftarrow \text{select}(P''(t), F(t), \mu, \Theta_s)$
9: $\quad t \leftarrow t+1$
10: **until** stopping condition

Evolutionary Computing (EC) is strongly based on the principles of natural evolution. A general framework for evolutionary optimisation that encompasses these principles is given in Algorithm 3 [109]. A population of μ individuals is initialised within the search space of an optimisation problem so that $P(t) = \{\mathbf{x}_i(t) \in \mathbb{S} \mid 1 \leq i \leq \mu\}$. The search space $\mathbb{S}$ may be the genotype or phenotype depending on the particular evolutionary approach being utilised. The fitness function f, which is the function being optimised, is used to evaluate the goodness individuals so that $F(t) = \{f(\mathbf{x}_i(t)) \in \mathbb{R} \mid 1 \leq i \leq \mu\}$. Obviously, the fitness function will also need to incorporate the necessary phenotype mapping if the genotype space is being searched.

Searching involves performing recombination of individuals to form offspring, random mutations and selection of the following generation until a solution emerges in the population. The parameters Θ_r, Θ_m and Θ_s are the probabilities of applying the recombination, mutation and selection operators respectively. Recombination involves mixing the characteristics of two or more parents to form offspring in the hope that the best qualities of the parents are preserved. Mutations, in turn, introduce variation into the population thereby widening the search. In general, the recombination and mutation operators may be identity transforms so that it is possible for individuals to survive into the following generation unperturbed. Finally, the λ new or modified individuals are re-evaluated before the selection operator is used to pare the population back down to a size of μ. The selection operator provides evolutionary pressure so that the most fit in-

University of Pretoria etd – Peer, E S (2005)

dividuals survive into the next generation. While selection is largely based on the fitness of individuals, it is probabilistic to prevent premature convergence of the population.

Genetic algorithms, which generally search the genotype space, are summarised in the next section. Section 2.3.2 covers a specialisation of genetic algorithms where the genotype is a space of executable program trees. Evolutionary programming, discussed in Section 2.3.3, concentrates on searching the phenotype space. Evolutionary strategies, which dynamically evolve strategy parameters, are discussed in Section 2.3.4. Finally, cultural and co-evolutionary extensions are considered in Sections 2.3.5 and 2.3.6 respectively.

2.3.1 Genetic Algorithms

Genetic Algorithms (GAs) [47] fit neatly into the general EC framework already presented in Algorithm 3. Thus, the only remaining requirement, to fully describe a GA, is the definition of a specific genotype representation along with suitable recombination, mutation and selection operators.

Traditional GAs [56] represent individuals as binary bit strings. Numeric phenotypes are usually encoded using Gray's code in the genotype to reduce pleiotropic variation in the phenotype. That is, the genotypic Hamming distance is minimised for small differences in phenotypic values. A real ($\mathbb{R}$) valued genotype, having an identical phenotype, is also possible, provided that recombination and mutation are suitably defined for real values. In fact, any representation, for which suitable operators can be defined, may be used. For example, genetic programming, presented in the following section, is a special type of GA having a tree based representation.

Reproduction, or the mixing of genetic material, between multiple individuals is known as crossover in the context of GAs. Figure 2.9 illustrates three types of crossover that can be defined for binary coded individuals. Each of them is defined in terms of a binary mask and is able to produce two offspring from a pairing of two parents. The mask determines the parent from which the offspring inherit their genetic material. In the case of uniform crossover, a random mask is generated that results in offspring composed of random components of the two parent's genetic material. For one-point crossover, a random offset in the mask is chosen, so that all components up to that offset are inherited from the one parent and the rest from the other. Similarly, for two-point

University of Pretoria etd – Peer, E S (2005)

Figure 2.9: Crossover Operators

crossover, there are two offsets chosen so that only the components between the two positions are inherited from the one parent. For real valued genes, arithmetic crossover may be defined for two individuals $\mathbf{x}_a$ and $\mathbf{x}_b$ as follows:

$$\mathbf{x}_a(t+1) = \rho\mathbf{x}_a(t) + (1-\rho)\mathbf{x}_b(t) \tag{2.35}$$

$$\mathbf{x}_b(t+1) = \rho\mathbf{x}_b(t) + (1-\rho)\mathbf{x}_a(t) \tag{2.36}$$

where $\rho \sim U(0,1)$ is a uniform random variate.

Mutation is typically performed with a fairly low probability, since existing good solutions may be disturbed if the mutation rate is too high. A suitable mutation operator for binary coded individuals inverts bits subject to a given probability, while real valued mutation can be achieved by adding Gaussian noise.

An elitism operator is usually implemented to select a few good individuals, the elite, to survive into the following generation. This can be achieved trivially, by adding the new and modified individuals, obtained through recombination and mutation, to the existing population and subjecting the entire pool to selection.

Various selection strategies exist, including tournament, proportional, and rank-based selection [31]. Tournament selection involves repeatedly selecting k individuals randomly from the population and then selecting the individual with the best fitness out of that group. A proportional strategy selects individuals in proportion to their fitness by sampling the following distribution:

$$P(\mathbf{x}_i(t)) = \frac{f(\mathbf{x}_i(t))}{\sum_{n=1}^{\mu} f(\mathbf{x}_n(t))} \tag{2.37}$$

University of Pretoria etd – Peer, E S (2005)

so that $P(\mathbf{x}_i(t))$ is the probability of selecting the ith individual from the population at time t. Finally, rank-based selection techniques sample the rank ordered distribution of individuals instead of considering absolute fitness values.

2.3.2 Genetic Programming

Any algebraic expression can be trivially represented in tree form. Non-terminal tree nodes represent mathematical operators so that their children correspond with the parameters of the operator in question. Variables and constants, in turn, are represented as terminal nodes in the tree. Figure 2.10 is an example tree for the expression $\sin(\frac{p}{q})(\log(r) - e^{s+1.5})$. In a similar fashion, a parse tree, for arbitrary computer programmes in any language, can be constructed.

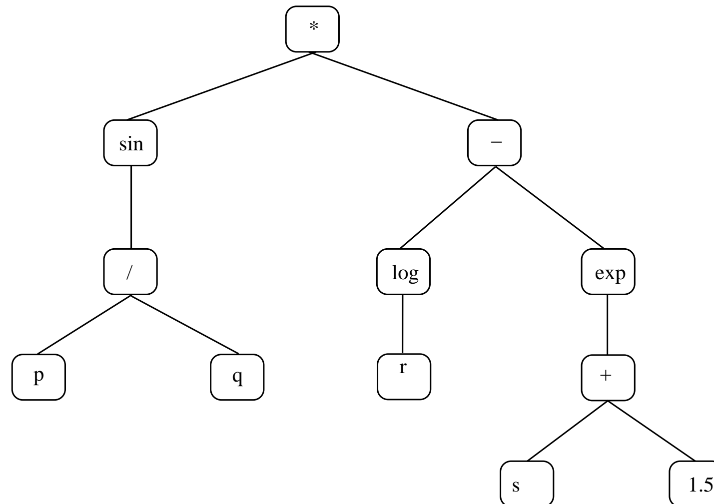


Figure 2.10: Genetic Program Tree Representation

Genetic programmes are nothing more than GAs, with the genotype being parse trees for executable programmes in a given language [67]. Consequently, the phenotype is the behaviour of those programmes at execution time. The fitness function is a measure of how well a programme performs a specified task. Selection is also analogous to GAs, so all that remains is to define suitable crossover and mutation operators for tree structures.

Crossover is trivial, a random node in each parent tree is selected. These two nodes, along with their descendents, are swapped, forming two possible offspring. That is, the selected subtree of one parent is replaced with the selected subtree of the other.

University of Pretoria etd – Peer, E S (2005)

Several mutation operators, which should be used together, can be defined [31]:

- **Function node mutation:** A randomly selected non-terminal node has its operator replaced with another operator that has the same cardinality.
- **Terminal node mutation:** A randomly selected terminal node is replaced with another valid terminal node.
- **Swap mutation:** A non-terminal node, having more than one child, is selected and order of its children are altered.
- **Grow mutation:** A randomly selected node is replaced with a randomly generated subtree that has a predetermined maximum depth.
- **Gaussian mutation:** A terminal node which represents a constant is randomly selected and mutated by adding Gaussian noise.
- **Trunc mutation:** A randomly selected non-terminal node is replaced with a valid terminal node.

2.3.3 Evolutionary Programming

Evolutionary Programming (EP) [36, 37] can be classified in the EC framework in Algorithm 3 by leaving out the fifth step, or equivalently, defining recombination as an identity transform. That is, EP relies solely on mutation and does not make use of any recombination. In addition, EP does not explicitly distinguish between genotype and phenotype. Rather, mutations are defined based on the problem domain, implicitly making EP a phenotypic optimisation process.

EP was originally developed to evolve finite-state machines by defining the following mutations: change an output symbol; change a state transition; add a state; delete a state; or change the initial state. Real valued domains can make use of Gaussian mutation, as is the case for real valued genotypes in GAs. In any event, the mutation operator used will be problem specific, since EP performs a search of the phenotype. Mutation should be biased towards making small changes but should allow for large mutations, particularly early on in the search, to enable the optimisation process to avoid local extrema.

University of Pretoria etd – Peer, E S (2005)

2.3.4 Evolutionary Strategies

The general EC framework defined in Algorithm 3 has many parameters that may affect its performance in various ways. In the context of Evolutionary Strategies (ES) [93, 94], these are known as strategy parameters. The primary principle of ES is to concurrently evolve these strategy parameters alongside the solution to the problem under optimisation. In this way, ES are able to more optimally adapt their strategy to the problem at hand.

Like other EC paradigms, implementations of ES also define their own representation as well as recombination, mutation and selection operators. Canonical ES specify mutation and crossover operators defined for vectors of real values, inherently making ES a phenotypic search process. Thus, the standard representation for ES is a real valued solution vector augmented by one or more strategy parameters so that:

$$\mathbf{x}(t) \in \{(\mathbb{R}^n, \mathbb{R}^s)\} \tag{2.38}$$

for an individual $\mathbf{x}(t)$ of solution dimension n with s strategy parameters. It is possible, however, to apply similar strategy parameters to genotypic search algorithms to enhance their performance. In general, any parameter that influences the evolutionary process can be appended to an individual's representation. Individuals that are performing poorly may have their strategy parameters adjusted more dramatically under the assumption that their poor performance is due to a bad choice of strategy.

Specifically, mutation is enhanced by associating additional parameters with each individual. The simplest of these schemes associates a standard deviation, $\sigma(t)$, with each member of the population so that the mutation operator perturbs the solution vector as follows:

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \sigma(t+1)\xi \tag{2.39}$$

where $\xi \in \mathbb{R}^n$ with each $\xi_i \sim N(0,1)$ a normally distributed random variate, while the standard deviation for each successive generation is updated according to:

$$\sigma(t+1) = \sigma(t)^{e^{\rho\sqrt{n}}} \tag{2.40}$$

where $\rho \sim N(0,1)$. More elaborate schemes that include a standard deviation along with a matrix of rotation angles have also been devised [31].

University of Pretoria etd – Peer, E S (2005)

Crossover can be applied to both the solution vector and the strategy parameters. ES define different crossover operators to standard GAs. Local crossover resembles uniform crossover in that an offspring is created by selecting random components from two parents. Global crossover, however, selects random components from the entire population to generate a single offspring. In addition to simply selecting random components, arithmetic crossover or simple averaging can be performed between multiple parents.

Two primary selection strategies have been defined for ES. The first, known as $(\mu+\lambda)$, selects successive generations from the combination of the previous generation and all the offspring. The second, known as (μ, λ), selects the following generation from the set of offspring only. The former implicitly implements a form of elitism operator while the latter does not allow for individuals to survive through successive generations and requires that $1 \leq \mu < \lambda < \infty$.

2.3.5 Cultural Evolution

Cultural evolution [96] is based on the premise that cultural properties in a population evolve at a faster rate than genetic properties. The search process is biased by a cultural belief space that focuses the search in areas that the population believes contains good solutions. This belief space, which stores the best behavioural traits of the population over time, is used to enhance and accelerate the search process.

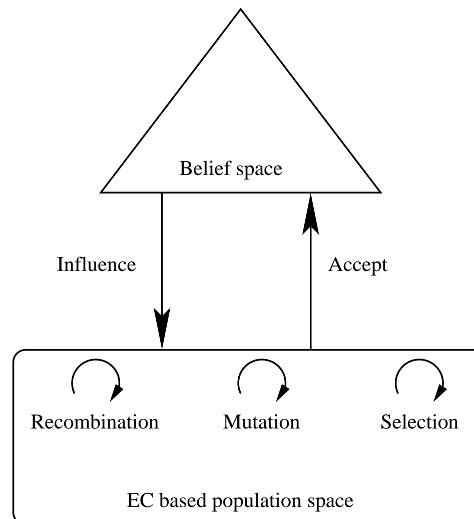


Figure 2.11: Cultural Algorithm

University of Pretoria etd – Peer, E S (2005)

Cultural algorithms deviate from the model given in Algorithm 3 by maintaining two separate search spaces. The first, the population space, is an instance of one of the already mentioned EC algorithms, perhaps a GA or an EP algorithm. Secondly, the belief space serves as a repository of knowledge gained by the main population during the entire search process. Figure 2.11 illustrates the relationship between these two spaces. An acceptance function specifies how this knowledge is communicated from the main population and incorporated into the belief space. An influence function, in turn, determines how the search process of the main population is influenced by the knowledge in the belief space.

The choice of functions that govern acceptance of knowledge into the belief space and the influence of that knowledge on the population are problem specific. In the case of $\mathbb{R}^n$ domains, the belief space may be defined by the intervals in which the solution is believed to exist in each dimension. Thus, the acceptance function is defined as the bounding hyper-rectangle created by a given percentage of the best performing individuals in the population. Influence of the population is achieved through a modified mutation operator. Individuals lying further outside the range defined by the belief space are subjected to larger mutation step sizes while those within the range are mutated by a smaller amount. In this way, individuals are encouraged to search the belief space more thoroughly. Constrained optimisation can also be supported by forcing the conformance of belief space to those constraints.

2.3.6 Coevolution

Coevolution is an extension of EC into multiple competing or cooperating populations which work together to solve a given problem. The fitness of a given individual becomes a subjective measure relative to the other populations being co-evolved.

For cooperating populations, the solution vector may be split into smaller dimensions with each subpopulation solving only the part of the vector for which it is responsible [117]. In this case, fitness must be measured within the context of the other populations since the objective function requires a full length solution vector to be calculated. Alternatively, the search space itself may be partitioned into intervals, or a global "black board" may be used for sharing partial solutions between populations.

In the case of competing populations, a key benefit is that an absolute fitness measure

University of Pretoria etd – Peer, E S (2005)

is not a requirement. The fitness of an individual in one population is measured relative to the performance of individuals in competing populations by playing the individuals against one another [55].

Various sampling strategies for selecting the individuals from other populations that take part in the relative fitness evaluation exist [31]:

- **All versus all:** The fitness for a given individual is calculated relative to all the individuals in other populations.
- **Random:** Fitness is calculated relative to a random group of individuals selected from the other populations.
- **Tournament:** The best individual within a random subgroup of the other populations is selected and fitness is calculated relative to this individual..
- **All versus best:** Fitness is calculated relative to the best performing individual in other populations.

2.4 Swarm Intelligence

Swarm Intelligence models the naturally observed phenomenon of a population, or swarm, of relatively unsophisticated organisms, through their social interactions, to be able to realise globally intelligent behavioural patterns. An example of this phenomenon is the ability of ants to find the most optimal routes to food sources. The individual ants themselves are very simple creatures lacking the ability to think or reason, yet as a colony, they appear able to perform the complex task of determining the optimal routes to food.

Like the EC paradigm discussed in Section 2.3, swarm intelligence approaches are also population based, however, that is where the similarity ends. EC is primarily concerned with evolutionary operators, such as mutation and recombination, to bring about variation in a population, and selection, as a means to focus the search into areas that promise the best results. Swarm intelligence, on the other hand, concentrates on modelling the social interactions between individuals in a population, which usually have a specific task to perform, and typically does not exhibit any kind of selection pressure that governs the survivability of particular individuals.

University of Pretoria etd – Peer, E S (2005)

Particle swarm optimisation, discussed in the following section, exchanges experiential knowledge about the search surface between particles as a means of social interaction. Section 2.4.2 overviews ant systems where interaction between individuals occurs indirectly by means of modifications to the environment in which they function. By modelling these social interactions useful algorithms have been devised for solving numerous problems including function and route optimisation as well as unsupervised clustering.

2.4.1 Particle Swarm Optimisation

Particle swarm optimisation [63, 28] was originally inspired by the flocking behaviour of birds. In terms of this bird flocking analogy, a particle swarm optimiser consists of a number of particles, or birds, that fly around a search space, or the sky, in search of the best location. Each of these particles corresponds to a simple agent that moves through a multi-dimensional search space sampling an objective function at various positions. The motion of a given particle is dictated by its velocity which is continuously updated in order to pull it towards its own best position and the best positions experienced by the rest of the swarm. This behaviour ultimately results in an optimiser that converges to good solutions of an objective function of the form $f : \mathbb{R}^n \to \mathbb{R}$.

The velocity update for each dimension, given by the subscript $j \in \{\mathbb{Z} \mid 1 \leq j \leq n\}$, of the ith particle with position $\mathbf{x}_i(t) \in \mathbb{R}^n$ and velocity $\mathbf{v}_i(t) \in \mathbb{R}^n$ at time t is given by the following equation [63, 28, 100]:

$$v_{i,j}(t+1) = wv_{i,j}(t) \ + c_1 r_{1,j}(y_{i,j}(t) - x_{i,j}(t)) \ + c_2 r_{2,j}(\hat{y}_{i,j}(t) - x_{i,j}(t)) \tag{2.41}$$

where $w \in \{\mathbb{R} \mid 0 \leq w < 1\}$ is an inertia weight that preserves some of the previous velocity; $c1$ and $c2 \in \{\mathbb{R} \mid 0 \leq c_1, c_2 \leq 2\}$ are acceleration coefficients; and $r_{1,j}, r_{2,j} \sim U(0,1)$ are drawn from two independent uniform random distributions. The vector $\mathbf{y}_i(t) \in \mathbb{R}^n$ is the best position found by the individual particle, while $\hat{\mathbf{y}}_i(t) \in \mathbb{R}^n$ represents the best position found by other particles in the swarm. Various neighbourhood strategies determine which particles participate in the social network of a given particle, so that $\hat{\mathbf{y}}_i(t)$ represents the best solution found by the particles in the neighbourhood of the ith particle.

The second term in Equation (2.41) is known as the cognitive component, since it takes into account a particle's own experience of the search terrain. Setting $c_2 \leftarrow 0$

University of Pretoria etd – Peer, E S (2005)

results in a cognition only optimiser having no social interaction between the particles. Conversely, setting $c_1 \leftarrow 0$ leaves only the social component, the third term in the equation. The acceleration coefficients can be chosen (or varied over time) to prioritise the influence of a particle's own cognition or its social interaction with the rest of the swarm. Whenever:

$$\frac{c_1 + c_2}{2} - 1 < w \tag{2.42}$$

holds, particles will exhibit convergent trajectories, otherwise they will not stabilise [113]. Alternatively, a $V_{\max}$ strategy can be used to reduce the likelihood of divergence by enforcing an upper bound on particle velocities.

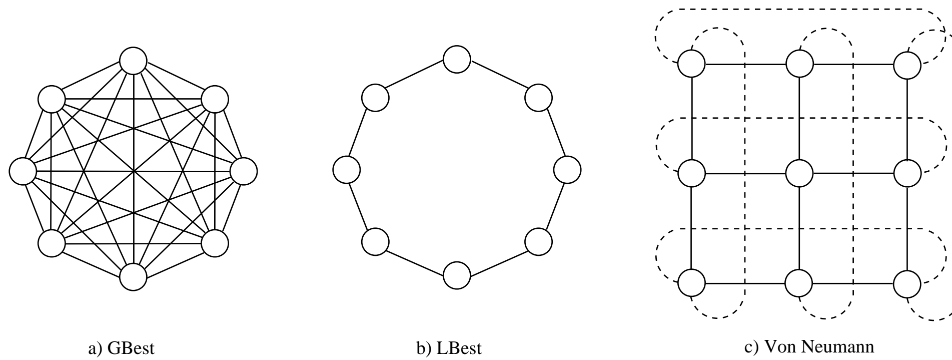


Figure 2.12: Typical Neighbourhood Topologies

The influence of various neighbourhood topologies on the PSO has been been studied extensively [29, 101, 61, 64, 108, 90]. Figure 2.12 illustrates the best known neighbourhood topologies. The GBest, or global best, topology includes every particle of the swarm within the social network of every other particle. LBest, or local best, only considers a particle's immediate neighbours, in terms of particle index, to be socially connected. Finally, the Von Neumann architecture, taking the form of a grid with wrap-around, considers the particles above, below, to the left and to the right to be within a given particle's neighbourhood. The more densely connected the neighbourhood, the quicker information about good solutions is communicated amongst particles in the swarm. Neighbourhood topologies such as LBest and Von Neumann result in superior solutions at the cost of slower convergence, since diversity within the swarm is maintained longer.

Algorithm 4 outlines the Particle Swarm Optimiser (PSO). Initialisation is performed by randomly placing the particles within the search space. All velocities are initialised

University of Pretoria etd – Peer, E S (2005)

Algorithm 4 Particle Swarm Optimiser

1: **for all** particles i **do**
2: Initialise $x_{i,j}(0) \sim U(x_{min,j}, x_{max,j})$
3: $\mathbf{y}_i(0) \leftarrow \mathbf{x}_i(0)$
4: $\hat{\mathbf{y}}_i(0) \leftarrow \mathbf{x}_i(0)$
5: $\mathbf{v}_i(0) \leftarrow \mathbf{0}$
6: **end for**
7: $t \leftarrow 0$
8: **repeat**
9: **for all** particles i **do**
10: **if** $f(\mathbf{x}_i(t)) > f(\mathbf{y}_i(t))$ **then**
11: $\mathbf{y}_i(t) \leftarrow \mathbf{x}_i(t)$
12: **if** $f(\mathbf{x}_i(t)) > f(\hat{\mathbf{y}}_i(t))$ **then**
13: $\hat{\mathbf{y}}_i(t) \leftarrow \mathbf{x}_i(t)$
14: **end if**
15: **end if**
16: Update $\mathbf{v}_i(t+1)$ according to Equation (2.41)
17: $\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$
18: **end for**
19: $t \leftarrow t + 1$
20: **until** stopping condition

to zero and the personal best positions of the particles are their initial positions. Steps 10 through 15 maintain the personal best positions, $\mathbf{y}_i(t)$, as well as the neighbourhood best position, $\hat{\mathbf{y}}_i(t)$, where the fitness function is given by f. Thus, the particle positions are moved, in step 17, towards their own best positions and the best positions found by the swarm according to Equation (2.41). Upon termination, the best solution found to the optimisation problem is given by the position of the particle with the best fitness.

2.4.2 Ant Systems

Artificial ant systems model the social interaction and seemingly intelligent behaviour of naturally occurring colonies of ants. These social interactions are due to a phenomenon

University of Pretoria etd – Peer, E S (2005)

known as stigmergy, characterised by a lack of centralised control and indirect communication by means of modifications to the environment. The emergent behaviour of the colony is observed in their ability to, amongst others, locate optimal food resources and perform nest brooming, including cemetery maintenance [31].

This section describes an optimisation algorithm, applicable to the TSP discussed in Section 2.1.2, followed by an algorithm for performing unsupervised clustering. The former models the way ants optimise paths to food sources, and the latter is based on their cemetery maintenance behaviour.

Ant Colony Optimisation

Foraging in ant colonies is governed by pheromone deposits along paths to food. In general, pheromones are invisible chemicals secreted by organisms which, when detected by the senses, cause an instinctual reaction in another organism. In particular, foraging ants tend to follow paths with higher concentrations of pheromone deposits.

Pheromones are deposited along a given path by the ants that traversed that path at an earlier time. The pheromone following nature of ants combined with the fact that pheromone deposits evaporate over time, results in the shortest paths containing the highest pheromone concentrations. This is because an ant that discovers a shorter path will return sooner, depositing more pheromones, on the way to a food source and again on the way back, as well as more recent pheromones than an ant on a longer path. As more and more ants start to follow the shorter path, due to a higher pheromone concentration, a positive feedback loop is created until virtually all the ants follow the shortest path. Thus, social interaction and coordination for foraging occurs indirectly through pheromone deposits which modify the environment.

Algorithm 5 models the foraging behaviour of ants to solve the TSP (refer to Section 2.1.2) [26]. Each edge of a TSP graph is associated with a pheromone intensity between city i and j at time t denoted by $\tau_{ij}(t)$. The probability, $\Phi_{ij,k}(t)$, for ant k at city i to choose j as the next city to visit is given by:

$$\Phi_{ij,k}(t) = \frac{\tau_{ij}(t)^{\alpha}\eta_{ij}^{\beta}}{\sum_{c \in C_{i,k}} \tau_{ic}(t)^{\alpha}\eta_{ic}^{\beta}} \tag{2.43}$$

where $C_{i,k}$ is the set of city indices that ant k still needs to visit from city i and η_{ij} is the economy of travelling from city i to j. The parameters, α and β, control the respective

University of Pretoria etd – Peer, E S (2005)

Algorithm 5 Ant Colony Optimiser for TSP

1: Initialise $\tau_{ij}(0) \sim U(0, max)$
2: Place all ants $k \in \{\mathbb{Z} \mid 1 \leq k \leq m\}$ at origin city
3: Let T^+ be the shortest tour, and L^+ its length
4: $t \leftarrow 0$
5: **repeat**
6:   **for all** ants k **do**
7:     Build tour $T_k(t)$ by choosing successive cities with probability $\Phi_{ij,k}(t)$ (refer to Equation (2.43))
8:     Compute length of route, $L_k(t)$
9:     **if** $L_k(t) < L^+$ **then**
10:       $T^+ \leftarrow T_k(t)$
11:       $L^+ \leftarrow L_k(t)$
12:     **end if**
13:   **end for**
14:   Update pheromone deposits using Equation (2.44)
15:   $t \leftarrow t + 1$
16: **until** stopping condition

importance of pheromone intensities, $\tau_{ij}(t)$, and local cost information, $\eta_{ij} = 1/d_{ij}$, where d_{ij} is a suitable Minkowski distance metric.

The algorithm randomly initialises the pheromone intensities, places a number, m, of ants at the originating city and then proceeds to iteratively build tours, T_k, for each ant k according to Equation (2.43) while continuously maintaining pheromone updates according to:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{2.44}$$

where ρ is known as a forgetting factor which causes pheromone depletion over time. The net change in pheromone intensity, $\Delta\tau_{ij}(t)$, at time t between city i and j is given by:

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t) \tag{2.45}$$

which is the sum of the deltas over all ants where the contribution of each ant is, in turn,

University of Pretoria etd – Peer, E S (2005)

given by:

$$\Delta\tau_{ij,k}(t) = \begin{cases} Q/L_k(t) & \text{if } (i,j) \in T_k(t) \\ 0 & \text{if } (i,j) \notin T_k(t) \end{cases} \tag{2.46}$$

where Q is of the same order of magnitude as the optimal route length and $L_k(t)$ is the length of the tour just taken by ant k. The contribution of an ant to the pheromone intensity between cities i and j is zero if the ant did not traverse that edge during its tour. When the algorithm terminates, the optimal tour found is given by T^+ and its length by L^+.

Ant Colony Clustering

Several species of ants have been observed to cluster corpses into cemeteries in order to tidy their nests. While not much is known about this behaviour, it has provided the inspiration for an algorithmic solution to the unsupervised clustering problem [15].

Algorithm 6 outlines an approach for clustering using a colony of artificial ants. The fundamental idea is to allow ants to roam a grid containing data vectors, picking up those vectors which are dissimilar from their surrounding vectors and dropping them in areas having more similar vectors.

The local density function, $f(\mathbf{z}_i, r)$, which is a measure of the average similarity of the vector $\mathbf{z}_i$ to the vectors in a neighbourhood around the location r is given by:

$$f(\mathbf{z}_i, r) = \frac{1}{s^2} \sum_{\mathbf{z}_j \in \mathcal{N}_{sxs}(r)} [1 - \frac{d(\mathbf{z}_i, \mathbf{z}_j)}{\alpha}] \tag{2.47}$$

where $\mathcal{N}_{sxs}(r)$ is the set of vectors in a square neighbourhood of width s around r and $d(\mathbf{z}_i, \mathbf{z}_j)$ is the dissimilarity, a Minkowski metric, between two vectors $\mathbf{z}_i$ and $\mathbf{z}_j$ with α controlling the scale of the dissimilarity measure.

An unladen ant at location r which is occupied by a vector $\mathbf{z}_i$ picks up that vector with probability:

$$p_p(\mathbf{z}_i, r) = \left(\frac{k_1}{k_1 + f(\mathbf{z}_i, r)}\right)^2 \tag{2.48}$$

where k_1 is a constant which can be used to tune the sensitivity of the resultant probability to $f(\mathbf{z}_i, r)$. Equation (2.48) has the property that vectors which are highly similar to those in their neighbourhood have a low probability of being picked up. Conversely, lower values of $f(\mathbf{z}_i, r)$ result in a high probability of $\mathbf{z}_i$ being picked up, since $p_p(\mathbf{z}_i, r) \to 1$ as $f(\mathbf{z}_i, r) \to 0$.

University of Pretoria etd – Peer, E S (2005)

Algorithm 6 Ant Colony Clustering

1: Place each data vector $\mathbf{z}_i$ randomly on grid
2: Place all ants $k \in \{\mathbb{Z} \mid 1 \leq k \leq m\}$ randomly on grid
3: **repeat**
4: **for all** ants k **do**
5: Let r be the location of ant k
6: **if** unladen(k) and occupied(r, $\mathbf{z}_i$) **then**
7: Compute $f(\mathbf{z}_i, r)$ and $p_p(\mathbf{z}_i, r)$ (refer to Equations (2.47) and (2.48))
8: **if** $U(0,1) \leq p_p(\mathbf{z}_i, r)$ **then**
9: Pick up data vector $\mathbf{z}_i$
10: **end if**
11: **else if** laden(k, $\mathbf{z}_i$) and empty(r) **then**
12: Compute $f(\mathbf{z}_i, r)$ and $p_d(\mathbf{z}_i, r)$ (refer to Equations (2.47) and (2.49))
13: **if** $U(0,1) \leq p_d(\mathbf{z}_i, r)$ **then**
14: Drop data vector $\mathbf{z}_i$
15: **end if**
16: **end if**
17: Move ant k to randomly selected neighbouring site not occupied by another ant
18: **end for**
19: **until** stopping condition

Alternatively, a laden ant carrying a vector $\mathbf{z}_i$ at an unoccupied location r drops its vector with probability:

$$p_d(\mathbf{z}_i, r) = \begin{cases} f(\mathbf{z}_i, r) & \text{if } f(\mathbf{z}_i, r) < k_2 \\ 1 & \text{otherwise} \end{cases} \tag{2.49}$$

where k_2 is a constant that biases towards dropping vectors as k_2 is made smaller, since $p_d(\mathbf{z}_i, r) \to 1$ as $k_2 \to 0$.

An obvious consequence of Algorithm 6 is that the grid must be large enough to accommodate all the data patterns as well as sufficient ants. Strategies that mitigate over-fitting, such as having ants moving at different speeds, can also be implemented [31].

University of Pretoria etd – Peer, E S (2005)

2.5 Fuzzy Systems

Traditional expert systems [45], which typically use first-order predicate calculus to represent rules, rely on boolean logic where an element either belongs to a set or it does not. That is, the law of the excluded middle applies and set membership is precise. Fuzzy inferencing systems, on the other hand, are based on the properties of fuzzy sets [125] where membership is no longer precise. Instead, an element belongs to a given set with an associated degree of membership.

The ability to model the fuzzy, or imprecise, membership of an element to a set enables inferencing based on linguistic terms. Production rules governing a fuzzy controller can be described using words or simple sentences in natural language as opposed to formal predicate calculus statements. This enables a domain expert, who typically would not have an advanced knowledge of first-order predicate logic, to describe the rules that govern a given system using domain specific linguistic terms which may be better understood.

Section 2.5.1 overviews the theory of fuzzy sets and linguistic variables. Fuzzy controllers, discussed in Section 2.5.2, build on this theory to provide a powerful inferencing engine that can be used to solve control problems based on domain knowledge provided by an expert.

2.5.1 Fuzzy Sets

Fuzzy sets [125] are characterised by a membership function of the form:

$$\mu_A : X \rightarrow [0, 1] \tag{2.50}$$

where $\mu_A(x)$, $\forall x \in X$, indicates the degree, or certainty, that x belongs to the fuzzy set A, and X is known as the universe of discourse. Traditional boolean set membership can be modelled by a membership function, $\mu_A(x)$, which strictly takes on the values 0 or 1.

Table 2.1 defines fuzzy set theoretic operators that are analogues for their traditional set counterparts. Two fuzzy sets are equivalent if and only if their membership functions are identical. A fuzzy set is a superset of another set if and only if it contains all the elements of the other set to at least the same degree of membership. The complement of a set contains the same elements as the original set, but with complimentary degrees of

University of Pretoria etd – Peer, E S (2005)

Table 2.1: Fuzzy Set Theoretic Operators

Operator	**Definition**
Equality	$A = B \Longleftrightarrow \mu_A(x) = \mu_B(x),\ \forall x \in X$
Containment	$A \subset B \Longleftrightarrow \mu_A(x) \leq \mu_B(x),\ \forall x \in X$
Complement	$\mu_{\overline{A}}(x) = 1 - \mu_A(x),\ \forall x \in X$
Intersection	$\mu_{A \cap B} = \min\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or $\mu_{A \cap B} = \mu_A(x)\mu_B(x),\ \forall x \in X$
Union	$\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\},\ \forall x \in X$, or
	$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x),\ \forall x \in X$

membership, so that an element having a high degree of membership has a proportionally low degree of membership to the complement. The intersection operator may be defined as the minimum of the degrees of membership of elements to each set, or it may be defined as the product of the membership functions. The product version is the stronger of the two operators, resulting in lower degrees of membership for the intersection. Similarly, the union may be defined in terms of the maximum degree of membership, or it may be defined algebraically. In the limit, a series of unions cumulatively tends to 1 and a series of intersections tends to 0, irrespective of the degrees of memberships to the individual sets.

Linguistic variables and their associated hedges [126, 127, 128] express words and sentences, in natural language, in terms of fuzzy set memberships. Consider as an example, the concept of a person's age as a linguistic variable. The linguistic variable *age* might take on values such as *young*, *middle aged* and *old*. Each of these values defines a fuzzy set, associated with a membership function that models its semantics. Figure 2.13 illustrates three possible membership functions, defined using Gaussians, for the values *young*, *middle aged* and *old* respectively. Further, hedges such as *very*, *fairly*, *somewhat* and *slightly* may be used to modify a membership function.

Numerous hedges may be defined, with the primary types of hedges given by the following equations:

$$\text{Concentrate}: \quad \mu_{A'}(x) = \mu_A(x)^p \tag{2.51}$$

$$\text{Dilate}: \quad \mu_{A'}(x) = \mu_A(x)^{1/p} \tag{2.52}$$

University of Pretoria etd – Peer, E S (2005)

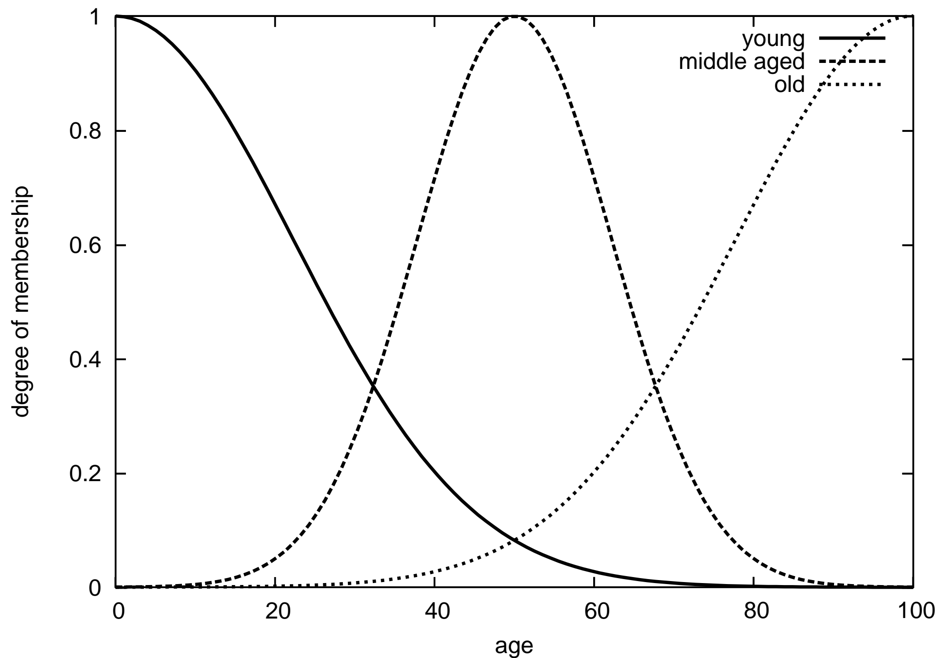


Figure 2.13: Membership Functions for *Age* Linguistic Variable

$$\text{Intensify :} \quad \mu_{A'}(x) = \begin{cases} 2^{p-1}\mu_A(x)^p & \text{if } \mu_A(x) \leq 0.5 \\ 1 - 2^{p-1}(1-\mu_A(x))^p & \text{otherwise} \end{cases} \tag{2.53}$$

$$\text{Blur :} \quad \mu_{A'}(x) = \begin{cases} \sqrt{\mu_A(x)/2} & \text{if } \mu_A(x) \leq 0.5 \\ 1 - \sqrt{(1-\mu_A(x))/2} & \text{otherwise} \end{cases} \tag{2.54}$$

where $p > 1$ may be tuned to control the intensity of the hedges in Equations (2.51) through (2.53). Concentration hedges, corresponding to linguistic terms such as *very*, *greatly* and *decidedly*, create modified membership functions where boundaries are shifted in favour of higher membership values. Dilation hedges have the opposite effect and correspond to terms such as *somewhat*, *sort of* and *fairly*. Terms such as *indeed* and, for higher values of p, *extremely*, correspond to intensification hedges which emphasise contrast. Finally, blurring hedges, corresponding to terms such as *seldom* and *more or less*, perform the opposite of intensification by introducing vagueness.

University of Pretoria etd – Peer, E S (2005)

2.5.2 Fuzzy Controllers

Figure 2.14 outlines a simple architecture for a fuzzy controller [75] consisting of three primary components. First, the condition interface, which is responsible for converting outputs from the system into a fuzzy form, hence the term fuzzifier, utilised by the fuzzy inferencing engine. Next, the engine performs inferencing, based on linguistic rules, to determine an appropriate control action. Finally, the action interface is responsible for interpreting the output of the inferencing process and converting it back into system specific actions through a process known as defuzzification. Thus, a feedback loop is realised where the controller constantly monitors the system while effecting control actions on the system according to its rule base.

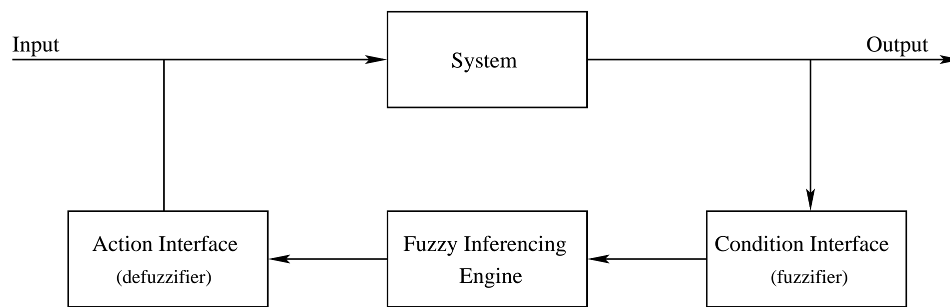


Figure 2.14: Fuzzy Controller Architecture

As a somewhat contrived example, consider a fuzzy system used to control a hypothetical cigarette dispensing machine. Rather than blindly supplying smokers with their selection, this particular machine is designed to wean them off their addiction by carefully limiting their supply of cigarettes. Further, assume that a domain expert, such as a lung specialist, has provided a number of linguistic rules. For example, "If the user is very old and a regular smoker then dispense as many cigarettes as requested." The reasoning behind such rule might be that a heavy smoker who has managed to survive to a ripe old age is likely to die of natural causes long before contracting lung cancer. Other rules might curtail the number of cigarettes dispensed to younger smokers depending on their average intake, or limit the provision to zero for casual smokers.

The dispensing machine provides the controller with two inputs requiring fuzzification, the actual age of the user and the average number of cigarettes consumed on a daily basis. Fuzzification entails identifying the fuzzy sets used by the inferencing engine and

University of Pretoria etd – Peer, E S (2005)

calculating the degrees of membership to each of these sets given the inputs. Continuing with the example rule, according to Figure 2.13, the membership function for the set corresponding to the linguistic term *very old* is given by:

$$\begin{aligned}\mu_{[\text{very old}]}(x) &= \mu_{\text{old}}(x)^2 \\ &= \begin{cases} \left(e^{-(x-100)^2/1000}\right)^2 & \text{if } x \leq 100 \\ 1 & \text{otherwise} \end{cases}\end{aligned} \tag{2.55}$$

where x is the actual age of the user and the concentration hedge for the term *very* is assumed to be implemented with $p = 2$. A membership function for $\mu_{[\text{regular smoker}]}$ can be defined in a similar fashion.

After fuzzifying the inputs, the next step is to perform inferencing using the fuzzy rule base. Typically, the rule base is made up of a list of rules of the form:

$$\text{if } \textit{antecedent} \longrightarrow \textit{consequent} \tag{2.56}$$

where the *antecedent* consists of one or more fuzzy sets combined using the operators in Table 2.1 to form a logical expression. In the case of a Mamdani [75] controller, the *consequent* consists of a single target fuzzy set. The value of the antecedent, also known as the firing strength of the rule, determines the degree of membership to the target set in the consequent. A Takagi-Sugeno [110] controller, on the other hand, permits higher order consequents.

The antecedent for the example sentence presented earlier may be calculated as either:

$$\mu_{[\text{very old}]}(x) \cap \mu_{[\text{regular smoker}]}(y) = \min\{\mu_{[\text{very old}]}(x), \mu_{[\text{regular smoker}]}(y)\} \tag{2.57}$$

or, the product:

$$\mu_{[\text{very old}]}(x) \cap \mu_{[\text{regular smoker}]}(y) = \mu_{[\text{very old}]}(x)\mu_{[\text{regular smoker}]}(y) \tag{2.58}$$

depending on the choice of intersection operator, where x and y are the age and average daily cigarette consumption respectively. The firing strengths for the remaining antecedents in the rule base are calculated in a similar fashion.

The defuzzification processes is performed for each output linguistic variable to determine a single non-fuzzy, or crisp, value to feed back to the system. In the example rule, the linguistic variable associated with the cigarette limit has a consequent of *unlimited*,

University of Pretoria etd – Peer, E S (2005)

however, this must still be combined in a sensible way with the consequents of any other rules pertaining to the same linguistic variable.

Various defuzzification strategies may be employed, the height of the centroid under the composite area defined by the chosen strategy is used as the crisp action result:

- **max-min strategy:** Only the membership function of the consequent associated with the rule having the highest firing strength is used.
- **averaging strategy:** All membership functions pertaining to the linguistic variable in question are clipped at the average firing strength of the combined rules.
- **root-sum-square strategy:** All membership functions pertaining to the linguistic variable in question are scaled to the firing strengths of their respective rules.
- **clipped centre of gravity:** All membership functions pertaining to the linguistic variable in question are clipped at the firing strengths of their respective rules.

Thus, all the consequents corresponding to a given linguistic variable are combined, based on the chosen defuzzification strategy, into a single crisp value. At the one extreme, the max-min strategy only takes into account the most dominant rule, while the averaging strategy dilutes the result, giving no preference to rules with higher firing strength. Further, it is possible to bias the rules, by scaling their firing strengths, based on the confidence placed on a given rule by a human expert.

2.6 Other Paradigms

One specific example of a relatively new CI paradigm is the Artificial Immune System (AIS) [24], which is a computational pattern recognition technique, based on how white blood cells in the human immune system detect pathogens that do not belong to the body. Instead of building an explicit model of the available training data, an AIS builds an implicit classifier that models everything else but the training data, making it suited to detecting anomalous behaviour in systems. Thus, an AIS is well suited for applications in anti-virus software, intrusion detection systems and fraud detection in the financial sector.

University of Pretoria etd – Peer, E S (2005)

Further, fields such as Artificial Life (ALife), robotics (especially multi-agent systems) and bioinformatics are application areas for CI techniques. Alternatively, it can be argued that those fields are a breeding ground for tomorrow's CI ideas.

For example, evolutionary computing techniques have been successfully employed in bioinformatics to decipher genetic sequences [35]. Hand in hand with that comes a deeper understanding of the biological evolutionary process and improved evolutionary algorithms.

As another example, consider RoboCup[1], a project with a very ambitious goal. The challenge is to produce a team of autonomous humanoid robots that will be able to beat the human world championship team in soccer by the year 2050. This is obviously an immense undertaking that will require drawing on many disciplines. The mechanical engineering aspects are only one of the challenges standing in the way of meeting this goal. Controlling the robots is quite another. Swarm robotics [6, 99], an extension of swarm intelligence into robotics, is a new paradigm in CI that may hold some of the answers. In the mean time, simulated RoboCup challenges, which are held annually, will have to suffice.

2.7 Hybrid Approaches

Attempting to produce an exhaustive list of all the possible hybrid approaches here is certainly an exercise in futility. There are, simply stated, so many ways in which different CI techniques can be combined that any attempt to survey them would probably require an entire dissertation dedicated to that task alone. Indeed, hybrid approaches need not even limit themselves to combining techniques drawn from the CI discipline alone, making the possibilities virtually endless. Instead, the purpose of this section is to emphasise the existence of hybrids, by means of a few examples, and to highlight the importance of a flexible software framework which enables composing various techniques together in new and interesting ways.

As a first example, consider the PSO, discussed in Section 2.4.1. One hybridised approach, dubbed the Dissipative PSO (DPSO) [122], builds on concepts borrowed from thermodynamics. The designers of the DPSO noted that the self organising nature of the PSO, where particles follow an irreversible process towards higher fitness, ultimately

[1] http://www.robocup.org

University of Pretoria etd – Peer, E S (2005)

lacks the capability for sustainable development. By introducing negative entropy into the algorithm and operating as a dissipative structure, the DPSO is able to maintain swarm diversity and improve the quality of solutions found by the search. Now, while it could be argued that the DPSO is not a true hybrid but rather a relatively simple extension of the PSO, the relevant issue is that a software implementation should, as far as possible, reuse an existing implementation of the PSO and simply compose it with something that implements the dissipative capability.

Another method to hybridise the PSO is to update the positions of the best performing particles using a different optimisation process. Consider the velocity update in Equation (2.41), the best particles in their respective neighbourhoods will have $\mathbf{x} = \mathbf{y} = \hat{\mathbf{y}}$, resulting in zero cognitive and social components. Eventually, the velocity components will also degrade to zero, since $0 \leq w < 1$, and these particles will stop moving. Further, it is possible for the rest of the particles to collapse onto these positions too, resulting in stagnation of the entire swarm. The Guaranteed Convergence PSO (GCPSO) [114, 113] replaces the velocity update for the neighbourhood best particles with a modified unimodal optimiser [103], in effect creating a hybrid of the two. Properties of the GCPSO include rapid convergence and a guarantee to at least converge onto a locally optimum solution. Once again, a software implementation should make provision for this kind of hybrid, perhaps by having a pluggable optimisation process for the neighbourhood best, or indeed any particle. This kind of flexibility would enable the optimisation process for any particle to be replaced by say, gradient descent, LeapFrog [102], an evolutionary algorithm, or perhaps even another PSO to create a hierarchical PSO-PSO hybrid. Further, it may be desirable to simultaneously compose GCPSO and DPSO into yet another hybrid.

As hinted in Sections 2.1.3 and 2.2.1, neural networks present another opportunity for hybridisation. By representing network weights as a single vector and the SSE over the training set as an objective function, neural network training can be re-framed as an optimisation problem. This opens the door for many hybrids, including using GAs, EP, ES, cultural evolution or PSOs to train neural networks. Again, a software implementation should enable neural network training using any optimisation algorithm in a flexible fashion.

One specific hybrid example, which spans multiple paradigms, is Blondie 24 [34]. Blondie 24 is an advanced game playing framework with the ability to understand and

University of Pretoria etd – Peer, E S (2005)

develop strategies for a game given only its rules as prior knowledge. The framework draws on three paradigms: game theory, neural networks and evolutionary computing. The approach involves evaluating a traditional game tree [85] using a neural network as an evaluation function. In order to find the optimal network for the task, Blondie 24 employs a competitive coevolutionary approach to evaluate network against network. Over time, neural networks evolve that are better able to evaluate the game state and as a result become stronger players. This approach has been taken one step further [79, 40] by extending the coevolutionary approach to particle swarms, producing a four way game tree, neural network, coevolution, PSO hybrid. Designing software flexible enough to support such hybrids is a challenging task.

Other hybrid approaches include fuzzy neural networks [88, 129], a breeding PSO that leverages evolutionary crossover [74] and evolutionary processes for learning rules for fuzzy controllers [22].

2.8 Software Requirements

Section 2.7 illustrated the importance of a flexible software framework. It should be possible to reuse and compose various algorithms in different ways with a minimum amount of recoding. Ideally, any permutation should be made possible by merely changing the configuration of the system at runtime.

Section 2.1 demonstrated that most problem classes can be re-framed as optimisation problems. For this reason, any optimisation algorithm should be able to operate on any problem which can be cast as an optimisation problem, as defined in Section 2.1.1.

It is tempting to make the next step and simply treat all problems as optimisation problems, that way the interface between algorithms and problems is reduced to a single set of interactions. To see why this is a poor idea, consider what the interface for an optimisation problem might look like. Optimisation algorithms, such as the PSO or a GA, require only two pieces of information from the problem. Firstly, they need to know the domain of the problem. Secondly, and most importantly, they need to know the fitness of a potential solution to the problem. Any more information would not be used by such optimisation algorithms. Indeed, many optimisation problems, such as function minimisation, simply cannot provide any more information either. Thus, an optimisation problem must be characterised by an interface that supplies the domain of the problem

University of Pretoria etd – Peer, E S (2005)

and the fitness of a given solution within that domain.

From an implementation perspective, contrast the functioning of a generic optimisation algorithm, which only needs to query the fitness of potential solutions to a problem, with a feed forward neural network. The neural network needs access to a set of training patterns with their associated inputs and targets. Thus, the neural network requires more information from the problem domain than a generic optimisation algorithm, which is satisfied with only having access to an objective function. Therefore, the software should have different interfaces for problems that make different information available to algorithms according to their context. The various algorithms, in turn, should be able to be applied to whatever types of problems they support, also by means of configuration at runtime. Further, any problem that can be represented as another type of problem, via some transformation such as those discussed in Section 2.1, should expose an interface to do so. For example, a TSP should expose an optimisation problem interface in addition to its more natural interface, which would expose a graph topology necessary for an algorithm such as ACO.

Stopping conditions are another important element of algorithms that should be handled in a pluggable way. All algorithms presented in this chapter loop until some stopping condition is met. Those stopping criteria exist independently of the particular algorithm. Any algorithm can have as a stopping criterion a maximum number of iterations. Optimisation algorithms may have as a stopping criterion a maximum number of evaluations of the objective function. Particle swarms may have a stopping criterion based on a minimum swarm diameter. Once again, stopping criteria should be configurable at runtime for any algorithm.

Finally, since the software will be used for scientific research it is important to be able to measure certain properties during the execution of any algorithm. Some of these properties may be dependent on the specific problem or algorithm being used, however, they should still be implemented in a reusable fashion externally to the algorithm. Measurements should not clutter the implementation of algorithms and should not even be present if they are not used, for example, if the software is deployed in a specific non-research application that has no need for measurements.

Creating a flexible software design is a challenging task. The next chapter presents patterns which are invaluable aids for creating such designs.

University of Pretoria etd – Peer, E S (2005)

Chapter 3

Design Patterns

> *"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."*
> *— Douglas Adams*

Design patterns succinctly encapsulate the knowledge of experienced programmers by specifying proven solutions to commonly recurring software design scenarios. Patterns are not specifically invented or designed, rather, they are discovered by observing best practices and recurring design solutions that have proven to be useful, efficient, and extensible in existing software.

The Gang of Four [41], or GoF as the pioneers of the field are usually referred to, presented a catalogue identifying core design patterns which apply to Object Oriented Programming (OOP) in general. In addition, catalogues have since been compiled for the following:

- high level architectural patterns [19, 39];
- distributed systems and concurrency patterns [98];
- database programming patterns [86];
- language or framework specific patterns [4, 80].

Catalogues of design patterns enable software developers to draw upon documented experience instead of reinventing the wheel. Good design is difficult to accomplish, particularly for novice programmers, usually requiring a number of redesign iterations.

University of Pretoria etd – Peer, E S (2005)

Pattern catalogues consist of mature and successful designs that have been frequently found in software written by experienced programmers. In this way patterns capture the experience of experts, providing it in a concise and easy to digest form.

An entry in a design pattern catalogue consists of four essential components. Firstly, a short and descriptive pattern name. These names define a vocabulary for communicating about entire designs at a higher level of abstraction. Secondly, an outline of the problem and its context together specify when it is appropriate to apply the pattern. The most important element of any pattern is obviously the solution to this problem. Solutions are described in abstract terms, along with class structure diagrams, that can be applied as a template in many different concrete situations. Sample code demonstrating the usage of the pattern is often presented. Finally, the impact and known consequences of the pattern are listed.

Software implementing design patterns does not only benefit from the expert experience derived from the patterns. The patterns themselves serve as documentation for that software too. Scholars of design patterns should be able to understand the design of such software with little more documentation than a reference to the applicable pattern and a brief explanation of any unusual implementation details. Furthermore, programmers unfamiliar with design patterns can simply refer to the catalogue where the design is discussed in detail. The self documenting nature of code that uses patterns is an important reason for patterns being discussed in this work, otherwise the patterns that have been used in the implementation, although very useful in ensuring good design, may just as well have been considered an irrelevant implementation detail.

This chapter summarises those GoF patterns that are applicable to CILib and CiClops. The patterns are separated, based on their purpose, into three distinct categories: creational patterns, presented in Section 3.1; structural patterns, presented in Section 3.2; and behavioural patterns, presented in Section 3.3. The intention, describing the primary purpose of a pattern, is quoted directly from the GoF catalogue [41] as an introduction to each pattern. The patterns are summarised in a less rigid form than the GoF catalogue without many examples. Chapters 6 and 7 will serve as adequate examples where the implementations of these patterns are discussed. High level architectural and framework specific patterns are implicitly covered, as required, when platforms such as Java 2 Enterprise Edition (J2EE) are discussed in Chapter 5. This chapter concludes with a short discussion in Section 3.4

University of Pretoria etd – Peer, E S (2005)

3.1 Creational Patterns

The common theme amongst the creational patterns is delegating the details of object creation in a particular system, or client, to other classes external to the client that can vary independently. That is, there is a decoupling between the use of objects and their creation.

Section 3.1.1 presents the *Abstract Factory* pattern, where the instantiation of objects is delegated to a polymorphic interface. The *Builder* pattern, in Section 3.1.2, abstracts the process of instantiating a complex set of objects into a reusable unit that can be used to construct different representations using the same build process. Section 3.1.3 discusses a pattern for creating objects by cloning existing prototype objects. Finally, the *Singleton* pattern, in Section 3.1.4, limits the instances of a given class.

3.1.1 Abstract Factory

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes" — GoF

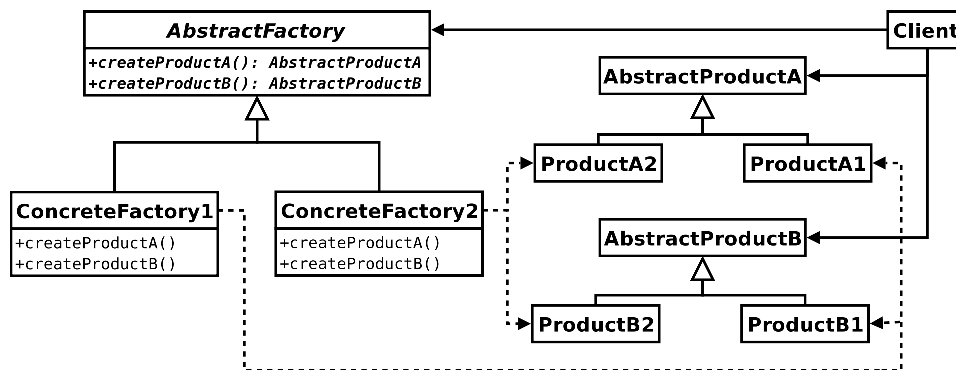


Figure 3.1: Abstract Factory

Figure 3.1 illustrates the design of the *Abstract Factory* pattern. The core participant in the pattern is the abstract factory interface which defines the contract that its client uses to instantiate objects. The most important aspect of the pattern is that the client is never exposed to the implementation details, including the class names, of the concrete factories or the classes that they create. Each concrete factory is responsible for producing its own

University of Pretoria etd – Peer, E S (2005)

family of concrete products with the only requirement being that the abstract interfaces are satisfied. Thus, if the client is written to conform to the abstract interfaces then the concrete factories, and by extension the products that they produce, may be interchanged without requiring changes to the client.

The decoupling of a system from how its products are created provides immense flexibility, to the extent that the entire behaviour of the system can be altered by simply changing the factory used to create the objects that it uses. Furthermore, dependencies between a family of products can be enforced, since a single concrete factory is responsible for all the different products at any given time. Unfortunately, a drawback of the design is that adding new products is difficult, since it entails a modification of the abstract factory interface. Such an interface change translates into changes to all existing concrete factory implementations to support the new product which, in turn, is likely to require new product implementations to be defined as well.

3.1.2 Builder

"Separate the construction of a complex object from its representation so that the same construction process can create different representations" — GoF

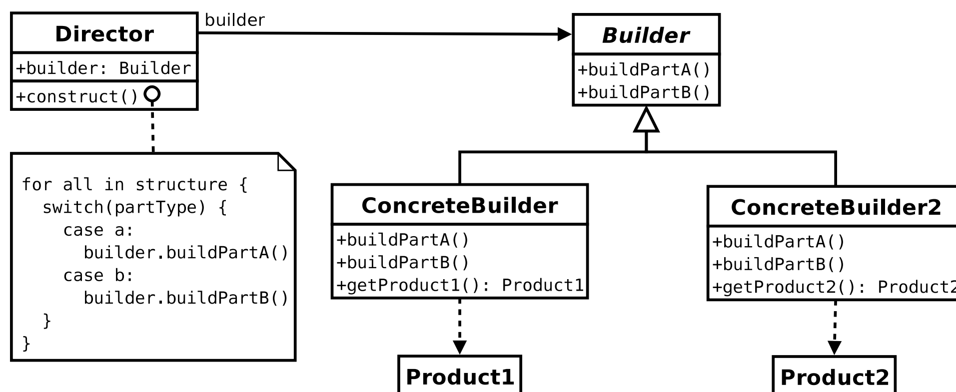


Figure 3.2: Builder

The *Builder* pattern, depicted in Figure 3.2, assembles complex objects in a piecemeal fashion, building them part by part. A director class controls the construction process while delegating the creation and assembly of parts of the product to an abstract builder

University of Pretoria etd – Peer, E S (2005)

interface. Thus, a concrete builder has jurisdiction over the implementation details of the parts as well as how they are assembled to create a larger complex product. Typically, the functioning of the director is dictated by the traversal of some data structure or document. The builder interface exposes the set of operations that may be utilised by a director to construct a product according the structure it traverses.

Products produced by a given concrete builder implementation need not conform to any given interface. Thus, it is possible for two different concrete builders to create two very different products using the same construction process, as specified by the director. Alternatively, different directors may use the same builder interface permitting different structures to be rendered into the same product representation. In addition, the director provides finer control over the construction process than the *Abstract Factory* which creates each of its products in a single shot.

3.1.3 Prototype

> *"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype" — GoF*

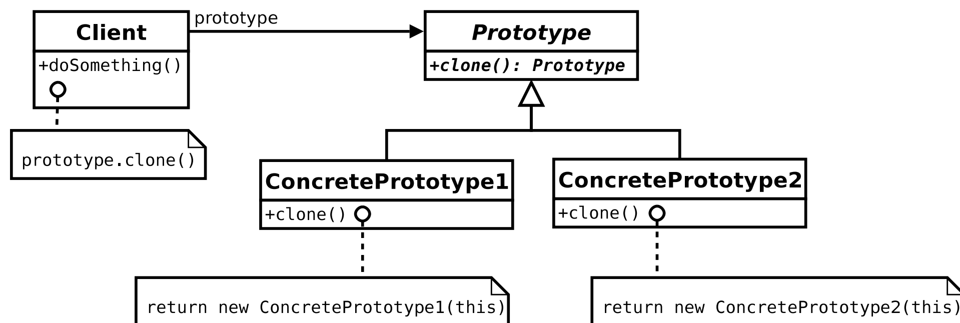


Figure 3.3: Prototype

The *Prototype* pattern creates new objects by copying, or cloning, existing objects. Importantly, the client making a clone of an object need not know the type of object it is dealing with, only the fact that the object implements the prototype interface. The responsibility of making the copy falls on the object being cloned, as shown in Figure 3.3.

One of the key benefits of prototypes is that they enable a client to instantiate objects that have been configured at run time. That is, objects with different run time state or

University of Pretoria etd – Peer, E S (2005)

object structures that have been composed together in different ways at run time may conceptually be considered to be instances of different classes. The *Prototype* allows these different run time configurations of objects to be treated as new classes that can be instantiated like any other class. Thus, an application can be configured with new classes dynamically.

When used in conjunction with the *Abstract Factory*, the *Prototype* pattern can mitigate the need to create concrete factories for every product. Instead, a single factory can simply be configured with different prototype instances as products.

The clone operation typically performs a deep copy which has an obvious caveat pertaining to circular references. Prototypes containing any circular references need to take appropriate measures to prevent infinite looping.

3.1.4 Singleton

> *"Ensure a class only has one instance, and provide a global point of access to it" — GoF*

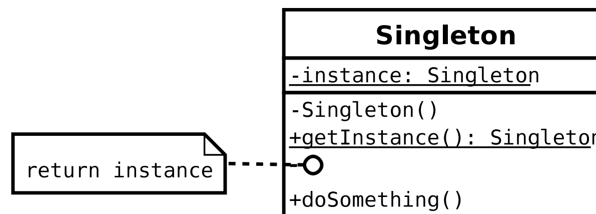


Figure 3.4: Singleton

The *Singleton* pattern, illustrated in Figure 3.4, is characterised by three properties. Firstly, any constructors are inaccessible so that clients can not arbitrarily create instances of the class. Secondly, the only existing instance is a static field, also known as a class scoped field, which is also not directly accessible to clients. Finally, a publicly accessible static method provides clients with access to the single instance. The single instance may be statically initialised or it may be initialised in a lazy fashion by the public accessor the first time it is called.

The purpose of the *Singleton* is to prevent a shared object from being instantiated by multiple clients. Limiting the number of instances not only saves memory, but more

University of Pretoria etd – Peer, E S (2005)

importantly, it prevents difficult to detect programming errors from occurring, where an object which is supposed to be shared is not being shared properly. Further, a singleton can be used as a namespace to store global application context cleanly, without resorting to global variables. Moreover, instead of restricting clients to a single instance, it is trivial to extend the pattern so that the implementation maintains a limited pool of objects for applications that require it.

3.2 Structural Patterns

Structural patterns describe methods to compose classes to form larger useful structures. That is, they illustrate flexible methods of interaction between classes by specifying how classes should be combined and used together.

The *Adapter* pattern, in Section 3.2.1, demonstrates how incompatible classes can be made compatible and used together. Section 3.2.2, the *Composite*, discusses a pattern that enables hierarchies of objects and individual objects to be treated in a uniform fashion. The *Decorator* pattern, which can be used to dynamically associate additional behaviour with objects, is discussed in Section 3.2.3. Complex systems of classes can be simplified into a single interface using the *Facade* in Section 3.2.4. Finally, the *Proxy* pattern provides a way to facilitate or control access to the objects which it stands in for.

3.2.1 Adapter

> *"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces" — GoF*

Figure 3.5 illustrates the most common form of the *Adapter* pattern, particularly in languages that only support single inheritance. The adapter class maintains a reference to the object which it is adapting, the adaptee, while conforming to the target interface expected by the client. Another form of adapter inherits both the target and adaptee interfaces which may not always be possible in languages that do not support multiple inheritance. The multiple inheritance version has the advantage of being able to triv-

University of Pretoria etd – Peer, E S (2005)

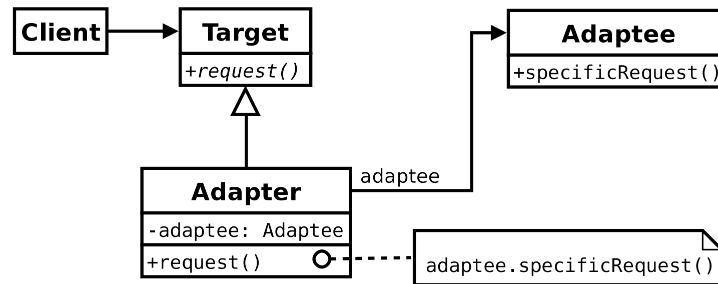


Figure 3.5: Adapter

ially override any operations belonging to the adaptee, if necessary, whereas the version presented here requires an auxiliary class to override adaptee operations.

The amount of work that needs to be done by the adapter is application specific and depends on how much the target interface differs from that of the adaptee. In some cases, particularly when reusing legacy classes in a new framework, all that may be required is changing the the name of an operation or converting the types of its arguments. In more extreme cases, the interface may be totally different, requiring more work to make the adaptee conform to the target interface expected in the context of the client.

3.2.2 Composite

> *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and the compositions of objects uniformly." — GoF*

The *Composite* pattern, depicted in Figure 3.6, represents hierarchical structures of objects in such a way that clients can treat the individual objects in exactly the same way as they treat the entire composite. Operations on leaf nodes in a composite structure behave according to the type of node that the operation is being executed on, whereas composite nodes typically delegate the requested operation to each of their child nodes. Hierarchies can be built recursively, since a composite node is itself a component which in turn contains components.

The primary benefit of the *Composite* pattern is also its weakness. The fact that clients should not need to differentiate between operations on leaf nodes and operations on composite nodes means that the root component interface needs to support all of the

University of Pretoria etd – Peer, E S (2005)

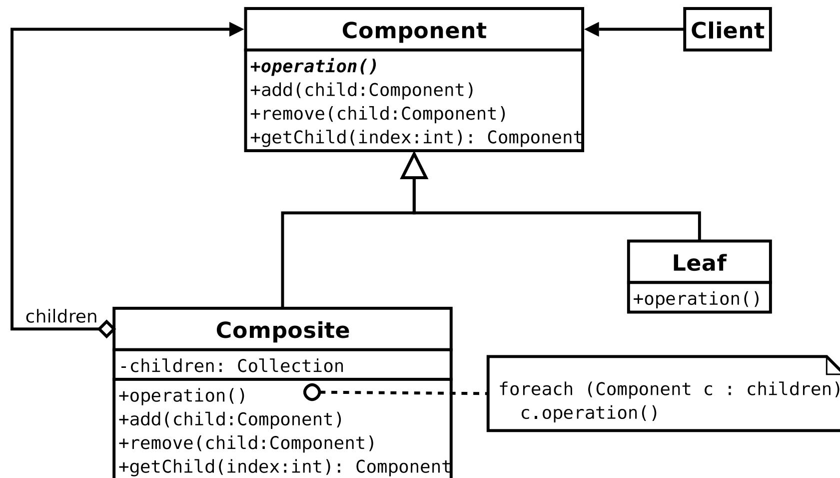


Figure 3.6: Composite

operations supported by any of the components, thus reducing type safety. For example, operations for maintaining the child nodes of a composite do not usually apply to leaf nodes, so these operations usually have an empty implementation in the root interface. Similarly, there may be operations specific to leaf nodes that do not make sense for composite nodes, or even other types of leaf node for that matter. Thus, even though all components must implement the same component interface by virtue of inheriting from it, some of them may have unexpected or default behaviours when certain operations are called.

3.2.3 Decorator

> *"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." — GoF*

Structurally, the *Decorator* pattern, in Figure 3.7, and the *Adapter* presented in Section 3.2.1 are similar. Both delegate operations prescribed by a target interface to another class which they reference, or wrap. In the case of the *Adapter*, the adaptee is an arbitrary class that must be made to conform to a target interface. The *Decorator*, however, delegates operations specified by the component interface to another class conforming to that same interface with the purpose of adding responsibilities to the original component, not to make the already compatible interfaces compatible with each other.

University of Pretoria etd – Peer, E S (2005)

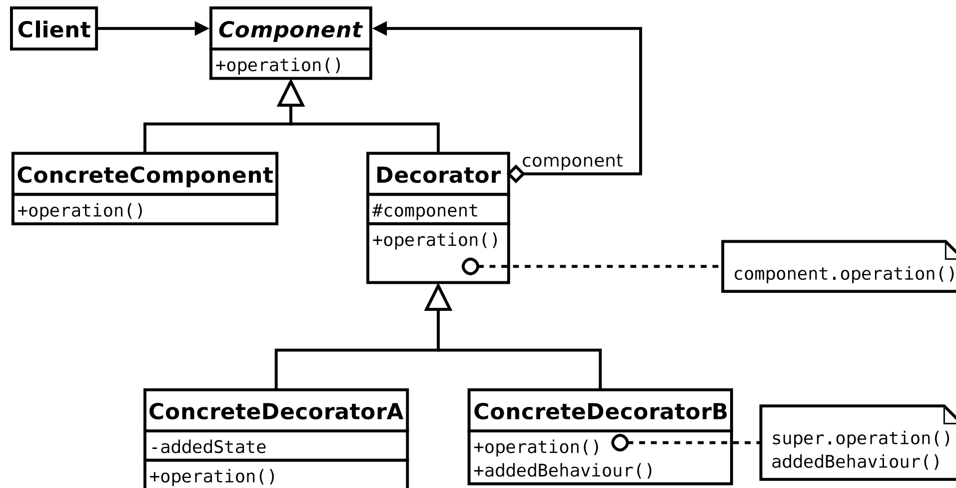


Figure 3.7: Decorator

Nevertheless, throughout design pattern literature, both the *Adapter* and the *Decorator* have been referred to by the same alternate name, namely the *Wrapper* pattern, probably owing to the fact that both have a similar structure.

Concrete decorator classes add a combination of additional state and behaviour to a target class without changing the interface that is exposed to the client. Typically, the base decorator class is simply an identity mapping for the operations defined by the component interface. That way, a concrete decorator need only override the operations necessary to achieve its goal. The primary benefit of the decorator is that these additional responsibilities can be dynamically added and removed from a component at run time, whereas extending the responsibilities of a class through normal inheritance is fixed at compile time and as such is less flexible. Concrete components need not implement seldom used functionality that can be added by decorators on an as needed basis. Unfortunately, decorators are not truly transparent, since clients cannot rely on the equivalence of decorators and their components based on their references.

3.2.4 Facade

> *"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use"*
> — *GoF*

University of Pretoria etd – Peer, E S (2005)

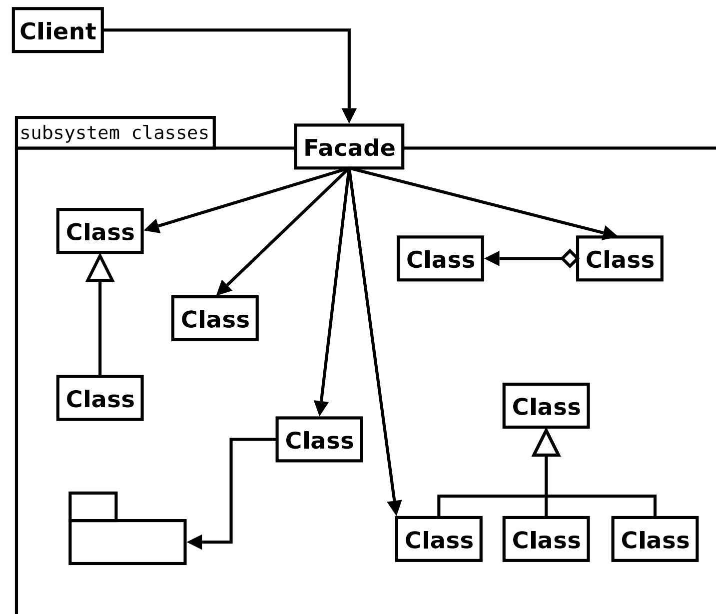


Figure 3.8: Facade

The *Facade* pattern, illustrated in Figure 3.8, decouples a complex system from its clients by providing a high level interface to access the system in a simplified way. The extra flexibility and extensibility that other design patterns bring to the table often has the net result of making a system of classes more complex. For example, a client may be able to configure a well designed system to better suit its needs by extending some of the classes that make up that system. The *Facade* provides a mechanism to counteract some of this complexity in the cases when a client does not need to alter the default behaviour of a system.

Structurally, the *Facade* is also similar to the *Adapter*, presented in Section 3.2.1, except that the facade typically maintains references to many objects within the system instead of only adapting the interface for a single class. In effect, the facade adapts the interfaces provided by an entire system and presents them as a single simplified interface to clients.

The most important feature, with respect to making a system more maintainable, is that the facade decouples the client from the system so that changes to the internals of the system do not affect clients. Further, the facade interface may be polymorphic so that the entire system implementation can be switched without the client's knowledge

University of Pretoria etd – Peer, E S (2005)

by simply changing the instance of the facade being used. The decoupling provided by the facade can also be extended to the interface between between different layers in a multi-layer framework. The refined interface reduces the communication between layers and thus reduces their dependency on one another while improving performance, particularly if the layers are implemented in different address spaces.

While the facade provides a simpler interface to the system, there is typically nothing preventing a client from accessing system classes directly. In fact, the facade interface may require the client to do so by accepting as arguments or returning system specific classes. Further, the client may need to use some complex features of the system that the facade does not provide access to. Obviously, the more that a client directly relies on the system classes, the tighter the coupling and harder it is to modify the system without affecting its clients.

3.2.5 Proxy

> *"Provide a surrogate or placeholder [sic] for another object to control access to it." — GoF*

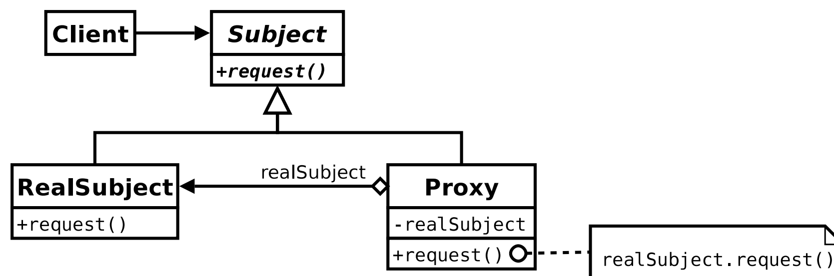


Figure 3.9: Proxy

According to Figure 3.9, the *Proxy* pattern is very similar to the *Decorator*, presented in Section 3.2.3. In fact, in certain cases, a proxy can also be considered to be attaching additional responsibilities to the object for which it stands proxy. The difference lies in the intent of the pattern, even though they are structurally very similar. The responsibilities associated with a proxy are typically more behind the scenes or house-keeping in nature than actually adding application specific behaviour to objects.

University of Pretoria etd – Peer, E S (2005)

There are four primary types of proxy. The first, a remote proxy, is a local representative that provides access to a complementary object in another address space. An example of this is a stub object, typically automatically generated, that implements calls to the same object on a remote machine via Remote Procedure Call (RPC). Secondly, virtual proxies are place holders, used to create and destroy their objects on demand, that are usually used to optimise memory or initial start up cost. Third, protection proxies prevent unauthorised client access to methods by implementing access control before delegating the method call to the real subject. Finally, smart references can be used to implement reference counting, locking or copy-on-write semantics.

3.3 Behavioural Patterns

Behavioural patterns model the flow of control and algorithmic interaction between objects. They specify how responsibility should be assigned to various classes to achieve communication between objects in the most flexible manner.

The *Interpreter* pattern, in Section 3.3.1, describes a method to represent a grammar as objects and use those objects to interpret the language. Section 3.3.2 discusses the well known *Iterator* pattern which specifies how objects in a collection should be traversed. Section 3.3.3 defines the *Observer* pattern which implements a flexible event model. The *Strategy* pattern, in Section 3.3.4, decouples a client from the algorithms it uses so that the algorithms can be varied independently. The *Template Method* pattern, discussed in Section 3.3.5, permits an algorithm to be defined in terms of abstract operations that are provided by subclasses. Finally, operations on collections or object structures can be encapsulated using the *Visitor* pattern, as discussed in Section 3.3.6.

3.3.1 Interpreter

> *"Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."*
> *— GoF*

Figure 3.10 shows the abstract structure of the *Interpreter* pattern, used to interpret sentences in a language defined by a given grammar. The dynamic, or run time, structure of the abstract syntax tree reflects a sentence in the language. Terminals in the language

University of Pretoria etd – Peer, E S (2005)

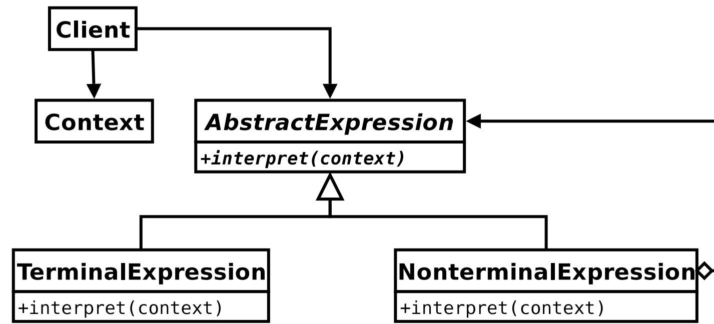


Figure 3.10: Interpreter

are represented by leaf nodes while non-terminals are represented by internal tree nodes. For arithmetic expressions, a separate non-terminal class would be defined for each of the arithmetic operators, while a single terminal expression class would suffice for representing constants. The value of the expression is then interpreted by simply calling the interpret method at the base of the tree, which is recursively propagated down the tree. Each operator is responsible for providing its own interpretation. For example, the interpret operation for an addition node would simply add the results of calling interpret for each of its children. The context is used to store global information, such as the current position in the sentence being interpreted.

The *Interpreter* pattern makes implementing and extending the grammar easy, since classes that represent the grammar have a one-to-one correspondence with its rules. Representing large grammars, however, requires many classes which becomes difficult to maintain. In addition, supporting a new interpretation of the grammar requires adding an operation to each of the expression classes which can become unwieldy if there are too many classes. Also, the *Interpreter* pattern does not address the process of parsing the language into its hierarchical representation, for which a traditional recursive descent or table-driver parser may be used.

3.3.2 Iterator

> *"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." – GoF*

The *Iterator* pattern, demonstrated in Figure 3.11, provides a method to access elements

University of Pretoria etd – Peer, E S (2005)

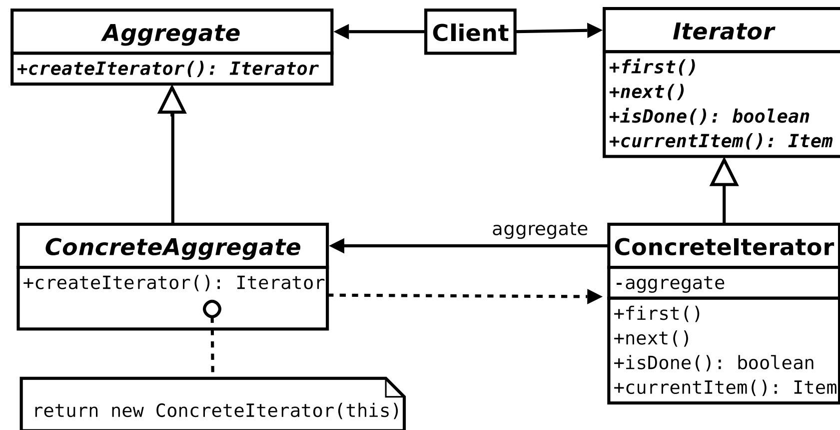


Figure 3.11: Iterator

of an aggregate object, or container, without exposing the client to the internal representation of the aggregate. The client obtains a reference to an iterator by calling an operation to create an iterator, a factory method, provided by the aggregate's interface. This operation returns an iterator that is specific to the concrete aggregate but which supports a well defined interface for performing the iteration. The iterator is responsible for keeping track of where it is in the traversal of the aggregate while providing operations for controlling the traversal. Using the iterator interface, the client can move the iterator to the start of the traversal, obtain the current element, move the iterator to the next element and determine whether there are any more elements left in the traversal. As long as all aggregates conform to the same interface, clients can access their elements in a uniform way.

The most important feature of the iterator is that it provides a standard mechanism for traversing aggregate structures. The interfaces of aggregates are kept clean, since new kinds of traversals can be implemented by simply replacing the iterator. Further, more than one traversal can be pending on the same aggregate because the iterator, and not the aggregate, is responsible for recording the state of the traversal.

University of Pretoria etd – Peer, E S (2005)

3.3.3 Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
— GoF

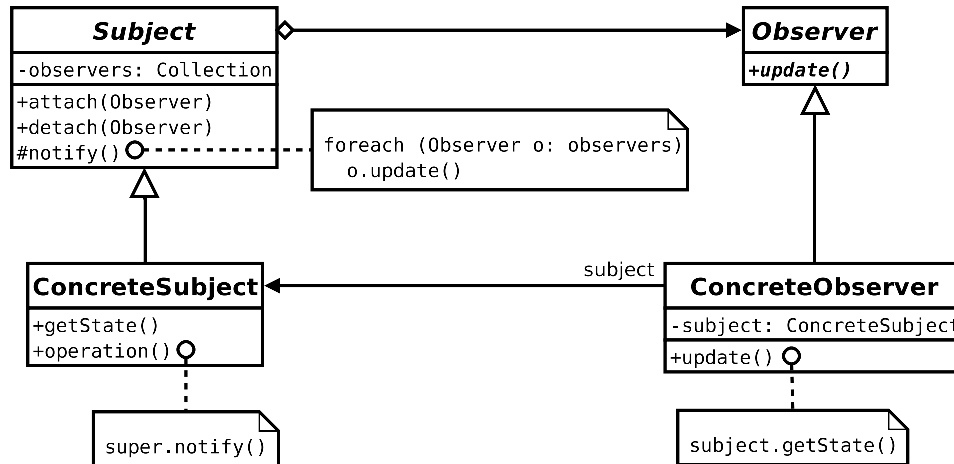


Figure 3.12: Observer

The *Observer* pattern, illustrated in Figure 3.12, models the dependency between a subject and its observers. Any number of observers may subscribe, by means of the attach operation, to be notified whenever the state of the subject changes. After detaching from a subject, an observer will no longer be notified of events. Upon being notified that the state of the subject has changed, an observer may query the state of the subject and take any appropriate actions.

The *Observer* promotes a very loose coupling between a subject and its observers. A subject knows nothing about its observers beyond that they conform to the observer interface. The observer interface presented here is fairly primitive, in that it does not provide any information about the change in state, other than the fact that some state change did occur on some subject. This means that an observer may have to expend considerable effort to determine exactly what state changed. A protocol that is more specific about any state changes would alleviate this problem. In addition, a single observer cannot differentiate between events from multiple subjects. Fortunately, the observer interface can be trivially extended to include a reference to the subject that

University of Pretoria etd – Peer, E S (2005)

originated the event, making many-to-many dependencies possible. Finally, observers have no knowledge about other observers attached to the same subject. This means they are blind to the cost of causing changes to the subject, which may cascade into more changes by other observers.

3.3.4 Strategy

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." — GoF*

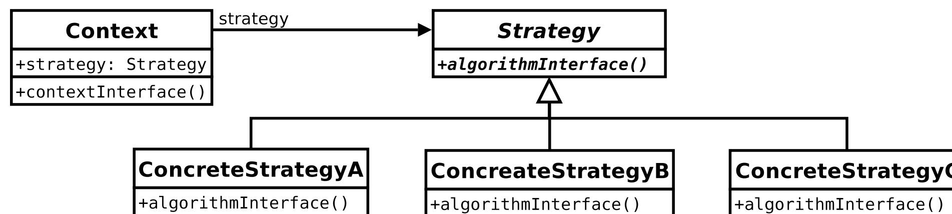


Figure 3.13: Strategy

Figure 3.13 shows the structure of the *Strategy* pattern. At first glance, it simply looks like a polymorphic class that implements multiple behaviours. The importance of the pattern, however, lies in the fact that it is the strategy interface which is polymorphic and not the context class itself. The context, which plays the role of the client, delegates the responsibility for a part of its implementation to an external strategy instance. Subclassing the context directly to provide the different behaviours would result in a less flexible design. By encapsulating the behaviour into a strategy, the context is simplified and different behaviours can be switched dynamically at run time. Also, the context can depend on multiple strategies, for different parts of its operation, simultaneously, which would be impossible to support by directly subclassing the context. For example, a client may rely on one hierarchy of strategies for one part of its implementation while maintaining an additional reference to another hierarchy of strategies for another. Subclassing the context directly would require a new subclass for each combination of the different strategies that can be independently interchanged.

University of Pretoria etd – Peer, E S (2005)

Another benefit of factoring the strategies into a separate hierarchy is that common functionality amongst a family of algorithms can be shared at the root of the strategy hierarchy without cluttering the context. Conditional statements in a client are prime candidates for factoring into a strategy, each branch is simply implemented as an additional concrete strategy, improving flexibility at the cost of increasing the number of classes in the system. The algorithm interface must provide access to the context data needed by any of the concrete strategies, which may create additional overheads for strategies requiring less context data. One possibility is to pass the context itself to the strategy and allow the strategy to query it directly.

3.3.5 Template Method

> *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." — GoF*

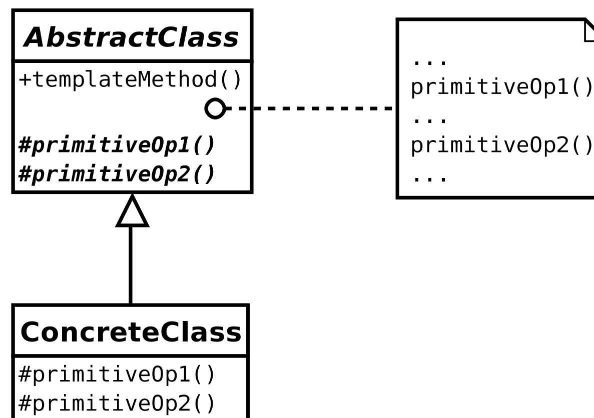


Figure 3.14: Template Method

The *Template Method* pattern, depicted in Figure 3.14, specifies the invariant parts of an algorithm in terms of primitive operations that may be overridden by subclasses. Primitive operations are usually abstract methods, however, they may also be empty methods or have default behaviours creating optional hooks that clients may choose to customise through subclassing. If any of the primitive operations are abstract then the template method is said to implement an abstract algorithm.

University of Pretoria etd – Peer, E S (2005)

The template method, particularly if it cannot be overridden, fixes a specific set of operations and their ordering for subclasses, promoting code reuse. Often, a subclass needs to perform some additional processing before or after a method in its parent class is called. A template method with an appropriate hook facilitates this kind of behaviour with the added benefit that the subclass cannot forget to call the original method which it would otherwise have overridden directly. Unfortunately, this approach can only be implemented one level deep without creating new names for the hook at each level of inheritance. Obviously, the template method doesn't restrict the placement of hooks to only the beginning and end of methods, giving a subclass far more flexibility in how it reuses the code in a parent class.

3.3.6 Visitor

> *"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."* — *GoF*

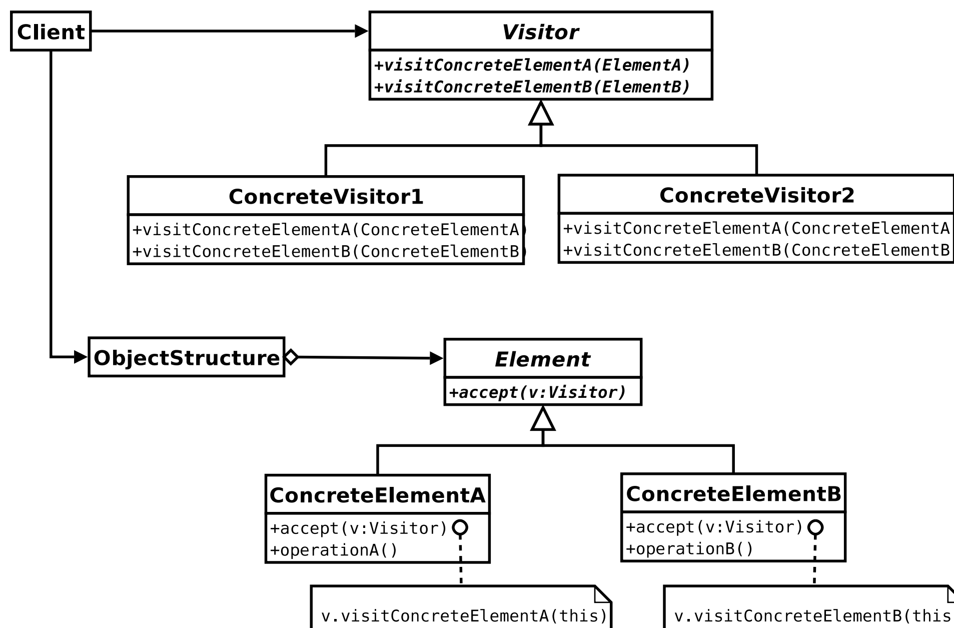


Figure 3.15: Visitor

Figure 3.15 illustrates the *Visitor* design pattern. The object structure can be any aggre-

University of Pretoria etd – Peer, E S (2005)

gate but is typically a tree structure such as an *Interpreter* hierarchy, as in Section 3.3.1, or a *Composite*, as in Section 3.2.2. A visitor encapsulates an operation which must be performed on each element of the object structure, while the accept method is responsible for traversing the object structure and calling the appropriate method for the type of element being visited. This calling strategy is known as double dispatch, since the method called to perform the operation is determined by both the type of the element in the object structure and the type of visitor.

Instead of spreading different parts of the same operation over multiple classes in a object structure, visitors enable related parts of an operation on multiple elements to be grouped into the same class. This clean encapsulation of an operation into a single class makes adding new operations easier, however, adding a new element type to the object structure requires changing all existing visitors to support it. Many of the special purpose methods in an *Interpreter* or *Composite* structure can be replaced with a single accept method for visitors that encapsulate those operations externally. Visitors also have the advantage of being able to accumulate state which may be difficult to distribute over multiple elements in an object structure. Unfortunately, because a visitor is external to the object structure, it may be necessary to provide a wider interface on the elements than would have otherwise been needed if the operations where supported internally within the structure. Thus, encapsulation for the elements may be adversely reduced so that visitors can perform their job.

3.4 Discussion

Design patterns are not an exact science. Patterns may be adapted and customised in the context in which they are being applied. Remember, design patterns are, for the most part, merely a way to encapsulate expert knowledge in an easy to digest form. They should be considered as guidelines for a good design rather than strict rules, since every situation is unique with its own set of constraints. Developers should still be free to be creative while building upon the knowledge gained from a study of patterns.

Patterns are also inter-related with certain patterns lending themselves to useful combinations. A few of these combinations have been hinted at in this chapter. Section 3.1.3 suggests that the *Prototype* can be used in conjunction with the *Abstract Factory* to alleviate the problem of parallel class hierarchies. The *Visitor* pattern, as discussed in

University of Pretoria etd – Peer, E S (2005)

Section 3.3.6, lends itself particularly well to a combination with the *Interpreter* or *Composite* patterns. Further, the *Abstract Factory* and *Facade* are often implemented as a *Singleton* when their implementations can be shared amongst multiple clients.

Finally, it should be noted that this chapter is not an exhaustive literary study of design patterns. There are more patterns presented in the GoF catalogue as well as many more ways that patterns are related to one another. Further, there are other catalogues that cover even more designs patterns, some of them specific to particular application domains. The content in this chapter is merely a terse summary of only those patterns that have been used in the implementation backing this work. Chapters 6 and 7 will refer back to the patterns presented in this chapter as appropriate.

University of Pretoria etd – Peer, E S (2005)

Chapter 4

Open Source Software (OSS)

> *"Gnu: n. [Hottentot gnu, or nju: cf. F. gnou.] (Zo["o]l.) One of two species of large South African antelopes of the genus Catoblephas, having a mane and bushy tail, and curved horns in both sexes. [Written also gnoo.]*
>
> *Note: The common gnu or wildebeest (Catoblephas gnu) is plain brown; the brindled gnu or blue wildebeest (C. gorgon) is larger, with transverse stripes of black on the neck and shoulders." — Webster's Revised Unabridged Dictionary.*

Open Source Software (OSS) [92], also known as free software [105], is any software distributed under a license conforming to the Open Source Definition (OSD) as published by the Open Source Initiative (OSI)[1]. An unannotated copy of the current OSD is attached as Appendix C, however, later versions may be published on the OSI web site as the definition is refined. An annotated version, describing the motivation for each clause of the definition, is also available from the OSI web site. Unlike the OSI, which approaches OSS from a very pragmatic perspective, the Free Software Foundation (FSF)[2] approaches OSS from a more ethical ideology concerned with civil liberties. Essentially, free software licenses are designed to protect four basic freedoms:

- **Freedom of use:** Recipients of OSS are granted the right to use the software for any purpose.

[1] http://www.opensource.org
[2] http://www.fsf.org

University of Pretoria etd – Peer, E S (2005)

- **Freedom to source:** Recipients of OSS are provided free access to the source code.
- **Freedom to modify:** Recipients of OSS are granted rights to prepare derivative works.
- **Freedom to distribute:** Recipients of OSS are granted rights to distribute the software, in original or modified form, either for free or for a fee.

While the OSI and FSF have somewhat different motives and are in disagreement about whether OSS should properly be called free software and *vice versa*, a common ground lies in the terms of the licenses that they both advocate. Therefore, the most popular OSS licenses and their characteristics are surveyed in Section 4.1.

OSS has many benefits for both developers and users of the software. From the user perspective, the zero marginal cost and high quality of OSS are often cited. Section 4.2 discusses the OSS ecosystem while concentrating on the benefits of OSS to developers. A common misconception regarding OSS is that it cannot be utilised for financial gain, however, it is certainly possible to make money from OSS through indirect sale business models such as those mentioned in Section 4.3. In fact, many large industry players such as IBM[3], Sun Microsystems[4] and Novell[5] have embraced OSS for profit.

OSS is of particular importance to developing countries. In particular, Section 4.4 discusses OSS in a South African context. Further, certain software pertaining to this work is distributed under an OSS license. Since this work constitutes University of Pretoria intellectual property, strong motives for releasing the software under such a license are provided in Section 4.5. Finally, this chapter concludes with credits in Section 4.6, listing the OSS that has been instrumental in completing this work.

4.1 Licenses

The characteristics of the most popular[6] and best known OSS licenses are compared in Table 4.1. The complete text of these licenses are provided in Appendix E as a reference.

[3]http://www.ibm.com
[4]http://www.sun.com
[5]http://www.novell.com
[6]According to SourceForge, http://sourceforge.net/softwaremap/trove_list.php?form_cat=14

University of Pretoria etd – Peer, E S (2005)

Terms and conditions for many other free software licenses are available on the OSI and FSF web sites. In addition, many OSS licenses have multiple versions and it should be noted that this work only considers the latest versions of those licenses at the time of writing. Newer versions will more than likely be published by the OSI or FSF as they come to exist.

While all of the licenses listed in Table 4.1 are OSI approved and are classified as free software licenses in terms of the four freedoms presented at the beginning of this chapter, they can be further divided into two broad categories: those that are copyleft, or GPL style; and those that are not, such as the BSD or MIT style licenses. Copyleft licenses place an additional restriction on the software, so they are less permissive and are therefore arguably less free licenses, requiring that any modifications, if distributed, must be made available under free terms again. A copyleft clause in a license essentially prevents free software from becoming non-free, which benefits the free software community as a whole even though the rights of any given individual within that community are curtailed.

The GNU General Public Licence (GPL), developed by the FSF as the license for the GNU Project[7], is probably the most important free software license in existence, with in excess of 39 thousand SourceForge[8] software projects licensed under its terms, including software developed for this work. The compatibility of other licenses to the GPL is an important characteristic of a license, since software licensed under incompatible terms cannot be linked against GPL software.

Table 4.1 further characterises licenses based on whether they permit additional warranty or liability protection to be sold and whether the license grants patent rights in addition to the four basic freedoms of free software.

Sections 4.1.1 through 4.1.9, in turn, detail the characteristics of each of the licenses presented in Table 4.1.

4.1.1 Academic Free License (AFL)

The Academic Free License (AFL, version 2.1), in Appendix E.1, is a non-copyleft license provided by the OSI. Software specific details are avoided in the license terminology,

[7]GNU: A recursive acronym for GNU's Not Unix; refer to http://www.fsf.org/gnu/thegnuproject.html for information about the GNU Project

[8]http://www.sourceforge.net

University of Pretoria etd – Peer, E S (2005)

Table 4.1: Open Source License Characteristics

License	Copyleft	GPL Compat.	Warranty Prov.	Patent Lic.
AFL	X	X	X	√
ASL	X	X	√	√
AL	X	√	√	X
BSD (original)	X	X	X	X
BSD (revised)	X	√	X	X
CPL	√	X	√	√
GNU GPL	√	-	√	X
GNU LGPL	-	√	√	X
MIT	X	√	X	X
MPL	-	-	√	√
OSL	√	X	X	√

making the license ideally suited for non-software works, such as documents, while still being general enough to apply to software.

The second clause grants a recipient of a work covered by the license a royalty-free right to use and sub-license patents. In addition, if a recipient enters into any patent infringement action against a licensor or licensee, that recipient's rights under the license are terminated. The patent termination clause makes the AFL incompatible with the GPL.

No provision is made for a licensor to sell additional warranty or liability protection. The work is licensed as is, without any warranties, aside from a warranty that applicable copyrights and patents are owned by the licensor, and disclaims all liability.

4.1.2 Apache Software License (ASL)

The Apache Software License (ASL, version 2.0) is a free software license with similar patent grant and termination clauses to the AFL, also making it incompatible with the GPL. Clause 9 permits anyone who distributes software under the ASL to provide additional warranty or liability protection. Finally, the license is not copyleft, meaning that any recipient may distribute the software under different license terms as long as

University of Pretoria etd – Peer, E S (2005)

all the obligations of the ASL, as specified in Appendix E.2, are met.

4.1.3 Artistic License (AL)

The Artistic License (AL, version 2.0beta4), presented in Appendix E.3, is designed to protect an originator's artistic control over future versions of the software. In essence it requires modified versions to clearly indicate any changes and satisfy one of the following conditions: i) the changes must be submitted back to the original contributor for consideration in the standard version, ii) the package must be renamed to something that cannot be confused with the original, or iii) it must be made available under free terms to whomever it is distributed to.

Although the AL is scattered with hints of copyleft concepts, clause 6(b) clearly allows the software to be made non-free, so long as any changes are documented and that it cannot be confused with the original work. The license is, however, GPL compatible and although no specific clause specifically applies to additional warranty provisions, the standard warranty disclaimer text, in clause 11, does permit such provisions to be stipulated in writing. Patent licenses are not covered.

4.1.4 BSD Licenses

The revised BSD license, presented in Appendix E.4, is an extremely permissive non-copyleft license which primarily ensures that copyright notices are properly maintained. The original version had an additional advertising clause, requiring the University of California, Berkeley and its contributors to be credited in any advertising material, making it incompatible with the GPL. Neither version permits the names of any contributors to be used as an endorsement to promote the licensed work. Both forms of the license explicitly disclaim all liability and warranties while saying nothing about patents.

4.1.5 Common Public License (CPL)

The Common Public License (CPL, version 1.0), in Appendix E.5, has been designed to facilitate the commercial use and distribution of software. The CPL is not compatible with the GPL. It has similar patent grant and termination clauses to the AFL and ASL, but unlike those licenses, it offers some copyleft characteristics.

University of Pretoria etd – Peer, E S (2005)

The copyleft terms in the CPL are not as stringent as the GPL, since separate modules may be licensed under their own terms. While derivative works are explicitly excluded from this concession, it is not explicitly clear where the boundary between a module and a derivative work lies. Binary distribution under another license is also permitted provided that i) warranty and liability exclusions are carried over, ii) source code is made available to a recipient on request, and iii) the terms of the other license do not otherwise conflict with requirements of the CPL.

Warranty and other liability protections may be offered provided that any other contributors are properly indemnified. That is, a distributor offering additional protections accepts all responsibility, including defending any legal claims made against any contributor.

4.1.6 GNU General Public Licenses (GPL and LGPL)

The GNU General Public License (GPL, version 2), presented in Appendix E.6, is a strong copyleft license. In fact, the GPL is the original definition of copyleft. That is, the copyleft terminology was coined by the FSF to encompass those properties of the GPL that keep software free. In the case of the GPL, copyleft is accomplished by requiring that any derivative work must again be distributed under the free terms of the GPL, if it is distributed at all. As a consequence, if a portion of a work is licensed under the GPL then the whole may not be distributed at all, except under terms of the GPL, since the whole would qualify as a derivative work.

On the other hand, the GNU Lesser, or originally Library, Public License (LGPL, version 2.1), in Appendix E.7, has more relaxed copyleft requirements. The LGPL was originally written to enable a free software library to be used by a non-free, or proprietary, work without requiring the whole to be made freely available. However, any improvements or other changes to the library itself are still required to be distributed under the free terms of the LGPL. That is, a work covered by the LGPL will remain free while any other separate work that links against it, technically a derivative work, is not required to be released under the terms of the LGPL. Since the LGPL is applicable to more works than just libraries, it was renamed the Lesser GPL, to reflect the less stringent copyleft requirements. Any recipient of a LGPL work may choose to redistribute it under the more restrictive copyleft terms of the GPL.

University of Pretoria etd – Peer, E S (2005)

Incompatibilities with the GPL arise from clauses 6 and 7 of the GPL, which state that a distributor may not impose any further restrictions on a recipient beyond what the terms of the GPL permit. To do so would render a work undistributable under the GPL. For example, a condition of the GPL is a royalty free right to use the software licensed under its terms, however, if a combined work consists of some non-GPL portions which would prevent such royalty free use, perhaps due to a patent, then the right to distribute the GPL portion falls away too, leaving the whole in a state which cannot be distributed under either license. For this reason, patent termination clauses in other licenses cause an incompatibility with the GPL. Neither the GPL nor the LGPL explicitly include a patent grant, however, clauses 6 and 7 do provide free software with a certain level of protection from patents, in so far as the free software cannot be distributed by a patent holder under terms other than the GPL. The FSF has recently announced plans to release a new version of the GPL[9], which is likely to have patent terms that are more compatible with other popular OSS licenses. Since the LGPL is essentially the same as the GPL, except for the more lenient copyleft terms, it is GPL compatible.

Both the GPL and LGPL grant distributors of software the freedom to offer additional warranties or liability cover to their recipients.

4.1.7 MIT License

The MIT license, presented in Appendix E.8, is probably the least restrictive free software license. Permission to use, modify and distribute the software is granted provided that the copyright and permission notice is preserved. The permission notice also includes a simple disclaimer which explicitly disclaims any liability or warranties. Since it essentially does not place any restrictions on the software it covers, it is GPL compatible and non-copyleft.

4.1.8 Mozilla Public License (MPL)

The Mozilla Public License (MPL, version 1.1), in Appendix E.9, has similar copyleft properties to the LGPL. Clause 3.7 permits a larger work to be composed and distributed under a different license provided that the MPL requirements are fulfilled for the covered code. In addition, patent licenses, subject to litigation termination terms, are granted

[9] http://www.eweek.com/article2/0,,1730102,00.asp

University of Pretoria etd – Peer, E S (2005)

by the MPL. Clause 3.5 explicitly provides for warranty support or liability obligations under the condition that other contributors are properly indemnified. Finally, an initial developer may, subject to clause 13, choose to license portions, or the whole, of the work under multiple license terms, including GPL, making those parts GPL compatible.

4.1.9 Open Software License (OSL)

The Open Software License (OSL, version 2.1), presented in Appendix E.10, is virtually identical to the AFL, except that copyleft properties are ensured by clause 1c, which requires derivative works to be distributed under terms of the OSL. Like the AFL, the OSL grants patent licenses, is not GPL compatible and makes no provision for additional liability or warranty cover.

4.2 The Open Source Ecosystem

Hardin's tragedy of the commons describes the inevitable demise of any freely shared resource, the commons, if no resource allocation policy is enforced [51]. As an example, Hardin considers the scenario of a public pasture which is freely shared amongst a number of cattle farmers. The grazing cost, in terms of damage to the pasture, of another head of cattle is diluted by the commons, while any given farmer still retains the full profits associated with owning more cattle. This imbalance gives each farmer the incentive to add more and more cattle, to extract the maximum value from the commons as quickly as possible before it degrades due to over grazing. There is no incentive to contribute to the maintenance of the commons, since any returns would again be diluted.

Freely available OSS, however, does not suffer this tragedy [92, 44, 104]. There are two contributing factors to the tragedy of the commons: i) there is a limited supply of resources; and ii) the lack of an enforced allocation policy drives demand up until the supply is depleted. Fortunately, in today's Internet connected world, software costs virtually nothing to duplicate. As a resource, software is not depleted by the act of copying, so free riders do not degrade the commons. On the contrary, a larger user base actually increases the value of OSS. Thus, the demand side of the equation is taken care of, and tragedy is avoided. On the other side of the equation, there are strong incentives for developers to contribute to the commons, ensuring sufficient supply of free software.

University of Pretoria etd – Peer, E S (2005)

Compelling reasons why people and organisations contribute to the free software commons include:

- **Peer review and reputation rewards:** A large user base can be a free software project's biggest asset. Aside from the benefits of having users provide bug reports and feature requests, high profile projects also offer the highest reputation rewards, attracting the attention and cooperation of other developers. The peer review process associated with more developers, in turn, improves the quality of the software.

- **Cost and risk sharing:** Customising existing free software to meet the specific needs of a user can be cheaper than developing a solution from scratch. Further, there is a strong incentive to contribute any improvements back to the community, even ignoring possible copyleft constraints on the existing software. To see why, consider the situation where a user chooses not to contribute those improvements back to the community. Now, that user needs to maintain a separate version, possibly merging it with improvements from the community version from time to time. This can be an expensive undertaking, particularly if the community version undergoes incompatible changes. Contributing the changes back avoids this problem. Thus, it is a reasonable assumption for an initial contributor of software to expect others to contribute improvements, initiating a cost sharing development excersise. Also, the community offers safety. The risk of having only a few people being able to maintain the software can be mitigated by sharing that maintenance burden with the community, so that more than one entity has a vested interest in the survivability of the software.

- **Growing secondary markets:** Very importantly, there is money to be made from free software. By growing the community around a free software product, related secondary markets are opened up. The indirect sale business models presented in the next section exploit this property.

4.3 Business Models

OSS licenses typically do not prevent the distribution of software for a fee, however, some do require that such a fee be at most the reasonable cost of copying. More importantly,

University of Pretoria etd – Peer, E S (2005)

all OSS licenses explicitly grant any recipient the right to freely distribute the software again, making it difficult to build a direct sale business based on OSS. That said, several indirect sale business models exist to exploit free software for financial gain [92]:

- **Loss leader/market positioner:** Free software is used to maintain or create a secondary market for other non-free software. Thus, the use of the free product drives sales of the non-free product. For example, giving away free development tools in order to maintain the market for application servers, which is what IBM is doing with the Eclipse platform to drive sales of their WebSphere[10] solution.

- **Widget frosting:** Hardware products typically require accessory software which does not have any sale value independent of the hardware. For example, drivers or configuration software. By opening up the software, a hardware vendor can benefit from a larger developer pool, better reliability through peer review and maintenance beyond the expected product life cycle. All without sacrificing any revenue stream, since it is the hardware that brings in the money. A concrete example is Apple's[11] decision to open source Darwin, the core of MacOS X, since they are primarily interested in selling the hardware on which the operating system runs.

- **Give away the recipe, open a restuarant:** The software is provided freely and services are sold to the created market. For example, vendors may choose to sell support contracts, performance assurances, customisation services, training and maintenance of the software according to the client's time table. RedHat[12], for example, sells support and patch management for their open source Linux product.

- **Accessorising:** Accessories to the software are sold. Trivial examples include mugs and t-shirts, while publishers such as O'Reilly sell high quality books about free software. Other accessories might include non-free plug-ins that enhance the functioning of the software.

- **Free the future, sell the present:** Software is initially sold under a closed license with the provision that it will be released under a open license at a later

[10]http://www-306.ibm.com/software/webservers/appserv/was/
[11]http://www.apple.com
[12]http://www.redhat.com

University of Pretoria etd – Peer, E S (2005)

date. Sales volumes are driven by the expectation that the software will become free later, while the vendor benefits from the reduced maintenance overhead later in the product life cycle.

- **Free the software, sell the brand:** The software implementation is free. Customers must satisfy compatibility requirements and pay for the certification of the brand.

- **Free the software, sell the content:** The software is free, while content subsciptions are sold. For example, a game engine might be given away freely while the story is sold for a price.

- **Dual licensing:** This model requires the vendor to own, or at least control, all copyrights pertaining to the software. The product is released to the public under a strong copyleft license, such as the GPL, making it impossible to distribute the free software component as part of other non-free commercial software. Simultaneously, the software is sold, under a non-free license, to clients that wish to incorporate the software into commercial software. A community is built around the free version of the software, building market awareness of the product. Typically, improvements from the community may only be incorporated into the non-free version with the permission of a contributor. Vendors may require copyrights to be signed over in order for improvements to be incorporated into the free reference version. Dual licensing has been successfully employed by MySQL[13], for their database product, and Sun Microsystems[14], for their StarOffice product which is available in a scaled down form as OpenOffice[15].

The common theme amongst open source business models: software is provided for free to produce a secondary market where additional value can be sold for a price.

4.4 Open Source in a South African Context

An official open source strategy [3] has been proposed by the local South African government. The proposal addresses the benefits of OSS in a South African context, rec-

[13] http://www.mysql.com
[14] http://www.sun.com
[15] http://www.openoffice.org

University of Pretoria etd – Peer, E S (2005)

ommendations for building local competencies in open source and a long term strategy for providing government support for open source projects.

Key economic benefits, amongst others identified in the report, are the development of local software development skills and the saving of foreign currency, since most commercial software is developed abroad. By leveraging open source as an educational vehicle, local skills in software development are developed, which in turn will stimulate SMME (Small, Medium and Micro Enterprises) growth in the IT (Information Technology) sector. Some responsibilities (quoted directly from the report) of educational institutions for building a capacity in open source are:

- "It is critical that strong linkages be set up with institutions of higher learning to build a national collaborative network that can be extended internationally."
- "Training for OSS developers and OSS users must be available. Institutions of learning must fulfil a role in this respect."
- "A well-run research programme will be needed to enable optimal understanding and decision making on OSS. The model for this research programme should be built on the networking nature of the OSS development model, harnessing the potential of institutions of higher leaning and schools."

The advantages that OSS holds for the local economy makes it the responsibility of every South African citizen to leverage OSS whenever it makes business sense, reducing foreign spending on software and creating a demand for local skills in the secondary markets discussed in the previous section. The Shuttleworth Foundation[16] is setting a fine example by actively promoting OSS in South Africa, targeting the general public with a wide reaching "Go Open Source"[17] awareness campaign, and facilitating the use of OSS in schools.

[16]http://www.shuttleworthfoundation.org
[17]http://www.go-opensource.org/

University of Pretoria etd – Peer, E S (2005)

4.5 University of Pretoria Intellectual Property

The University of Pretoria (UP)[18], like most universities, retains ownership of any Intellectual Property (IP) submitted by students for degree purposes[19]. This means that any decision to license the source code pertaining to this work, which is material covered by UP copyrights, to third parties legally rests with the university's IP authorities. Therefore, permission to publish the CILib source code under the GPL needed to be granted officially. A draft of the letter granting this permission is included as Appendix D. The following reasons were offered as motivation for obtaining this permission:

- **Collaboration, reputation and peer review:** The CIRG@UP would like to solicit the collaboration of third parties to accelerate the development of CILib through a mutually beneficial sharing of development resources. By releasing the source code under the GPL, the group hopes to benefit from the OSS peer review process, with a goal of producing a reliable and error free software platform capable of engendering a community's trust in its code base. Further, the copyleft nature of the GPL should encourage those who find the software useful to contribute any improvements they may make back to the community. If successful, the University of Pretoria, as initial contributor and founder of the community, will benefit from the reputation associated with such a project.

- **Use of other GPL software:** Distributing software under the GPL enables it to incorporate other GPL software. For example, CILib makes use of simulation quality random number generators ported from the GNU Scientific Library (GSL)[20], which is only licensed to the university under the GPL. This also means that CILib may not be distributed under any license terms besides the GPL. At that time, the university could have chosen not to distribute the software at all, keeping it secret and losing out on all the other benefits mentioned here. Since the university currently owns the rest of the copyrights pertaining to CILib, it may choose to distribute those components which it owns under its own terms at any point in the future. That is, provided the GSL components are removed, that version of CILib may be licensed under other terms, however, the quality of any simulations

[18] http://www.up.ac.za

[19] According to the contract signed by students upon application for a degree.

[20] http://www.gnu.org/software/gsl/

University of Pretoria etd – Peer, E S (2005)

performed using the software would be severely diminished, reducing the value of the software as a product. Note that nothing can retroactively revoke any rights that the university has granted to any third party who has already received a copy of CILib under the GPL.

- **Social Responsibility in a South African context:** Given the discussion in the previous section, it is important for the university to be a good citizen of the open source community. In fact, the UP is actively pursuing open source research through initiatives such as digital@SERA [111], a division of the Southern Education and Research Alliance (SERA)[21] which is a joint venture between the UP and the CSIR (Council for Scientific and Industrial Research)[22] focused on fostering collaborative and sustainable research. CILib is simply another opportunity to develop local skills while researching the applicability of the OSS development model with respect to collaborative research.

- **Business opportunities:** Building a community around a freely available software product creates the potential to exploit secondary markets, due to increased visibility of the product in the market place.

  In the case of CILib (refer to Chapter 6), it is conceivable that a future third party might like to utilise the software in a commercial product offering. As discussed previously, the university may license the software on its own terms to such a third party for a fee, provided it satisfies its GPL obligations, by excluding any GPL material not covered by university copyrights. Further, the university may be able to co-operate in some kind of profit sharing scheme with other copyright holders to offer a product of increased value to commercial third parties. Policies requiring potential contributors to sign over their copyrights or grant permission for their work to be included commercial offerings should be avoided, since such policies may discourage contributions.

  For CiClops (refer to Chapter 7), the CIRG@UP is still undecided as to an appropriate course of action. The university may choose to keep it proprietary, following a “loss leader/market positioner” business model. Under this model, CiClops is used to maintain a central repository of CILib simulation data while selling the

[21] http://www.seralliance.com/

[22] http://www.csir.co.za

University of Pretoria etd – Peer, E S (2005)

services of the software and the use of the data repository. The difficulty will be gaining the trust of third parties, if they cannot access the source code, they cannot verify the correctness of the software or the integrity of the data repository. On the other hand, a "free the software, sell the content" model which does not have this problem could be pursued. In this model, only the data repository and university computing resources are sold as a service. The danger with this is that it opens the door to competing repositories, discouraging collaboration on a single data repository.

4.6 Credits

Table 4.2: Instrumental Open Source Software

Package	**License**	**Web Site**
Apache Ant	ASL	http://ant.apache.org
CVS	GNU GPL	http://www.cvshome.org
Dia	GNU GPL	http://www.gnome.org/projects/dia/
Eclipse	EPL	http://www.eclipse.org
Emacs	GNU GPL	http://www.gnu.org/software/emacs/emacs.html
GNU/Linux	GNU GPL	http://www.fsf.org/gnu/linux-and-gnu.html
		http://www.gentoo.org
JBoss	GNU LGPL	http://www.jboss.org
JUnit	CPL	http://www.junit.org
Mozilla	MPL	http://www.mozilla.org
MySQL	GNU GPL	http://www.mysql.com
NetBeans	SPL	http://www.netbeans.org
teTeX	Various OSS	http://www.tug.org/teTeX/
XDoclet	BSD (revised)	http://xdoclet.sourceforge.net/xdoclet/index.html
Xfig	Xfig custom	http://www.xfig.org

Table 4.2 presents a list of free software titles which can be credited for making this work possible. The distribution license and web site where futher information can be obtained

University of Pretoria etd – Peer, E S (2005)

are also listed alongside each title.

On the software implementation front, Eclipse, distributed under the Eclipse Public License (EPL), and NetBeans, distributed under the Sun Public License (SPL), have both been used as development environments. Software version control is maintained using the CVS (Concurrent Versioning System), since it is the only version control system currently supported by SourceForge. A recent SourceForge circular announced plans to support the more modern Subversion[23] system in the near future. The Apache project's Ant is the tool used to script the build process for all the software developed for this work. Software unit testing is performed using the JUnit framework. Components of the software are deployed on a JBoss application server using XDoclet to generate the necessary deployment descriptors and ancillary interfaces. MySQL has been used to provide the relational database back-end used by the application server.

This dissertation has been composed using the Emacs text editor and typeset with the teTeX LaTeX processor. All UML diagrams were composed using Dia, while the remaining figures have been drawn using Xfig. The Mozilla browser has been used for researching resources on the web. Finally, underlying all this excellent software has been the GNU/Linux operating system. This work would not have been possible, at least not within budget constraints, without the aid of free software.

[23]http://subversion.tigris.org/

University of Pretoria etd – Peer, E S (2005)

Chapter 5

Languages and Tools

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning." – Rich Cook

This chapter addresses various language and tool prerequisites for working with the software implemented for this research.

Section 5.1 introduces XML, which is used as a configuration and data representation language. Java and J2EE, which were chosen as implementation platforms are discussed in Sections 5.2 and 5.3 respectively. Section 5.4 presents the XDoclet tool, which enables attribute oriented programming. The JUnit framework, used for writing software unit tests, is introduced in Section 5.5. Finally, the chapter concludes with a brief summary in Section 5.6.

5.1 XML (eXtensible Mark-up Language)

XML (eXtensible Mark-up Language) is a recommendation by the World Wide Web Consortium (W3C)[1], for defining structured documents [53]. Structure is imposed on a text document by marking up the content with user defined tags. Figure 5.1 is an example of a simple XML document, a structured list of phone numbers.

Note that, given the proper choice of tag names, a document is reasonably self describing. It should be clear, even to somebody unfamiliar with XML, that the example

[1] http://www.w3c.org/XML/

University of Pretoria etd – Peer, E S (2005)


```
<?xml version="1.0"?>
<!DOCTYPE phoneBook SYSTEM "phonebook.dtd">
<phoneBook>
  <contact>
    <name>Joe Bloggs</name>
    <phone type="home">012-315-7834</phone>
    <phone type="cell">082-243-4244</phone>
  </contact>
  <contact>
    <name>John Doe</name>
    <phone type="home">012-514-1423</phone>
    <phone type="work">011-612-3431</phone>
    <phone type="cell">083-561-9542</phone>
  </contact>
  <!-- possibly more contacts -->
</phoneBook>
```

Figure 5.1: A Simple XML Phone Book Document

is a list of contacts in a phone book with their associated phone numbers. More importantly, because the document is structured, according to the `phonebook.dtd` document type definition, software can make sense of it too. The power of XML stems from the fact that standard tools can be used for manipulating any well formed document and that the grammar for a particular type of document can be defined and extended to suit its natural structure.

For example, the logical structure of a book can be broadly defined in terms of chapters, sections and paragraphs. DocBook [115], which defines an XML document type for marking up books and technical documentation, enables an author to write a book based on its natural logical structure. Since the book is just another XML document, the structure is machine readable and so standard style sheet templates can be used to transform the document into any format, in any desired medium.

Section 5.1.1 defines the syntax requirements for XML documents to be well formed. Next, document types and schemas are discussed in Section 5.1.2. Finally, the Document

University of Pretoria etd – Peer, E S (2005)

Object Model (DOM) is explained in Section 5.1.3.

5.1.1 Well Formed Documents

A document and its tags, more formally known as elements, must satisfy certain rules in order to be well formed [123]. Any well formed document is guaranteed to be parsed without error by a standard XML parser.

There are three simple rules pertaining to elements: i) there must be one and only one root element; ii) an opening tag must be followed by a corresponding closing tag, where matching is case sensitive; and iii) elements must be properly nested, so an opening tag which is outside the scope of a nested element must be closed in the same outer scope.

Elements may contain optional attributes, such as the type attribute in the phone elements in the example. Further, elements may be empty, in which case the element may be closed, using a shorter syntax, by suffixing the opening tag name with a forward slash, for example <element/>, instead of <element></element>. Empty elements may still contain attributes. Special cases include id attributes, which are used to associate a document scoped unique identifier with an element, and corresponding idref attributes, which are references that can be followed to elements identified by an id attribute.

Further, there are restrictions on the characters that may be used in attribute and tag names. Only alphanumeric characters, hyphens, underscores and periods may be used. Throughout a document, the literal strings "&" and "<" must be used in place of the "&" and "<" symbols respectively, otherwise they would be mistaken as mark-up. Similar string literals are defined for quote, apostrophe, and greater than symbols, but their use is optional. Another way to prevent character data from being processed as mark-up is to include it within a special CDATA tag, for example "<![CDATA[text that should not be processed]]>". Finally, comments are enclosed within the "<!--" and "-->" tags.

5.1.2 Document Types and Schemas

Documents that conform to a given structure, constrained by either a DTD (Document Type Definition) or a schema, are known as valid documents. These constraints are enforced by the XML parser before an application sees a document. Validated documents permit software to make assumptions about the structure of a document, making XML

University of Pretoria etd – Peer, E S (2005)

processing software easier, and safer, to write.

```
<!ELEMENT phoneBook (contact*)>
<!ELEMENT contact (name, phone+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ATTLIST phone
  type CDATA #REQUIRED
>
```

Figure 5.2: Phone Book Document Type Definition (phonebook.dtd)

Figure 5.2 provides the DTD for the phone book example. A DTD defines all the valid document elements and their relationships with their children. A suffix of "?", "*" or "+" after a child element name determines the number of children elements which may occur, namely, zero or one, zero or more and one or more respectively. Sequences are indicated by a comma separated list of children. Thus, the second line indicates that a `contact` element must consist of a `name` element followed by one or more `phone` elements. Legal attributes are defined by an `ATTLIST` description. The `PCDATA` type corresponds to character data that will be parsed for further mark-up, while the `CDATA` type is ordinary character data. Note that an attribute value may not be of type `PCDATA`, it will never be processed as mark-up. The `DOCTYPE` reference in the document instance specifies which element in the DTD should be considered as the root element.

Instead of using a DTD, an XML Schema [112, 13] can be used to define a document type. Schemas have several advantages over DTDs. Firstly, because the schema language is just another XML document, there is no need to learn a separate DTD language, and standard parsers and tools can be used to read and manipulate schemas. Furthermore, XML Schema has a more extensive type system that supports inheritance. Most importantly, because schemas are supported using namespaces, a single document can mix document elements from multiple schemas, simply by declaring multiple namespaces that reference different schemas.

The schema for the phone book example is presented in Figure 5.3. The `xmlns:xs` attribute in the root element defines the `xs` namespace. Thus, elements prefixed by `xs:` are instances of the `http://www.w3.org/2001/XMLSchema` schema. In this par-

University of Pretoria etd – Peer, E S (2005)


```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="phoneBook">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="contact"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="contact">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="phone"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="phone">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="type" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 5.3: Phone Book Schema (phonebook.xsd)

University of Pretoria etd – Peer, E S (2005)

ticular case, the namespace is a reference to the definition of the valid elements for an XML Schema document, which is also defined in terms of XML Schema. Default namespaces of documents can also be defined by schemas. Thus, in the phone book example, the `DOCTYPE` line can be omitted and the root element altered to read `<phoneBook xmlns="phonebook.xsd">`, where `phonebook.xsd` is the file containing the phone book schema.

5.1.3 Document Object Model (DOM)

The Document Object Model (DOM) provides a language neutral interface for manipulating XML documents programmatically [58]. XML documents are represented by an in memory tree based object structure, where nodes are defined for all possible components of an XML document, including elements, attributes, comments and free standing text. Since DOM bindings exist for all major programming languages, XML content to be accessed, processed and manipulated in a standard way on any platform.

As an alternative to the DOM, the Simple API for XML (SAX)[2] provides an event model interface for processing documents. SAX, which is an extension of the *Observer* pattern in Section 3.3.3, enables documents to be processed without the need to build a, possibly large, in-memory representation.

5.2 Java

Java is a modern, high level, general purpose, object oriented programming language [33, 59]. Programs written in Java are compiled into an intermediate language, known as byte code, which is interpreted at run time by a Java Virtual Machine (JVM). Benefits of Java include:

- **Platform and Vendor Independence:** A cornerstone of Java has always been the concept of write once, run anywhere. This goal has been achieved by virtue of the JVM, since only the underlying virtual machine need be ported to each platform where Java is supported. Supported platforms include Windows, Linux and MacOS. Further, the Java specification is guided by the Java Community Process

[2] http://www.saxproject.org/

University of Pretoria etd – Peer, E S (2005)

(JCP)[3], giving multiple vendors the opportunity to contribute and participate in the decisions that dictate the future direction of Java. Competing JVM implementations are available from multiple Java vendors, including Sun Microsystems, IBM, BEA[4] and the Blackdown project[5], ensuring diversity in the market place and the future safety of the Java platform. A completely free JVM implementation is also being worked on by the GNU Classpath community[6], along with a native Java compiler as part of the GNU Compiler Collection (GCC).

- **Garbage Collection:** Garbage Collection (GC) relieves a programmer from having to explicitly manage memory deallocation, resulting in safer code due to the reduced risk of introducing difficult to find memory leaks. GC is associated with at least some overhead, since an additional process must be executed from time to time to recycle unreferenced memory. Counter intuitively, in spite of this overhead, GC can have a net increase in the performance of an application[7]. For example, heap compaction performed by GC increases the likelihood of cache hits. Further, since GC only executes when memory is tight, programmes with a low memory footprint may never need to run a GC cycle. Another factor to consider is that smart pointer based reference counting techniques, which are typically employed to simplify memory deallocation in non-GC languages, can carry a much higher overhead than GC, since counters need to be updated for every assignment. Worse, reference counting techniques are dangerous because they cannot deal with circular references or anonymous objects. Finally, explicit destructors can be a significant performance overhead for stack allocated resources.

- **Java Foundation Classes (JFC):** The Java platform, which is guaranteed to be available on any compliant JVM, is defined in terms of the JFC. The JFC, or Java APIs, provide XML processing, Input/Output (I/O), Graphical User Interface (GUI) and networking services to applications. Further, since version 1.5 of the JFC, a type safe collections framework using templates is also provided. Also, the reflection API is a fundamental reason Java was chosen as an implementation

[3]http://www.jcp.org
[4]http://www.bea.com
[5]http://www.blackdown.org/
[6]http://www.gnu.org/software/classpath/classpath.html
[7]http://www.digitalmars.com/d/garbage.html

University of Pretoria etd – Peer, E S (2005)

language for this research. The JFC has been through many revisions, gradually improving its design, which is heavily based on design patterns. For example, I/O services such as buffering and compression are provided using *Decorators* (refer to Section 3.2.3) and the collections framework supports *Iterators* (refer to Section 3.3.2).

- **Tool Support:** Many high quality Java development tools are freely available. At least two good enterprise class development environments are available for free, namely Eclipse and NetBeans. The Javadoc tool, packaged with the standard Java SDK (Software Developer Kit), extracts special comments in the source code into a navigable HTML (HyperText Mark-up Language) format. XDoclet (refer to Section 5.4), originally based on Javadoc, can be used to generate various artifacts from meta-data embedded in special Javadoc comments. Debugging distributed and server side applications can be made simpler with a logging framework such as Log4j[8]. JUnit (refer to Section 5.5) is a unit testing framework for Java. The build process of complex Java projects is script-able using the Apache Ant[9] build tool.

  These are only the tools that have been used, or are being considered, for this research. There are many other free third party tools, frameworks and APIs available for Java, supported by a diverse Java community.

- **Performance:** Java is still plagued by the stigmatism of poor performance due to early and immature implementations of the JVM. This situation is further exacerbated by the intuition that interpreted languages with additional GC overheads must have inferior performance to natively compiled languages. Modern HotSpot [1] JVMs, however, have dramatically improved the performance of Java, to the point where it is comparable and in certain circumstances superior in performance to natively compiled languages such as C/C++ [25, 95]. HotSpot JVMs sport state of the art generational GC algorithms, speculative run time optimisation using dynamic profiling, and Just In Time (JIT) compiling of critical code, known as hot spots, to instructions optimised for the local processor. Numerous micro-benchmarks [72, 76, 23, 70] have been conducted, which show Java performance to

[8]http://logging.apache.org/log4j/docs/index.html
[9]http://ant.apache.org

University of Pretoria etd – Peer, E S (2005)

be on par with other languages.

A simple benchmark, called NastyPSO, was performed around the time the decision to port the implementation code used in this research to Java was being made. NastyPSO[10] is a quick and dirty implementation of a simple particle swarm optimiser (refer to Section 2.4.1) in C#, C++ and Java. To make the benchmark fair, no language specific libraries are used. For example, the random number generator used in the code is implemented by NastyPSO in each language. Thus, the only differences between the implementations are syntactic. Further, no OO features of are used, purely testing the number crunching ability of each language. The source code for NastyPSO is made available so that the results presented in Table 5.1 can be verified independently by the reader.

Table 5.1: NastyPSO Performance

Language	**Compiler / VM**	**Time (seconds)**
C++	Intel Compiler (-O3 -march=pentium4)	391.3
C++	Intel Compiler (-O3 -march=pentium4 -mp)	570.6
Java	Sun HotSpot VM 1.4.2.03 (-server)	584.6
Java	IBM VM 1.4.1	584.8
Java	Sun HotSpot VM 1.5.0_beta1 (-server)	600.8
C++	GNU Compiler 3.3.2 (-O3 -march=pentium4)	742.8
C++	GNU Compiler 3.3.2	754.0
Java	JRockit 8.1	756.8
Java	GNU Compiler (GCJ)	934.4*
Java	Blackdown 1.4.1 (-server)	945.0
Java	Sun HotSpot VM	992.4*
Java	Sun Classic VM	1596.5*
C#	Mono 0.28	2572.9

Times recorded are the CPU scheduled time given by the Unix `time` command, so the results are invariant to varying load on the machine. Unfortunately, some parameters,

[10]http://cilib.sourceforge.net/NastyPSO/

University of Pretoria etd – Peer, E S (2005)

which are hard coded in the implementation, have changed since the time the benchmark was performed and were not properly recorded. Further, times suffixed by an asterisk have been interpolated based on a run conducted several months earlier, where the versions of the compilers and virtual machines were not recorded. The scaling was performed relative to the performance of the Sun HotSpot (Server) VM, which was a common denominator in both sets of results, even though the versions may not have matched. That said, conclusions about the relative performance of the implementations are still valid, even though the times may not exactly match those produced by the current version of the code.

The first conclusion evidenced by the results is that the choice of JVM can have a measurable performance difference. In fact, selecting the server parameter of the Sun JVM made the difference from one of the worse performing configurations to one of the best. The server JVM performs more aggressive run time optimisations at the cost of slower startup times, making it suitable for long running processes. Surprisingly, the free GNU compiler was unable to match the best JVM performances, even under very heavy optimisations for the platform. The Intel[11] compiler was able to outperform the best Java configuration, however, if the compiler was forced to reject optimisations that may affect the floating point precision then this difference was not significant. C# was tested under the free Mono platform and was found to perform significantly worse than any of the other configurations. Microsoft's[12] implementation of the .NET platform was not tested, since it is not platform independent.

Unfortunately, OO polymorphic method calls are still expensive, even in C++ although less so than Java, making object based polymorphic numeric types expensive, particularly in the tight loop applications needed by CI algorithms. Fortunately, object in-lining [18] may provide a solution to this problem in future. Object in-lining is a compile time optimisation that essentially unpacks code into a calling class whenever polymorphism is not required, so a developer can write clean OO code while leaving the hard work of making it perform well to the compiler.

[11] http://www.intel.com
[12] http://www.microsoft.com

University of Pretoria etd – Peer, E S (2005)

5.3 Java 2 Enterprise Edition (J2EE)

Java 2 Enterprise Edition (J2EE) is centred around Enterprise Java Bean (EJB) technology, enabling the development of scalable multi-tiered enterprise class applications [8]. EJBs are software components that are managed in the context of an application server container. The container forms the interface between EJB components and the underlying platform, providing caching, clustering, security, session, transaction, and persistence management services.

An EJB comprises three essential components: i) an application interface; ii) a home interface; and iii) an implementation class. The application interface, also known as a business interface, specifies the services that a bean provides to clients. Programming to an explicit interface with no direct knowledge of the implementation means that the implementation can be switched without affecting any clients. The Java Naming and Directory Interface (JNDI) provides an additional level of indirection, making implementation classes configurable at application deployment time. Thus, EJB clients are not aware of the implementing class details, they are only exposed to an abstract JNDI name for the implementation providing the service. Primarily, home interfaces are responsible for managing the life cycle of individual beans, providing methods for locating and creating them. Beans are destroyed by calling a remove method directly on an instance. Services that apply to more than one particular bean instance are also provided by the home interface, making those services analogues for class scope, or static, methods. Further, EJB interfaces for local and remote clients are differentiated in J2EE, so different subsets of a bean's services can be provided to local and remote clients. Finally, an implementation class for an EJB provides the code behind both the home and application interfaces.

The J2EE architecture is layered, cleanly separating different responsibilities into separate layers. At the lowest level, the persistence layer, discussed in Section 5.3.1, is responsible for managing the storage of application data. Above that, the application layer, in Section 5.3.2, is responsible for handling all the application logic, also known as business logic. The presentation layer, discussed in Section 5.3.3, provides the interface to the user. In general, separating the architecture into even more layers is possible, if it makes sense to do so in the context of the application. The purpose of the layers is to improve maintainability of the code by decreasing the dependencies between layers,

University of Pretoria etd – Peer, E S (2005)

changes to one layer should at worst affect the layer immediately above it. In addition, the separation of application and presentation logic means that the same application logic can be used for multiple presentation mediums. For example, a rich GUI client and a web interface, both separately implemented in the presentation layer, should share the same application logic. Finally, the deployment of J2EE applications is discussed in Section 5.3.4

5.3.1 Persistence Layer

Two types of persistence EJB exist in the J2EE specification, Container Managed Persistence (CMP) beans and Bean Managed Persistence (BMP) beans. BMP beans require persistence logic to be manually coded by the developer, while CMP beans delegate persistence logic to the application server container.

Persistence EJBs, also known as entity beans, present an OO view of an underlying relational database [30], or indeed any data store. Although the object relational mapping need not necessarily be a one-to-one correspondence with the underlying database tables, each entity bean instance typically represents a single row in a relational database table. Each column corresponds to a property of the CMP bean, where a property has its usual OO definition of a field with an accessor, or get method, and a mutator, or set method. Relationships are represented by collection valued properties. These relationships are typically bidirectional, with many-to-many relationships being supported by collections on each side of the relationship. Figure 5.4 illustrates how the one-to-many relationship between between a customer and a number of accounts would be represented by entity beans.

Note how the home interface, only shown for the customer entity, can be used to locate existing- and create new entities. More importantly, for CMP beans, it is not necessary to provide implementations for any of the methods, they are simply declared abstract, and the private fields are omitted. The container provides all the necessary functionality to query the underlying database and ensure that the interface works as expected. The database is automatically updated whenever collections are manipulated or a mutator is called. Further, CMP beans can have a significant performance advantage over hand crafted database interactions, due to entity caching and preloading.

University of Pretoria etd – Peer, E S (2005)

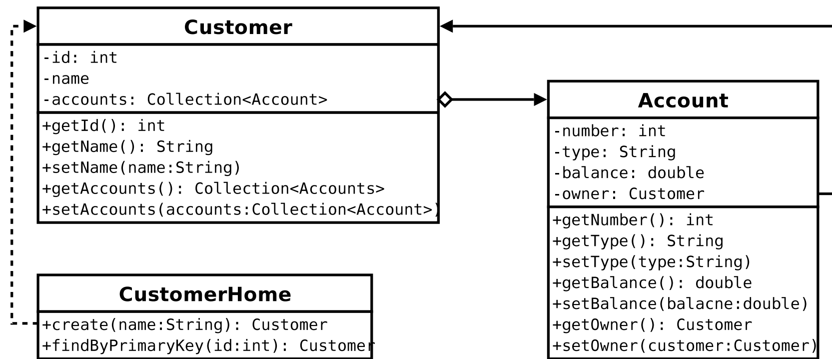


Figure 5.4: EJB Entity Relationship

5.3.2 Application Layer

Three types of application layer EJBs exist, message driven beans, stateless session beans and stateful session beans.

Message driven beans provide an asynchronous interface for clients accessing application layer objects via the Java Messaging Service (JMS), typically using XML based messages. A message driven bean's interface consists of a single on message method, which must unpack the message and do something sensible with it; perhaps calling other application layer beans or sending off other messages as a result.

Session beans typically present a session *Facade* [4] (refer to Section 3.2.4) to clients, which only exposes those parts of the system which are interesting to a given client. Some clients may require application state to be stored over multiple synchronous requests. For example, a shop application would need to store the contents of a shopping cart for the duration of the user session. A J2EE application server container automatically handles session state by creating a new instance of a stateful session bean for each client. Sessions that do not require state should use stateless session beans, enabling the container to share a single instance amongst multiple clients, if it is more efficient to do so.

Having the server container manage message and session beans means that applications can easily be scaled up over multiple servers. For load balancing, an application server simply needs to ensure enough stateless beans are instantiated for a particular service to saturate the given hardware. Fault tolerance is achieved by ensuring that stateful beans are distributed to one or more backup servers. All this is achieved without the explicit knowledge of the developer, making it easier to write scalable, fault tolerant

University of Pretoria etd – Peer, E S (2005)

applications.

5.3.3 Presentation Layer

The presentation layer presents a developer with many choices. The interface presented to users might be a heavyweight rich client implemented using the JFC or it may be a highly accessible web application powered by a combination of any number of existing presentation tier frameworks. For example, Struts[13] with JSP (Java Server Pages)[14] or the recently released JSF (Java Server Faces)[15] framework. It could even be a very thin layer that simply forwards messages to an underlying message driven bean, perhaps implementing an electronic mail interface.

GUIs should make use of the Model View Controller (MVC) [4] architectural pattern. The model, which represents data or functionality behind the user interface, is accessed via session beans in the application layer. A view is responsible for presenting its model to the user and returning control to the controller after the user takes action. The controller then determines the next view based on the current view and the action taken by the user. In the case of Struts, the controller is implemented by a single Servlet[16] which directs application flow between various views which are implemented by JSPs.

5.3.4 Deployment

The real power of J2EE stems from the ability to customise an application at deployment time without altering any source code. Depending on the application server, this deployment configuration, also known as a deployment descriptor, is usually specified in one or more XML documents. The following are some of the most important configurable aspects of J2EE applications:

- **Security:** J2EE provides a declarative security model based on the Java Authentication and Authorisation Service (JAAS)[17] specification. User and role based access rules for beans and their individual methods are declared in the deployment descriptor. The container performs run time security checks for each method

[13] http://struts.apache.org/
[14] http://java.sun.com/products/jsp/
[15] http://java.sun.com/j2ee/javaserverfaces/
[16] http://java.sun.com/products/servlet/index.jsp
[17] http://java.sun.com/products/jaas/

University of Pretoria etd – Peer, E S (2005)

call and throws a security exception if a client attempts to call any unauthorised method. The open source JBoss[18] application server provides this functionality by wrapping EJBs inside a security *Proxy* (refer to Section 3.2.5), which performs any necessary checks before delegating requests to the actual bean.

- **Entity Relational Mapping:** Even though the container can provide the implementation for database interactions, it is still necessary to inform the container about the type of database, along with table and column names onto which entities are mapped. Further, the entity methods that participate in relationships need to be declared.

- **Transactions:** EJB containers are capable of providing full ACID (Atomicity, Consistency, Isolation and Durability) transaction support [30]. Transaction boundaries are specified in the deployment descriptor for beans and methods. For methods, a transaction is opened at the start of a method call and is closed again when the method exits normally. If an EJB exception is thrown then the transaction is rolled back with no side effects. The Container may also perform deadlock detection and roll back transactions that cause deadlock. The isolation level of transactions is typically also configurable. Transaction support is also provided using a *Proxy* in JBoss.

- **Application Server Configuration:** The configuration, pertaining to a given application, for the application server is usually also specified in the deployment descriptor. For example, the caching and preloading behaviour for entities is configurable in JBoss. Clustering strategies and other performance related settings, such as bean instantiation policies, can also be configured.

5.4 XDoclet

XDoclet[19] is a free attribute oriented programming tool, which can be used to generate artifacts from annotations embedded as special Javadoc comments in source code.

XDoclet is an invaluable tool for EJB developers, enabling them to automatically generate any required interfaces and deployment descriptors directly from an annotated

[18]http://www.jboss.org
[19]http://xdoclet.sourceforge.net

University of Pretoria etd – Peer, E S (2005)

implementation class for an EJB. For example, to mark a method for inclusion in the application interface, a developer need only include an `@ejb.interface-method` annotation in the Javadoc comments preceding the method. Declaring JAAS access rules for a method can be achieved by prefixing the method with an `@ejb.permission` tag followed by the appropriate user or role based permissions. Similar tags are defined for declaring entity relation mappings, transaction boundaries and application server specific configurations.

The recent syntax enhancements for annotations in Java 1.5 means that future versions of XDoclet may move their annotations out of Javadoc comments into the actual code. An advantage of proper annotations will be the ability to query these attributes using the standard Java reflection API. For example, it would be possible to query security annotations before calling a method, where currently the only way to determine these permissions is to attempt the operation and catch the security exception that might be thrown.

XDoclet is more general than simply an EJB tool, with tags defined for various other applications, including the Spring framework, Hibernate, JDO, Axis, Struts and JSF amongst many others.

5.5 JUnit

Unit testing is the practice of performing automated tests on units of code, typically testing the behaviour of the public interface of individual classes. The fact that the tests are automated is the most important factor. Automated tests are easy to run, meaning they can be scripted into the build process to give early warning of something getting broken during code maintenance. This safety net gives developers more confidence to work on the code, particularly when maintaining code they did not write, since even small changes can be tested against the entire test suite, rooting out any unexpected side effects. If the tests pass then chances are nothing got broken, assuming the tests are representative of the required behaviour.

Tests should be maintained in tandem with the code. The XP (eXtreme Programming) paradigm [11] advocates writing a complete test suite for a unit before writing its implementation, so that passing all the tests becomes the measuring standard for the completeness of the implementation.

University of Pretoria etd – Peer, E S (2005)

Unit tests serve another important purpose, namely documentation. Unlike comments which can easily fall out of synchronisation with the code implementation, automated tests immediately show any discrepancies that need to be addressed. Unit tests implicitly document the intended behaviour of the code, since that is precisely what they are testing.

JUnit[20] is a free framework that facilitates unit testing in Java. Figure 5.5 illustrates the Test *Composite* (refer to Section 3.2.2) employed by JUnit.

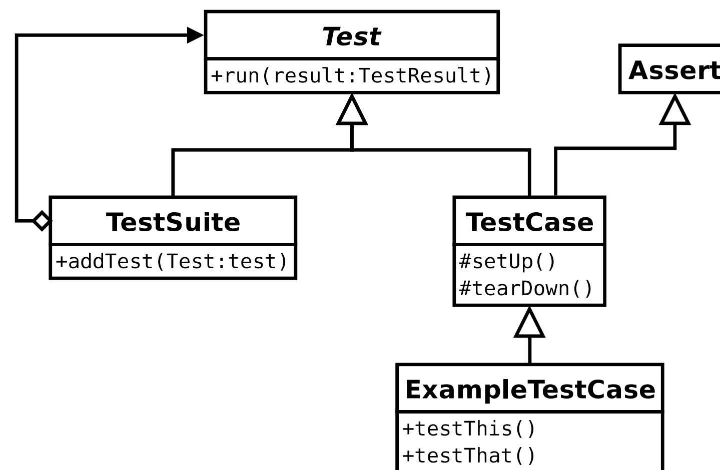


Figure 5.5: JUnit Composite Test Framework

Graphical and command line tools which are capable of executing a `Test`, which may be an entire suite of tests, are provided. The `TestSuite` composite can be used to build a hierarchy of test cases that mirrors the package hierarchy of the software, with one `TestCase` dedicated to each class being tested. Adding new tests for a class is made trivial, only requiring the developer to write another method prefixed with the string “test”. The JUnit framework uses the Java reflection API to introspectively call each test method in turn. The `setUp()` and `tearDown()` methods are called by the framework before and after each test method respectively. These methods can be used to configure a fixture that is available to all the test methods. Various methods for testing assertions are inherited in via the `Assert` class. Assertions that fail are gathered into a test result and are reported by the tool after all the tests have been executed.

[20]http://www.junit.org

University of Pretoria etd – Peer, E S (2005)

5.6 Summary

XML and Java were introduced as languages used in the development of CILib and CiClops. In particular, Java was motivated as an appropriate choice of implementation language due to its platform and vendor independence, garbage collection, the Java foundation classes, good tool support and high performance.

Next, an overview of the J2EE framework, which is used by CiClops, was presented. J2EE provides powerful services, such as container managed persistence and transactions, to applications built using EJBs.

Finally, the XDoclet tool and its role in easing EJB development was discussed, followed by a brief introduction to the JUnit testing framework.

University of Pretoria etd – Peer, E S (2005)

Chapter 6

Computational Intelligence Library

"Ah, well, I am a great and sublime fool. But then I am God's fool, and all His work must be contemplated with respect." — Mark Twain

CILib (Computational Intelligence Library) is a software framework designed to accommodate scientific research in Computational Intelligence, providing implementations for many CI algorithms, problems definitions and a simulator for conducting experiments. In order to maximise collaboration and solicit third party peer review, CILib is published under the GNU GPL (refer to Section 4.1.6) and is available for download from SourceForge[1]. The following high level project goals were identified:

- **Flexibility:** Design patterns should be exploited to create a reusable framework capable of supporting the complexity of the CI field. Whenever possible, hybrid algorithms and new functionality should be achieved by composing various existing classes in a pluggable fashion.

- **Experimentation:** The framework should facilitate scientific experimentation, making it possible to measure any property of an algorithmic simulation. Different simulations, in terms of various class compositions and algorithm parameters, should be configurable at run time without making changes to the source code.

- **Efficiency:** It is commonly accepted that developer time is more expensive than CPU time, however, CI algorithms can be very computationally intensive. Thus,

[1] http://cilib.sourceforge.net

University of Pretoria etd – Peer, E S (2005)

a scientific simulation framework may at times have to trade off clean OO design against improved performance.

- **Separability:** There should be a clean separation of algorithms and problems, so that any algorithm can be applied to any suitable problem. Further, algorithms should be independent of any scientific simulation and measurement components, so that algorithms can also be used in non-research applications.

- **Reliability:** The open source peer review process should increase the probability of any software errors being found and corrected. A clean OO design and extensive unit testing should be used to further reduce any chance of errors.

- **Collaboration:** The framework should maximise collaborative opportunities. By sharing a common open source code base, researchers may be more aware of what others are doing and can reuse parts of the framework developed by others without reinventing the wheel. Good documentation should be provided to keep the barrier to entry as low as possible.

Section 6.1 recommends some coding conventions for CILib developers. Following that, the implementation details of CILib are covered in Section 6.2. Collaborative contributions to CILib are mentioned in Section 6.3. Finally, some limitations of the framework are discussed in Section 6.4.

6.1 Coding Conventions

To date, no coding conventions have been enforced on contributions to CILib, however, it is the recommendation of this work that developers adopt the Java coding conventions published by Sun Microsystems [57], which reflect those presented in the Java Language Specification [59]. A single coding standard is necessary despite the fact that developers may have different stylistic preferences. Adopting a standard results in code that can be unambiguously understood and easily read, since developers know what to expect even though it may not be their personal preference. This is particularly important in an open source context, where the source code itself is a primary means of communication between developers.

University of Pretoria etd – Peer, E S (2005)

The specification outlines some guidelines pertaining to the commenting of code. Java supports two types of comments, namely implementation comments and doc comments. Implementation comments apply to the implementation details of the code itself, while doc comments can be extracted as separate documentation independent of the code using the Javadoc tool [33]. Doc comments should be used to describe the purpose and function of interfaces, classes and methods in an implementation independent way. Implementation comments should be kept to a minimum, the code should rather be made as self documenting as possible, since comments can easily fall out of synchronisation with the code. Good doc comments, design patterns, unit testing and careful consideration of the naming of methods and identifiers should be sufficient documentation for any developer to understand the implementation. If the implementation is not self documenting then there is probably something wrong with the design that needs to be fixed. In the case of implementations of research algorithms, a proper reference to any pertinent articles should be provided in the doc comments for the implementing class. JUnit tests (refer to Section 5.5) should be provided whenever possible. Unfortunately, the stochastic nature of many of the algorithms in CILib means that a researcher is not likely to know what its acceptable behaviour should be, which is typically what is being researched in the first place.

Further, the specification lists naming conventions. A convention of prefixing a package name with the reversed Internet domain of the package owner should be followed, to ensure there are no conflicts in the package namespace, hence CILib packages fall in a hierarchy under `net.sourceforge.cilib`. Interface and class names should be mixed case with the first letter of each word capitalised. Abbreviations should be avoided. Methods and variables follow the same convention except that the first character is lower case. Constants should be written in upper case with underscores as word separators.

Finally, the document specifies formatting conventions. A particularly contentious issue, particularly with C/C++ developers, is the Java convention of having opening braces for blocks at the end of the line that defines the block. Closing braces should be indented to align with the statement, method or class that forms the start of the block. A level of indentation is defined to be four spaces. Further, a space should occur between keywords and parentheses, after commas in an argument list, between binary operators, except the class membership operator, between expressions in a `for` statement and after a type cast. Blank lines should also be used liberally to group related sections of code,

University of Pretoria etd – Peer, E S (2005)

especially between blocks and methods. Lastly, parentheses should be used to group arguments in complicated expressions to make them easier to read, instead of relying on the reader's knowledge of operator precedence rules.

6.2 Implementation Details

CILib's implementation is heavily based on design patterns (refer to Chapter 3) to maximise its flexibility. The type system used for representing problem domains is discussed in Section 6.2.1. CILib's representation for problems and implementation of algorithms are discussed next in Sections 6.2.2 and 6.2.3 respectively. Section 6.2.4 demonstrates the framework's facilities using particle swarms as a specific example. Stopping criteria for iterative algorithms is handled in Section 6.2.5. Finally, scientific experimentation is supported by measurements, in Section 6.2.6, and a simulator, which is covered in Section 6.2.7.

6.2.1 Domains and Types

Domains define a type system based on a string representation of a data type. A partial grammar for describing types consisting of combinations of bits, integers and real values is provided in Figure 6.1. These domains are used to describe, amongst other things, the search domains of computational intelligence problems. For example, a multidimensional real valued optimisation problem, as described in Section 2.1.1, would have a domain representation of "R^N", where N is replaced with the actual dimension of the problem. A genetic program which searches a tree space (refer to Section 2.3.2) might operate on a domain characterised by a description of the valid non-terminal nodes, a list of terminal symbols and a maximum tree depth.

Vectors of any given type are represented by composite and compound domain components. A compound represents a repetition of a type, while a composite is used to represent a mixture of different types. Further, compound components can represent variable length vectors.

For example, the compound type "Z^5" represents 5 dimensional vectors of integers. Equivalently, the composite "[Z,Z,Z,Z,Z]" represents the same 5 dimensional vector type. Compound domains permit constructs such as "Z^3~2", which represents an integer

University of Pretoria etd – Peer, E S (2005)

domain	::=	type | compound | composite
composite	::=	’[’ domain { ’,’ domain } ’]’
compound	::=	domain ’^’ int [’~’ int]
type	::=	’B’ | ’Z’ [’(’ [int] ’,’ [int] ’)’] | ’R’ [’(’ [real] ’,’ [real] ’)’]
real	::=	int [’.’ digit_sequence] [(’e’ | ’E’) int]
int	::=	[’+’ | ’-’] digit_sequence
digit_sequence	::=	digit { digit }
digit	::=	’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

Figure 6.1: Partial Domain Grammar

vector type of length ranging between 3 and 5 inclusive. That is, the second number which follows the tilde symbol, corresponds to the amount of slack permitted by the type. A composite type permits constructs such as “[R,R,R,Z,Z]”, or equivalently “[R^3,Z^2]”, which represents a mixed vector type of 3 real values followed by 2 integers. Note that compound and composite types can be arbitrarily nested.

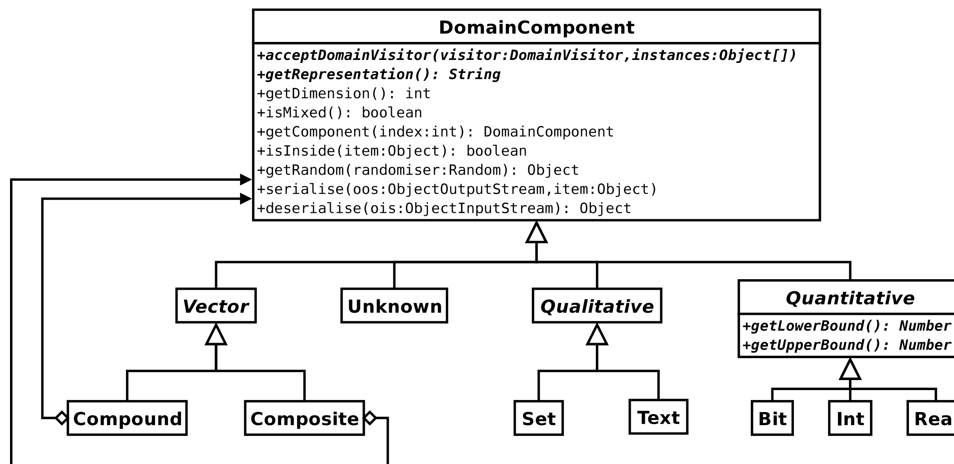


Figure 6.2: Domain Composite/Interpreter

Figure 6.2 illustrates how types are mapped into a *Composite* (refer to Section 3.2.2) object structure. The object structure can also be considered to be an instance of the *Interpreter* pattern, in Section 3.3.1, since the class hierarchy, although it has slightly more structure, to a certain extent mirrors the grammar. A *Singleton* (refer to Sec-

University of Pretoria etd – Peer, E S (2005)

tion 3.1.4) component factory is responsible for parsing domain strings and constructing their corresponding domain description, in terms of a hierarchy of domain components.

Types are divided into three categories: the composite and compound vector types which have already been discussed; qualitative types which represent ordinal or nominal data [106]; and quantitative types which represent numeric data. The quantitative types have the option of declaring bounds. In the grammar, these bounds are represented between parentheses. For example, a multi-dimensional search space bounded by $[-1, 1]$ in each dimension is represented by the string "R(-1,1)^N", where N is the dimension of the search space. Alternatively, a composite vector can be used to represent different bounds in each dimension. Lower and upper bounds are taken to be $-\infty$ and ∞ respectively if they are not specified.

The string representations for integer, real value, and vector types have already been discussed. Bits are represented by the string "B". String types are represented by the text component with representation "T". Sets are represented by the prefix "S" followed by a comma separated list of valid elements between braces. Graphs and trees might, in future, be represented by a prefix "G" followed by a list of terminal and non-terminal node descriptions. Any type which is not incorporated into the domain hierarchy is allocated an unknown type with representation "?".

The most important function of the domain hierarchy is producing random instances of a type, which are used as initial points in search spaces for optimisation algorithms. Care has been taken to return the most efficient concrete instance of any given domain. For example, a single bit returns a `java.lang.Byte` with a value of one or zero, but a vector of bits returns a `java.util.BitSet` instead of a memory inefficient array of bytes. Vectors of integers and real values return arrays of their respective `int` and `double` primitive types, which provide for the most efficient processing without polymorphic object overheads. Mixed composites return an array of generic objects containing as elements the largest possible groupings of more specific types. For example, the domain string "[R^30,B^20]" would result in a domain hierarchy that returns instances of the form `Object[] { double[30], BitSet }`, where the `size()` method of the bit set has been overridden to return the logical number of bits, in this case 20, as opposed to the actual number of bits used by the implementation. All a client of the domain hierarchy need do is cast the result into the type it expects. Domain validators are provided in the `net.sourceforge.cilib.Domain.Validator` package in order for a client to test those

University of Pretoria etd – Peer, E S (2005)

expectations before performing any casts. Clients that support multiple domains must query the domain hierarchy to determine what instances of the domain will look like and deal with them appropriately.

Beyond generating random instances inside a domain, a client may query: the dimension of a domain; whether a multi-dimensional domain contains mixed types; whether a given instance falls within the domain; and in the case of quantitative types, the bounds. The methods to get the dimension and the ith component of a vector present a flattened view of nested compound and composite vectors, so that indexing components does not need to take into account any effect of nesting. This means, equivalent domains, such as "[R^10,R^20]", "[R^20,R^10]" and "R^30", are identical from the client's perspective, even though they all have different hierarchical structures.

Measurements (refer to Section 6.2.6) are another aspect that require domain information, since they can be of any type and a common measurement interface is desired. The serialisation methods are provided so that instances of a domain, particularly measurements, can be stored and retrieved in a more space efficient fashion than the standard Java serialisation method.

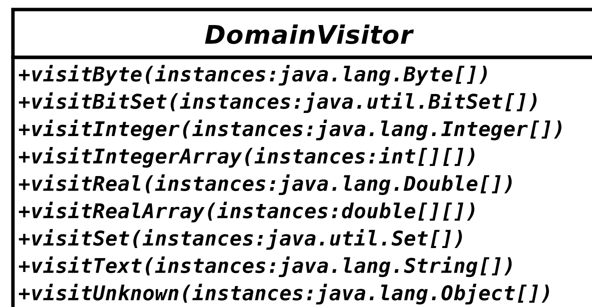


Figure 6.3: Domain Visitor Interface

Unfortunately, there are some design flaws in the domain strategy presented here. The most important being that clients cannot treat type instances in a uniform way, because the types described by a domain do not share a useful polymorphic interface. That is, a client needs to explicitly know how to deal with every type of domain that it supports. For example, an algorithm capable of dealing with both real valued and bit vectors needs to query the domain, directly or using validators, and conditionally execute one of two branches, one for each type, even though both branches probably

University of Pretoria etd – Peer, E S (2005)

contain similar logic. The domain *Visitor* (refer to Section 3.3.6) interface presented in Figure 6.3 alleviates this problem slightly by providing a cleaner interface for clients, but it is still clumsy and confusing, since an array of instances on which the visitor operates needs to be passed around, and its implementation is currently not very speed efficient.

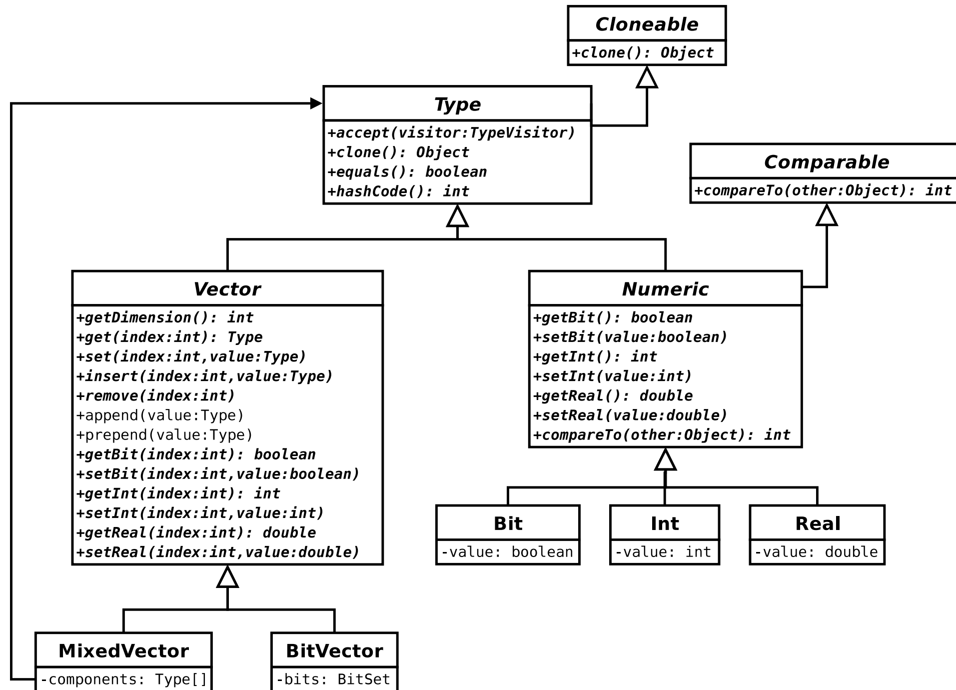


Figure 6.4: Partial Type System

The proper solution, assuming the object in-lining technology mentioned in Section 5.2 gets incorporated into future compilers, is to implement a polymorphic type system. The JFC already provide for numeric types using the `java.lang.Number` hierarchy. Unfortunately, this hierarchy consists of immutable numeric types, requiring object creation and collection overheads for even simple arithmetic operations, which are likely to be executed in tight loops by many algorithms. Thus, work has begun on the polymorphic type system presented in Figure 6.4.

Note that a client need only care whether it works on a vector or non-vector type, which is fine, since, for the most part, it will be one or the other exclusively. A client that does not care about the specific numeric type with which it works can simply utilise whichever units are most convenient. Those clients that do need to differentiate them, can make use of a more traditional *Visitor* (refer to Section 3.3.6) interface which does

University of Pretoria etd – Peer, E S (2005)

not require instances to be passed around as an additional parameter. Further, bit vectors and other arbitrary vectors present a uniform interface, meaning clients will not need to treat vectors of bits as a special case, while still benefiting from the storage efficiency of a bit set.

The problem with the type system presented in Figure 6.4 is that domain information cannot safely or efficiently be incorporated into the hierarchy. Bounds on numeric types and constraints on vectors can be cleanly implemented using *Decorators* (refer to Section 3.2.3), however, the extra level of indirection will have a severe performance penalty for types used in tight loops. In addition, bounds information which would be shared by a compound domain must be inefficiently stored for each individual vector component along with an additional memory reference. Further, although it may seem like a good idea to store the domain information implicitly in the type system, because clients have the freedom to modify the type, the integrity of the domain information may be compromised. For example, if the type system keeps track of the fact that it is an instance of "R^N" simply by virtue of the fact that it is a vector of real values, then a client which changes a component into an integer would alter the domain as a side effect. Finally, while serialisation can be supported in the type system relatively cleanly, deserialisation and generating random instances within a specified domain become very clumsy, since the type instance which would contain the necessary information does not yet exist.

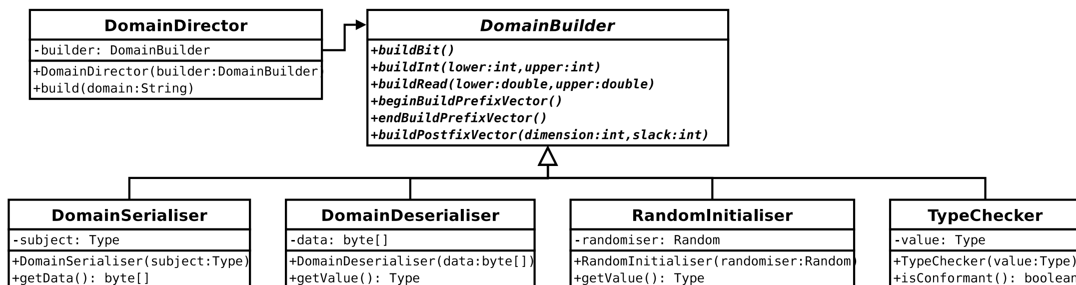


Figure 6.5: Domain Builder

The limitations of the type system just described seem to indicate that a parallel domain hierarchy still needs to be maintained, however, another possibility that is currently being investigated is the use of the *Builder* pattern (refer to Section 3.1.2) as illustrated in Figure 6.5. Instead of storing a domain hierarchy explicitly, only the original domain

University of Pretoria etd – Peer, E S (2005)

string is stored and different concrete builders are used to realise the same functionality. For example, a type checker can be used to determine whether a given type instance conforms to the domain string passed to the builder.

6.2.2 Problem Classes

Figure 6.6 demonstrates how the broad problem classes defined in Section 2.1 can be represented in software. The optimisation problem interface is characterised by: a domain, which defines the search space; and a fitness function, which evaluates the goodness of a given solution. Route optimisation problems, such as the TSP (refer to Section 2.1.2, are simply characterised by the graphs that define their routing networks. Both supervised and unsupervised learning problems are characterised by their data sets. In the case of supervised problems, patterns consist of an input part and a target part, which is encapsulated by the `Pattern` type. Both provide traversals of the data set using an *Iterator* (refer to Section 3.3.2). Patterns may conform to different domains, which are accessible via the respective problem interfaces. Additionally, unsupervised problems provide information about the number of clusters inherent in the data set, or alternatively, the constant `UNKNOWN_CLUSTERS` if such information is unknown.

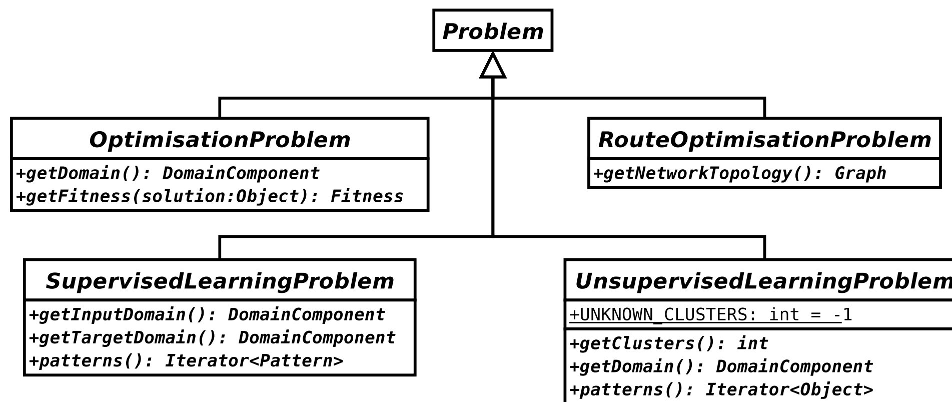


Figure 6.6: Problem Interfaces

These problem interfaces need to be implemented by concrete problem classes that take into account any context specific to a given situation. Concrete problems that are defined in terms of data sets, which can be true of any type of problem, can access their data via the `net.sourceforge.cilib.Problem.DataSet` interface. The data set

University of Pretoria etd – Peer, E S (2005)

interface does not enforce any structure on data. It simply provides input stream and byte array views of the raw data. The responsibility of interpreting the data falls upon the concrete problem implementation. Some problems may have their data represented as a structured XML document, while others may be constrained to operate on less structured data defined by the context of the problem. For example, a clustering problem defined for banking data may be constrained to the data format utilised by the bank's database. Each new application may require another concrete problem description, which encapsulates the characteristics of the application domain, presenting itself in terms of one of the general problem interfaces. The general framework will need to be extended as new problems arise which cannot fit into the model presented in Figure 6.6.

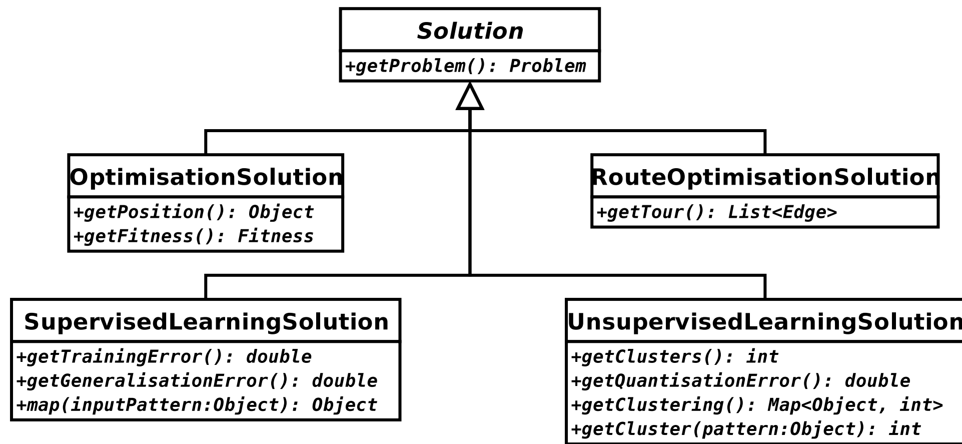


Figure 6.7: Solution Classes

Figure 6.7 shows the solutions corresponding to the given problem interfaces. First and foremost, solutions must exist within the context of some problem, hence there is a method providing access to their problems. The solution to an optimisation problem is characterised by a position and its fitness. Route optimisation solutions consist of an ordered list of the edges of the graph that form the optimal tour. The learning problems have solutions that are characterised by a model that fits the data. In the case of supervised problems, the model provides a method to determine the mapping for unseen input patterns, while an unsupervised model provides a method to determine the cluster index for an unseen pattern and access to the clustered training data. Both provide methods for determining the accuracy of the learned model.

Figure 6.8 illustrates some further specialisations of optimisation problems. Multi-

University of Pretoria etd – Peer, E S (2005)

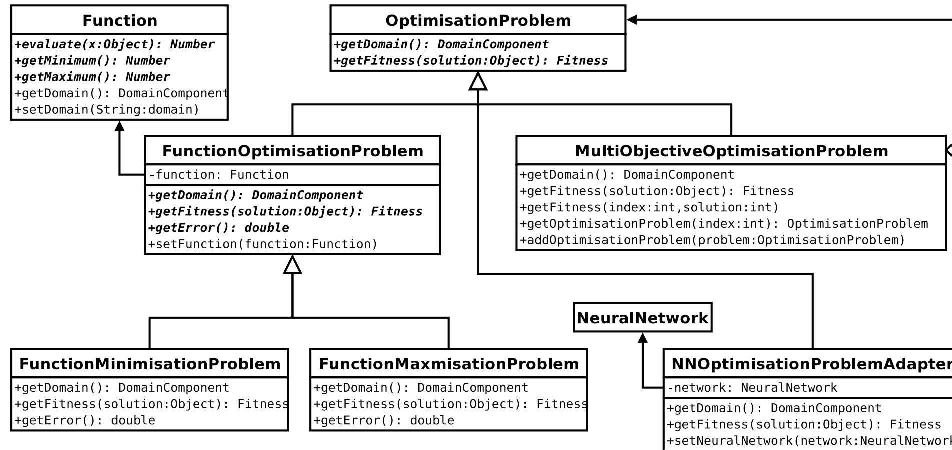


Figure 6.8: Optimisation Problems

objective optimisation problems turn the hierarchy into a *Composite* (refer to Section 3.2.2) so that a multi-objective problem still presents a single objective view, while permitting access to individual objectives for algorithms that support multi-objective optimisation. While the neural network code is currently in an incomplete state, it is easy to imagine a problem *Adapter* (refer to Section 3.2.1) that enables neural network training by means of an optimisation algorithm. In a research context, it is desirable to test optimisation algorithms on various benchmark functions. For this reason, an extensive set of benchmark functions is provided in the `net.sourceforge.cilib.Functions` package. Another *Adapter*, the `FunctionOptimisationProblem` class provides the glue between the optimisation problem interface and a benchmark function. Function optimisation is further specialised into minimisation and maximisation problems, which respectively minimise and maximise a benchmark function.

Earlier versions of CILib treated fitness as a single `double` value, which was negated in the case of function minimisation problems, so that larger values of fitness always indicated a more optimal solution. This simplistic approach had limitations when working with constrained optimisation problems, since constraint handling code needs access to the unaltered function surface. The fitness hierarchy in Figure 6.9 was introduced to solve this problem. Fitnesses now implement the comparable interface so that a fitness, when compared, performs the necessary transformation for minimisation problems, while still leaving the original function value accessible. Thus, fitness is always maximised, even for

University of Pretoria etd – Peer, E S (2005)

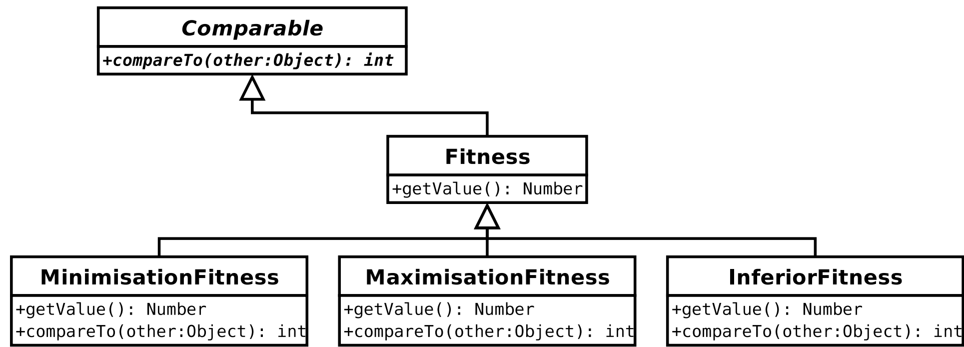


Figure 6.9: Fitness Classes

minimisation problems. The inferior fitness class always compares worse than other fitnesses, and is ideal for initialising the fitness of individuals in a population based search algorithm that have not yet been evaluated. Switching to a fitness type hierarchy also added the flexibility to handle discrete optimisation problems in a uniform way.

6.2.3 Algorithms

The `Algorithm` class, depicted in Figure 6.10, implements behaviour common to all iterative CI algorithms. These responsibilities include handling stopping criteria, notification of algorithm events, presenting an interface for threads and any other common house-keeping tasks.

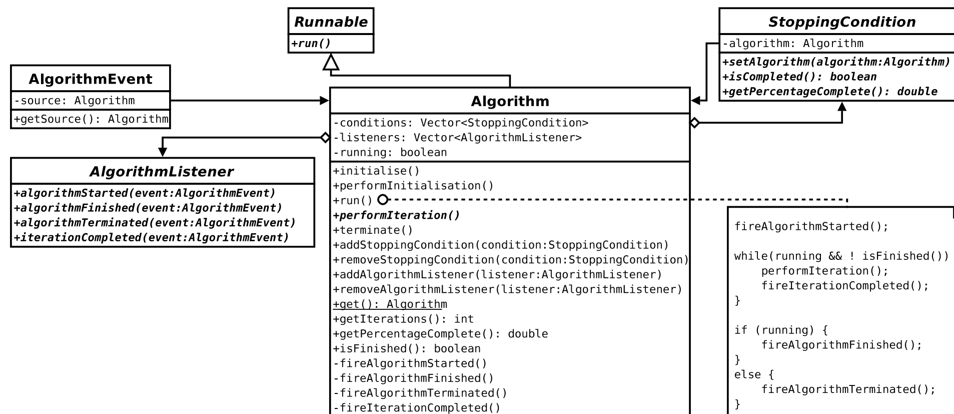


Figure 6.10: Algorithm, Stopping Conditions and Events

University of Pretoria etd – Peer, E S (2005)

The `run()` method is an example of a *Template Method* (refer to Section 3.3.5), which delegates the responsibility for executing a single iteration of the algorithm to a subclass that must override the abstract `performIteration()` method. The `initialise()` method is also a *Template Method*, performing initialisation tasks common to all algorithms before deferring to the `performInitialisation()` method, which is responsible for any algorithm specific initialisation, if necessary.

Stopping conditions monitor the progress of an algorithm, providing two methods to measure this progress. Firstly, the `isCompleted()` method is called for every iteration to determine when execution of the `run()` method should finish. Second, the `getPercentageCompleted()` method, which is typically more expensive to calculate, is primarily intended for updating progress indicators in a user interface, but can also be used as a value that increases linearly (depending on the particular stopping condition being used) over the execution duration for those algorithms that need it. Multiple conditions are accommodated simultaneously by maintaining them in a list, so that `isFinished()` returns true as soon as any one of the stopping conditions fires and `getPercentageComplete()` returns the average over all the conditions.

The event interface, which is an extension of the *Observer* pattern (refer to Section 3.3.3), is used to notify a list of observers, or listeners, whenever an algorithm, is started, finished, terminates early or completes an iteration. Unlike the basic *Observer*, which provides a listener with very little information about the subject, the event interface provides information about the kind of event that occurred as well as the source of the event, enabling many-to-many relationships between algorithms and listeners.

The class scope `get()` method returns a thread local instance of the algorithm which is currently executing. This provides a global method for objects lower down in the object reference graph to access the root algorithm class, so that they can navigate from that point to any required object. This contributes to keeping many interfaces simpler, reducing the need to pass additional objects around that are only used in rare circumstances. Also, it enables objects to access parts of the reference graph that were unforeseen in the design of certain interfaces. Unfortunately, there is a major problem with this approach, which is yet to be resolved, an object lower in the hierarchy may not know how to navigate the reference graph, since classes may be composed differently at run time.

An interface for accepting problems is not specified by the `Algorithm` class, since

University of Pretoria etd – Peer, E S (2005)

only its subclasses know what kind of problems they can be applied to. Figure 6.11 illustrates how optimisation problems fit into the CILib framework, showing that any algorithm implementing the `OptimisationAlgorithm` interface can be applied to an `OptimisationProblem`. For example, since PSO implements `OptimisationAlgorithm`, it can be applied to solve optimisation problems. Algorithm interfaces for other types of problems, such as routing or learning, can be implemented in a similar fashion. Having an algorithm interface for each type of problem enables an algorithm to be selective about the problems it can be applied to. Also, an algorithm may implement any number of these interfaces simultaneously, one for each type of problem that it can be applied to. For example, a feed forward neural network (refer to Section 2.2.1) would accept a `SupervisedLearningProblem`, while a SOFM (refer to Section 2.2.4) would accept unsupervised learning problems in addition to supervised learning problems.

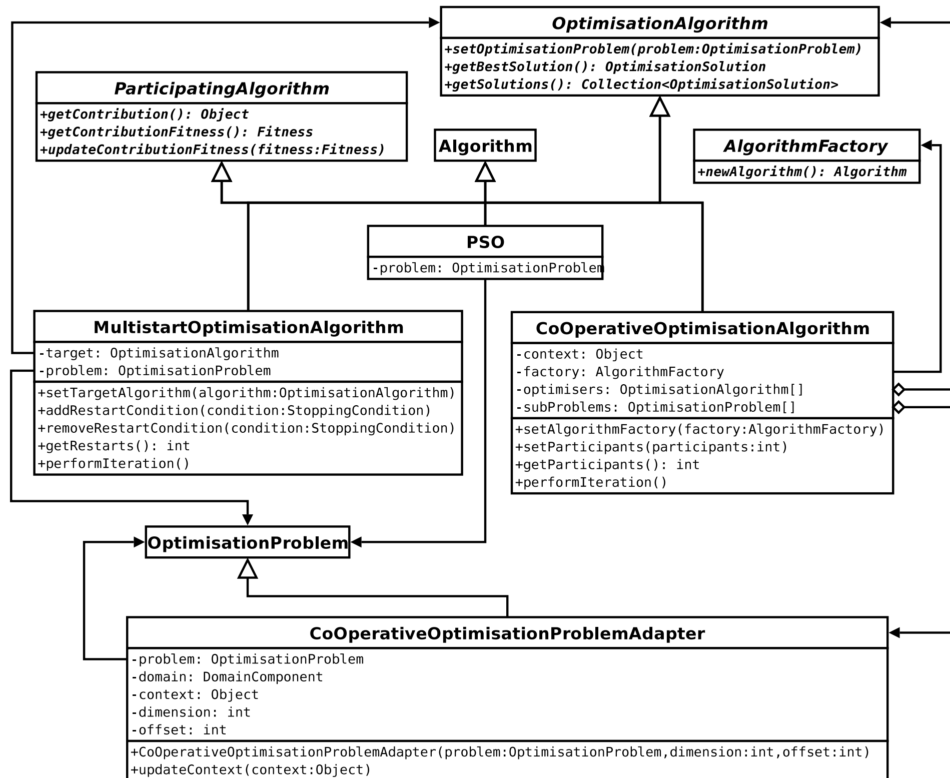


Figure 6.11: Optimisation Algorithms

Focusing again on optimisation problems, it is clear that any optimisation algorithm, including EC algorithms such as GAs, and not only PSOs can be implemented within

University of Pretoria etd – Peer, E S (2005)

the CILib framework by simply implementing the `OptimisationAlgorithm` interface, however, care should be taken to factor out any commonalities so that they can be reused and composed in various ways.

For example, the multi-start PSO (MPSO) [113] calls for restarting a PSO multiple times in order to find better solutions, since a PSO may prematurely converge onto suboptimal local extrema. By realising that this behaviour is generally applicable to all optimisation algorithms and not only PSOs, it can be factored out into a generic multi-start optimisation algorithm. The multi-start optimisation algorithm re-initialises a target algorithm whenever a restart condition is satisfied. For example, in the case of a PSO it may be appropriate to restart the algorithm whenever the average distance between particles drops below a certain threshold. This threshold would need to be captured in a stopping condition and applied to the multi-start algorithm as a restart criterion. Thus, any optimisation algorithm can have multi-start behaviour, provided a suitable restart condition can be defined. Indeed, it may be sensible to make this behaviour more general still, so that it can be applied to any algorithm as opposed to only optimisation algorithms. Such refactoring will be performed when it becomes evident how best to achieve it, bearing in mind that the multi-start optimisation algorithm needs to keep track of the best optimisation solution found during all the runs.

Coevolutionary techniques (refer to Section 2.3.6) also apply more generally than only to EC. As examples, consider the use of particle swarm optimisation instead of EC for Blondie 24 (refer to Section 2.7) or the cooperative PSO (CPSO) [113] which applies a technique used for cooperative coevolutionary GAs [91] to PSOs. The cooperative optimisation algorithm implemented in CILib, which factors this common behaviour into a more generic algorithm, only caters for optimisation algorithms that cooperate by splitting the solution vector up into smaller components. This is accomplished by a problem *Adapter* (refer to Section 3.2.1), which calculates the fitness of a smaller component of the vector in the context of the other cooperating algorithms. The cooperating algorithms, or participants, are created by the cooperative optimisation algorithm using an *Abstract Factory* (refer to Section 3.1.1), so that the type of the participants can be specified externally. Any algorithm used as a participant must implement the `ParticipatingAlgorithm` interface, which provides a mechanism for the cooperative algorithm to access the individual parts of the solution worked on by each participant. Thus, by implementing the `ParticipatingAlgorithm` interface, any optimisation al-

University of Pretoria etd – Peer, E S (2005)

gorithm, PSO, GA or otherwise (including combinations of different algorithms at the same time), can participate in a coevolutionary strategy that splits up the solution vector amongst multiple cooperating algorithms. Other coevolutionary approaches, such as sharing solutions using blackboard or having competing populations, are currently being worked on by another contributor (refer to Section 6.3). Competing populations could conceivably be implemented relatively transparently using a `Fitness` *Adapter* (refer to Section 3.2.1), which evaluates fitness relative to individuals in other populations.

6.2.4 Particle Swarm Optimisers

This section explores CILib's PSO (refer to Section 2.4.1) architecture in more detail as a demonstration of the framework's support for the implementation of an optimisation algorithm. Implementations of other algorithms, optimisation or otherwise, were not provided by the author and as such are not discussed (refer to Section 6.3 for information about other contributions).

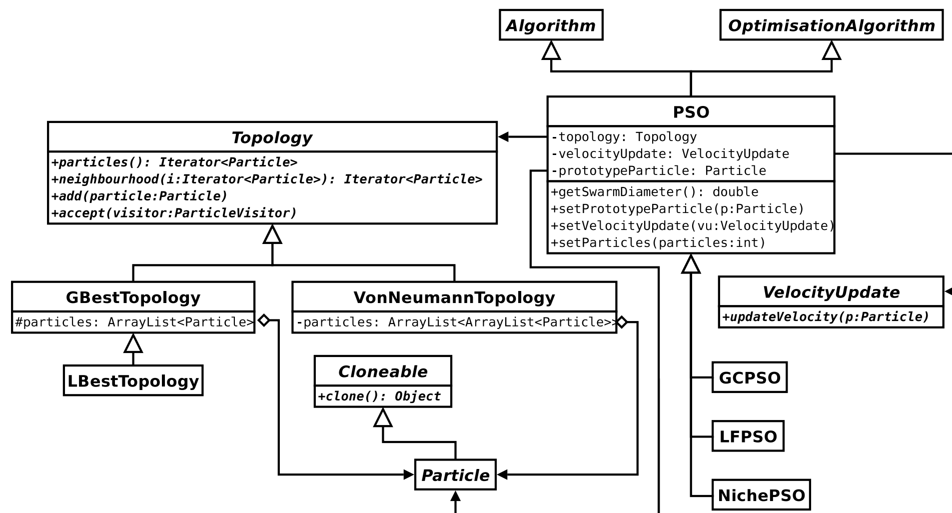


Figure 6.12: Overview of PSO Architecture

An overview of the PSO architecture implemented in CILib is provided in Figure 6.12. Particle swarms differ in terms of the neighbourhood topology of the particles and velocity update equation used to govern their trajectories. These two aspects are implemented as *Strategies* (refer to Section 3.3.4) which can be varied independently. Thus, any velocity update can be used in combination with any neighbourhood topology and *vice*

University of Pretoria etd – Peer, E S (2005)

versa.

The algorithm interface for `VelocityUpdate` is characterised by a single method, which is passed to the particle that it must update. The topology interface is more complex, exposing *Iterators* (refer to Section 3.3.2) based on the standard `java.util.Iterator` interface provided by the JFC. The PSO can use iterators to traverse all particles in the topology or only those particles within the neighbourhood of another particle, for which it must provide a pointer in the form of another iterator. Topologies in CILib are dynamic, particles can be added and removed at will. Removal of particles is achieved using the `remove()` method which is available through the iterator interface. Recently, *Visitor* (see Section 3.3.6) support was also added to topologies.

The fact that the LBest topology inherits from GBest requires some explanation, since GBest is a special case of LBest with the neighbourhood being equivalent to the entire swarm (refer to Section 2.4.1). To see why this is the case, consider that the LBest topology must implement a special *Iterator* with the ability to handle wrap-around in order to traverse the neighbourhood of any given particle. The GBest topology, however, does not require this specialised behaviour, since it can use an *Iterator* that simply traverses the whole array of particles for both the swarm and neighbourhood cases. Thus, LBest is the more specific case in terms of the implementation. The Von Neumann topology (refer to Section 2.4.1) is implemented as a two dimensional matrix, with a special neighbourhood *Iterator* that traverses the immediate particles in each compass direction.

Certain PSO algorithms require particles to store additional state or have special behaviour, an ideal opportunity to apply the *Decorator* pattern (Section 3.2.3), as illustrated in Figure 6.13. Particles may be configured differently depending on the particular type of PSO being used, but the `PSO` class is responsible for creating and initialising particles within the search space. For this reason, `Particle` implements the *Prototype* pattern (refer to Section 3.1.3), enabling the PSO to clone additional particles as necessary from a run time configured prototype. The particle positions are then initialised using the `DomainComponent` provided by the optimisation problem, by overriding the `performInitialisation()` hook provided by `Algorithm`. The inheritance depth weakness of the *Template Method* pattern (refer to Section 3.3.5) is clearly illustrated by this architecture. For example, both `PSO` and `GCPSO` may need to perform additional initialisation tasks, but only one can override the hook provided by the template method.

University of Pretoria etd – Peer, E S (2005)

Fortunately, in this case, the `GCPSO` class does not need to override it, but it is conceivable that some algorithm eventually will need to. In future, it may become necessary to store a list of initialisers in the base `Algorithm` class that must be executed in turn during initialisation, each initialiser performing the initialisation tasks specific to its algorithm.

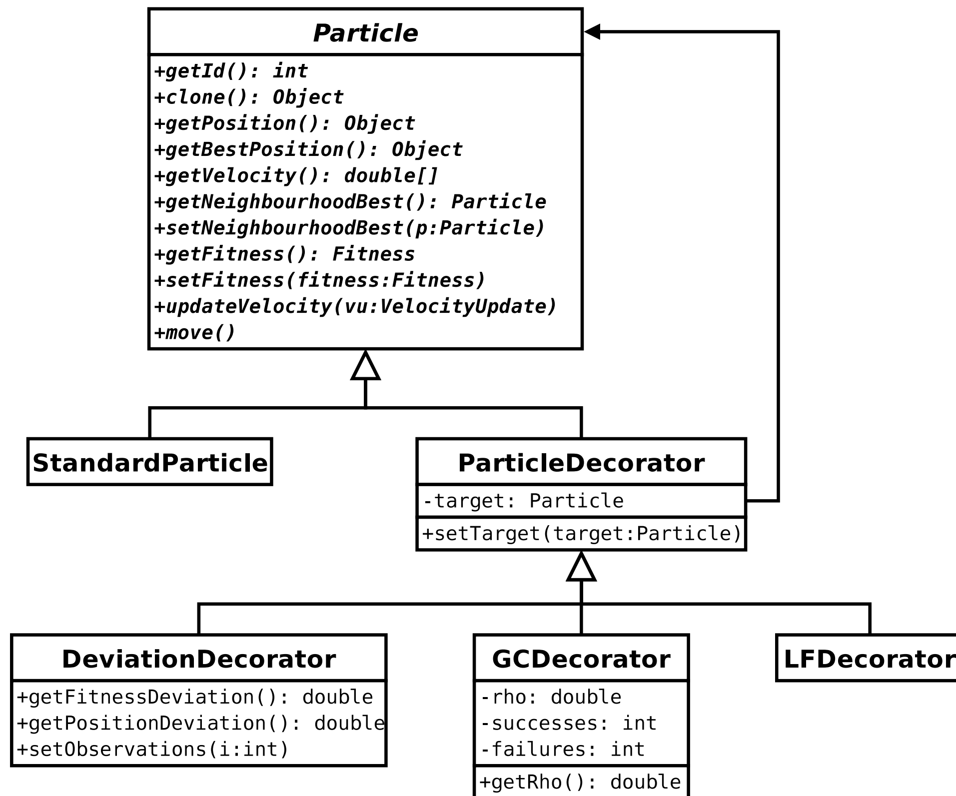


Figure 6.13: Particle Decorators

Figure 6.13 further illustrates the responsibilities of particles, each having to store its position, velocity, fitness and a reference to the best particle within its neighbourhood. In addition, each particle must be allocated a unique identifier, as a side effect of the *Decorator* pattern (refer to Section 3.2.3), so that they can be compared without regard to the dynamic nature of decorators that may be added and removed during the execution of an algorithm.

The deviation decorator, currently only used by the NichePSO [17], is used to track the standard deviations of the position and fitness of particles over time. This is an expensive operation. In terms of space, requiring a number of observations of position and fitness to be stored for each particle, and in terms of time, since these observations

University of Pretoria etd – Peer, E S (2005)

need to be updated every time a particle is moved. Thus, it makes sense to separate this functionality into a decorator that can be dynamically applied only when needed.

Both the GCPSO [114, 113] (refer to Section 2.7) and LFPSO (LeapFrog PSO, also refer to Section 2.7) algorithms implement a different velocity update equation for the neighbourhood best particles, each requiring additional state to be stored for these particles. The `GCDecorator` and `LFDecorator` decorators are used to store this additional state for their respective algorithms.

Specifically, the GCPSO velocity update performs a directed random search for the neighbourhood best particles. The step size of this search is controlled by a value, ρ (rho), which is dynamically updated based on the particle's past history. Particles which repeatedly improve their positions have their step size increased while particles that repeatedly fail to find better positions have their step size reduced.

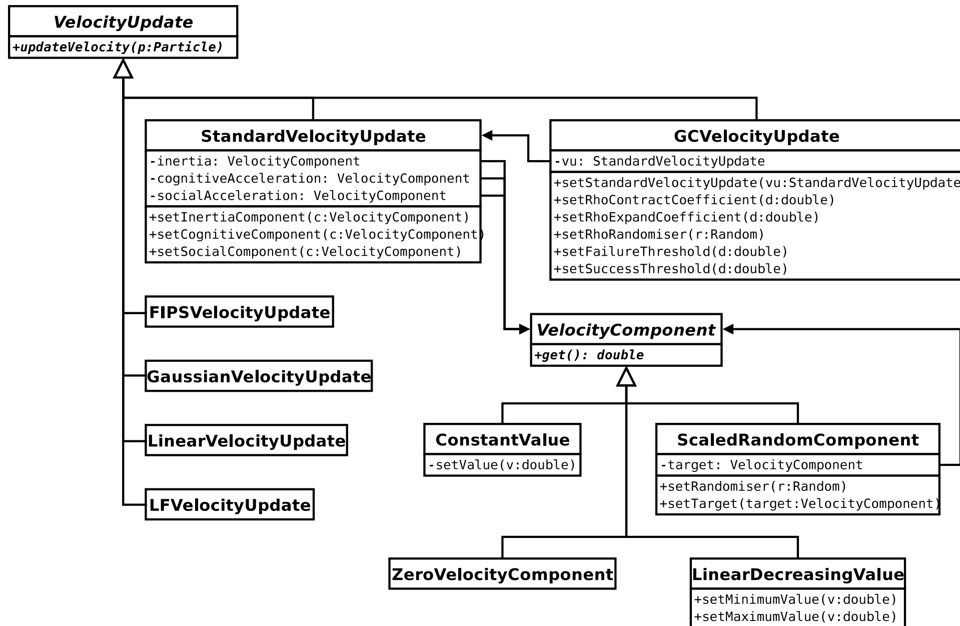


Figure 6.14: Velocity Updates

Figure 6.14 illustrates a number of velocity update *Strategies* (refer to Section 3.3.4), including the `GCVelocityUpdate` class, which implements the velocity update for the GCPSO. For non-neighbourhood best particles, it simply defers the velocity update to a standard velocity update instance. Thus, it only performs the directed random search for the best particle in each neighbourhood.

University of Pretoria etd – Peer, E S (2005)

The `StandardVelocityUpdate` class implements Equation 2.41, where the values for w, c_1r_1 and c_2r_2 are each delegated to a velocity component *Strategy* (refer to Section 3.3.4), giving a user a great deal of control over the velocity update. For instance, a linear decreasing inertia can be accomplished by simply replacing the default constant inertia component with a `LinearDecreasingValue`. By default, accelerations are implemented using a `ScaledRandomComponent` with a `ConstantValue` target, but they could be replaced with any velocity components, including a `ZeroVelocityComponent` to disable their influence, which is the equivalent of a `ConstantValue` with a value of zero.

The `LinearDecreasingValue` class is a good illustration of the usefulness of the global `Algorithm.get()` method described earlier, since it needs access to a value that scales linearly over the execution of the algorithm. A suitable value for this is available using the `getPercentageComplete()` method in `Algorithm`, however, it does not make sense to clutter the `VelocityUpdate` interface with this value, since it is not used by most velocity updates.

The remaining velocity update *Strategies* (refer to Section 3.3.4) implement a number of further PSO variants. The `LinearVelocityUpdate` class implements a variant suited for linearly constrained optimisation problems [87].

A bare bones PSO [62], which discards the notion of particle velocities and simply mutates their positions by sampling from a Gaussian distribution, is implemented by the `GaussianVelocityUpdate` class.

LFPSO is implemented by the `LFVelocityUpdate` class by following a similar approach to the `GCVelocityUpdate` class. The commonalities between the two approaches suggest that there may be merit in implementing a generic `OptimiserVelocityUpdate` which implements the `OptimisationProblem` interface, so as to replace the motion of neighbourhood best particles with the results of any `OptimisationAlgorithm` as suggested in Section 2.7.

The `FIPSVelocityUpdate` (for the Fully Informed Particle Swarm [78]) requires access to the entire neighbourhood of particles for the particle which is being updated. Since this was not foreseen when the `VelocityUpdate` or `Particle` interfaces were created, the current implementation is forced to make use of the global `Algorithm.get()` method. Unfortunately, it has to perform a linear search for the particle to obtain an iterator that can be used to access the neighbourhood, since particles do not know anything

University of Pretoria etd – Peer, E S (2005)

about the topology. This will be fixed in a later version of CILib, either by extending the `Particle` interface to make the entire neighbourhood accessible or by making particles aware of their position within a topology, by means of a *Decorator* (refer to Section 3.2.3), so that they can be located efficiently.

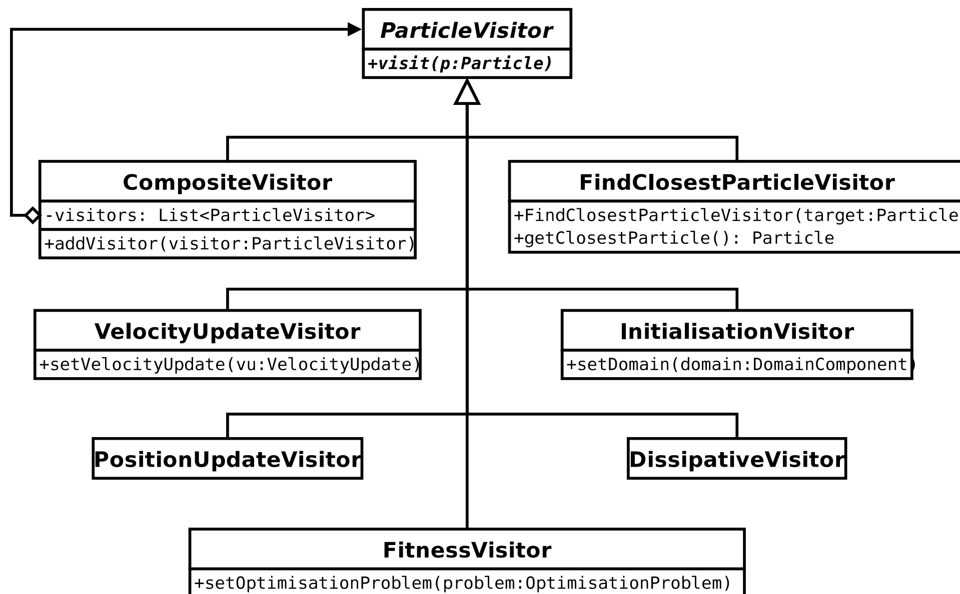


Figure 6.15: Particle Visitors

Most of the control logic for a PSO is currently in a monolithic `performItertion()` method. This is inflexible because that logic cannot be changed by simply composing different classes, but only by sub-classing the `PSO` class. Figure 6.15 represents the proposed next step in the evolution of the PSO code in CILib, the moving of parts of the internal PSO logic into external *Visitors* (refer to Section 3.3.6) which can be composed and reused in various ways. Of course, treating everything as visitors has the obvious danger that an inappropriate visitor will be used when something else is expected. Time will tell if this proposed design is a good idea or not.

The `VelocityUpdateVisitor` class is an *Adapter* (refer to Section 3.2.1) which makes any existing `VelocityUpdate` conform to the visitor interface. Perhaps velocity updates should have been implemented as visitors from the start, however, implementing velocity updates as visitors does restrict the `VelocityUpdate` interface to only accepting particles with no easy way to extend it. New velocity updates would not even need to implement the `VelocityUpdate` interface at all, but could implement `ParticleVisitor`

University of Pretoria etd – Peer, E S (2005)

directly. At this time, the global `Algorithm.get()` method appears to be a general enough mechanism for obtaining information not provided by the visitor interface.

The `PositionUpdateVisitor` class is analogous to the velocity update except that it moves the particle by altering its position instead of changing its velocity. This will have the side effect of cleaning up the `Particle` interface by removing the need for a separate `move()` method. In addition, the `GaussianVelocityUpdate` should rather be implemented as a position update, since it doesn't affect a particle's velocity at all.

The `InitialisationVisitor` class will be used to initialise particle positions based on a given domain. Delegating initialisation to a visitor enables a PSO to use an alternate means of initialisation, perhaps not even making use of the domain information, which is currently not possible.

The *Composite* (refer to Section 3.2.2) visitor is intended to allow multiple visitors to be used where only one is expected, with each visit method being called sequentially for each particle. For example, a position update visitor could be replaced by a composite containing both the position update and a dissipative visitor, which implements the logic required for the DPSO [122] (refer to Section 2.7).

Ultimately, subclasses of `PSO` will have to do far less work, perhaps as little as changing one of the visitors. This leads to the next improvement, an *Abstract Factory*, say `PSOComponentFactory`, with methods defined for creating particle, initialisation, velocity update and position update products. Thus, different particle swarm variants can be realised by merely supplying a different factory to the `PSO` class, negating the need for subclasses of `PSO` for every variant, only those that have radically different algorithms.

6.2.5 Stopping Conditions

Figure 6.16 shows some specific stopping conditions, which were discussed only generally in Section 6.2.3. Some conditions may be applied to any algorithm, while others are specific to certain types of algorithms.

For example, the maximum iterations condition can be applied to any algorithm, causing the algorithm to finish execution when the configured number of iterations has been reached. It makes use of the `getIterations()` method in `Algorithm` to determine when to fire. The condition for fitness evaluations, as another example, only applies to optimisation algorithms, which can be stopped when the objective function has been

University of Pretoria etd – Peer, E S (2005)

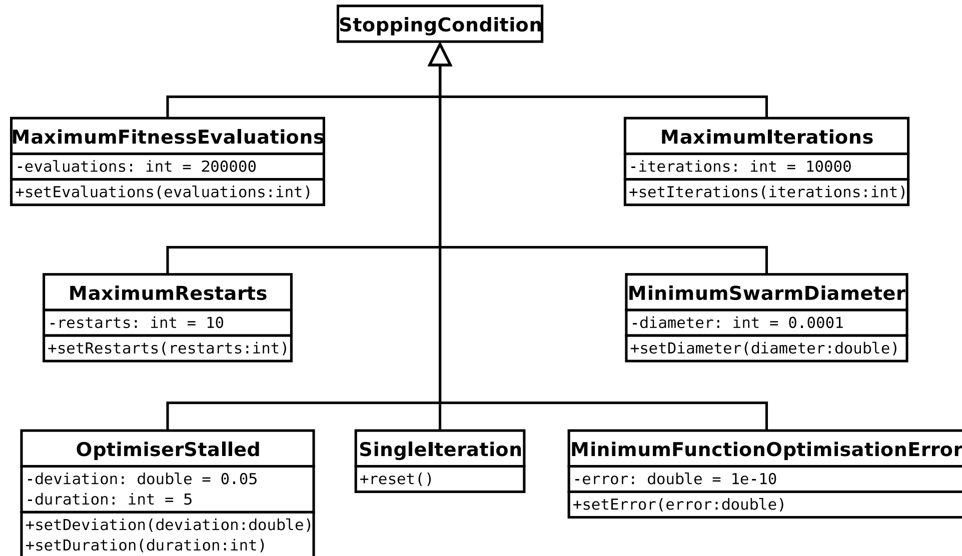


Figure 6.16: Stopping Conditions

tested a predetermined number of times. Implementations of conditions that apply to more specific algorithms must cast the algorithm they are passed into the type they expect it to be, throwing a `ClassCastException` if the user attempts to apply an unsuitable stopping condition to an algorithm. Table 6.1 lists the legal types of algorithm for each stopping condition.

The minimum swarm diameter condition fires when the average distance between particles and the global best drops below a threshold. Similarly, the minimum function optimisation error condition fires when the optimisation error, given by $|f(\mathbf{x}^*) - f(\mathbf{x})|$ for an objective function f with global extremum $\mathbf{x}^*$ and solution $\mathbf{x}$, drops below a threshold. Further, the `OptimiserStalled` condition fires when the standard deviation of an optimisation solution over a configurable number of iterations is less than a threshold. The single iteration condition is a special case condition, which fires after one iteration and does not permit execution again until it is reset. Finally, the maximum restarts condition fires whenever the number of restarts of a multi-start optimisation algorithm exceeds a threshold.

Wherever possible, an implementation should return a linearly increasing value in the range $[0, 1]$ for the `getPercentageComplete()` method (refer to Figure 6.10). For example, the maximum iterations condition returns the fraction $(\frac{\text{current iteration}}{\text{maximum iterations}})$. Con-

University of Pretoria etd – Peer, E S (2005)

Table 6.1: Legal Algorithms for Stopping Conditions

Stopping Condition	**Legal Algorithms**
`MaximumFitnessEvaluations`	Any optimisation algorithm
`MaximumIterations`	Any algorithm
`MaximumRestarts`	Only the multi-start optimisation algorithm
`MinimumSwarmDiameter`	Any particle swarm optimiser
`OptimiserStalled`	Any optimisation algorithm
`SingleIteration`	Any algorithm
`MinimumFunctionOptimisationError`	Only optimisation algorithms applied to function optimisation problems

ditions such as those based on the swarm diameter or optimisation error cannot make this guarantee, since they are dependent on the non-linear behaviour of the algorithm. However, they should still ensure to return a value in the correct range, even if it is only a binary 0 or 1 based on the output of `isFinished()`.

6.2.6 Measurements

Any platform designed for scientific research must be able to perform proper measurements during an experiment. The framework should enable a researcher to choose any property to measure and not dictate its type.

The CILib simulator, discussed in the next section, makes use of measurements to evaluate such properties during the execution of an algorithm. No restrictions are placed on the type of property, measurements return a `java.lang.Object`, with each measurement specifying its own domain, as a domain string which can be used to generate a domain description (refer to Section 6.2.1). Thus, irrespective of the property being measured, a measurement presents a uniform interface to a client, usually the simulator, as shown in Figure 6.17.

New measurements can be crafted to access any property in an algorithm's publicly accessible object reference graph. That is, measurements access the currently executing algorithm using the global `Algorithm.get()` method (refer to Section 6.2.3). Like stopping conditions, they need to cast the algorithm into the type they are expecting

University of Pretoria etd – Peer, E S (2005)

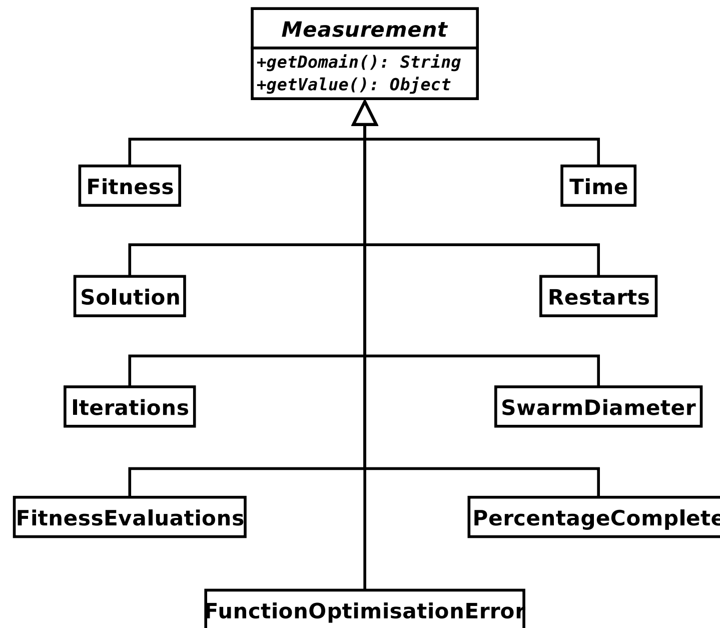


Figure 6.17: Measurements

and navigate to the property they are interested in. The implementation, however, may have difficulty locating properties if objects are composed in unexpected ways, particularly if they are deep in the graph. Using the global algorithm accessor enables a single measurement instance to be shared, provided they do not store any non-sharable state, since they do not maintain a reference to the algorithm (in future measurements may be implemented as *Singletons*, refer to Section 3.1.4).

Figure 6.17 shows a number of reusable measurements, so a researcher only needs to create new measurements if they are measuring something unusual. As was the case for stopping conditions, some measurements are specific to certain types of algorithms. Measurements have been defined for monitoring the solution and its fitness (for optimisation algorithms), number of fitness evaluations, current time, number of restarts (for the multi-start optimisation algorithm), number of iterations, percentage complete, swarm diameter (for particle swarms) and function optimisation error (for algorithms optimising functions). In fact, many of these are precisely the same properties which are monitored by stopping conditions.

Implementing stopping conditions using measurements has been considered as a means to reduce these parallel class hierarchies. That way, only two stopping con-

University of Pretoria etd – Peer, E S (2005)

ditions would be necessary, a maximum threshold condition which fires whenever the measured value exceeds a threshold and a complementary minimum version. For example, the maximum iterations stopping condition could be implemented using a maximum threshold condition and the `Iterations` measurement. The problem with this approach stems from the fact that measurements can have any type, numeric or otherwise. Thus, even for simple numeric types, which are handled very efficiently by stopping conditions, a measurement needs to perform an expensive object instantiation, creating a new `java.lang.Number`. Since measurements used by the simulator are typically only executed every k^{th} iteration for fairly large values of k, they can afford this inefficiency for the benefit of being able to deal with any type of property. Further, the measurement interface would require the stopping condition to perform an additional down cast before it can use the value. If measurements are to be used in stopping conditions, then the performance implications of the extra work performed after every iteration needs to be considered and properly bench-marked first.

Algorithm implementations are not aware of any clients which are performing measurements, since the client simply needs to declare itself as an *Observer* (refer to Section 3.3.3) and can execute any measurements, by calling their `getValue()` method, as it sees fit. Thus, all scientific measurement code is kept out of the implementations of algorithms, which do not need to concern themselves with how their behaviour will be monitored beyond providing sufficient public accessors for any interesting properties. This ensures that algorithm implementations do not become polluted with measurement code, which may not required in all circumstances. For example, if an algorithm is implemented in a non-research context, as part of another application.

6.2.7 Simulator

The simulator is CILib's mechanism for configuring and executing experiments. The heart of the simulator is an XML object factory, which enables algorithms, problems and measurements to be constructed, configured and composed at run time according to a simple XML document. The `XMLObjectFactory` class, which accepts a DOM element (refer to Section 5.1.3) describing its configuration, can be used over and over again to construct objects with the same configuration. Further, it can be trivially *Adapted* (refer to Section 3.2.1) to be the implementation for any *Abstract Factory* (refer to

University of Pretoria etd – Peer, E S (2005)

Section 3.1.1) interface, as shown in Figure 6.18.

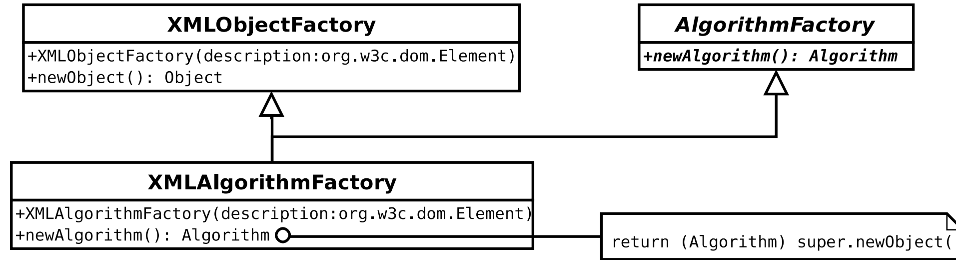


Figure 6.18: XML Object Factory

Figure 6.19 is an example configuration for the CILib simulator, using a standard PSO with a linear decreasing inertia component to find the minimum of the spherical function on its default domain of "R(-100,100)^30", given by:

$$f(\mathbf{x}) = \sum_{i=1}^{30} x_i^2, \text{ with } x_i \in \{\mathbb{R} \mid -100 \leq x_i \leq 100\} \tag{6.1}$$

while measuring the number of iterations and function optimisation error, by default every 100 iterations, and outputting the results to a file named "inertia.txt". By default, the simulator repeats the experiment 30 times, actually it runs them in parallel threads, outputting all the results to the same file, where they can be later analysed.

The simulation engine searches the document for `<simulation/>` elements, each containing the configuration for running a single algorithm on a given problem while measuring certain properties. All objects must have a default constructor and should provide sensible defaults for all of their properties. Any publicly accessible property can be set by specifying a corresponding tag name in the configuration. The document's legal tag names are dictated by the properties available in the source code at run time, using the Java reflection API, so it is impossible to construct a rigid schema that describes valid simulator documents (refer to Section 5.1.2).

For example, because the `PSO` exposes a public velocity update property, via the `setVelocityUpdate(VelocityUpdate vu)` method, it can be set using a tag corresponding to that property name. A `class` attribute specifies the name of a class that should be instantiated by the factory and passed to the property specified in its element. Class names are specified relative to the `net.sourceforge.cilib` package, however, fully qualified class names are also permitted.

University of Pretoria etd – Peer, E S (2005)


```
<simulator>
  <simulation>
    <algorithm class="PSO.PSO">
      <addStoppingCondition class="StoppingCondition.MaximumIterations"/>
      <velocityUpdate class="PSO.StandardVelocityUpdate">
        <inertiaComponent class="PSO.LinearDecreasingValue">
          <minimumValue>0.25</minimumValue>
          <maximumValue>1.0</maximumValue>
        </inertiaComponent>
      </velocityUpdate>
    </algorithm>
    <problem class="Problem.FunctionMinimisationProblem">
      <function class="Functions.Spherical"/>
    </problem>
    <measurements class="Simulator.MeasurementSuite">
      <file>inertia.txt</file>
      <addMeasurement class="Measurement.Iterations"/>
      <addMeasurement class="Measurement.FunctionOptimisationError"/>
    </measurements>
  </simulation>
</simulator>
```

Figure 6.19: Simple Simulator Configuration

Strings and primitive typed properties can be set by simply enclosing their value within the element body. Thus, in the sample, the minimum and maximum values for an instance of `LinearDecreasingValue` are set to 0.25 and 1.0 respectively. Similarly, the name of the file into which the measurement suite will output its results is specified within a `<file/>` element, which corresponds to the `setFile(String fileName)` method in the `MeasurementSuite` class.

Arbitrary methods can be called by using the method name as the tag name, the XML object factory simply provides a short hand for properties (indicated by a method with the prefix "set"). Thus, multiple stopping conditions and measurements can be added

University of Pretoria etd – Peer, E S (2005)

using the `addStoppingCondition()` method in `Algorithm` and the `addMeasurement()` method in `MeasurementSuite` respectively. Methods with an arbitrary number of parameters are also supported by nesting each parameter as a separate element (their names do not matter) within the method element in the order they appear in the method signature.

Figure 6.20, in turn, illustrates another slightly more complex configuration file. This example demonstrates how portions of the document can be reused by making use of ID references (refer to Section 5.1.1). Typically, more descriptive identifiers than “A”, “B”, “M” and “S” would be used, they were shorted here purely for formatting reasons. Note that the fact that multiple algorithms and simulations are specified within `<algorithms/>` and `<simulations/>` elements is immaterial. The simulator merely searches for simulation elements and follows any identity links to their targets, irrespective of where they are defined in the document. Further, the sample demonstrates two short hand ways to set properties. Primitive and string valued properties can be specified directly as attributes in the parent element instead of nesting them as separate elements. Alternatively, they can be specified using the `value` attribute of their own property tags instead of placing the value in the body of the element. Properties in reused portions of the document can be overridden where they are referenced. For example, the same measurement suite configuration is used to output to two different file names. In addition, the measurement suite has two additional properties: i) the number of repetitions of the experiment, or samples; and ii) the resolution, which specifies how often results are written to file. Finally, the cooperative optimisation algorithm uses the `XMLAlgorithmFactory` *Adapter* demonstrated in Figure 6.18. An XML algorithm factory expects its configuration to be specified in a nested `<algorithm/>` element and from there on down functions in exactly the same manner as the XML object factory.

Further examples of configuration files are distributed with the CILib source code. Additional examples which demonstrate all the features of the XML object factory are also available for download from the CILib project page.

6.3 Collaborations

To date, CILib has relatively mature implementations of particle swarm and ant colony frameworks. An early EC framework which is in need of some refactoring, to take into account improvements to the core framework since it was contributed, has also been

University of Pretoria etd – Peer, E S (2005)


```
<simulator>
  <algorithms>
    <algorithm id="A" class="Algorithm.CoOperativeOptimisationAlgorithm">
      <algorithmFactory class="XML.XMLAlgorithmFactory">
        <algorithm idref="B"/>
      </algorithmFactory>
      <participants value="10"/>
    </algorithm>
    <algorithm id="B" class="PSO.PSO">
      <topology class="PSO.VonNeumannTopology"/>
      <addStoppingCondition class="StoppingCondition.MaximumIterations"/>
    </algorithm>
  </algorithms>
  <problem id="S" class="Problem.FunctionMinimisationProblem">
    <function class="Functions.Spherical" domain="R(-50,50)^100"/>
  </problem>
  <measurements id="M" class="Simulator.MeasurementSuite" samples="50">
    <addMeasurement class="Measurement.FitnessEvaluations"/>
    <addMeasurement class="Measurement.FunctionOptimisationError"/>
  </measurements>
  <simulations>
    <simulation>
      <algorithm idref="A"/>
      <problem idref="S"/>
      <measurements idref="M" file="data/cpso.txt"/>
    </simulation>
    <simulation>
      <algorithm idref="B"/>
      <problem idref="S"/>
      <measurements idref="M" file="data/pso.txt"/>
    </simulation>
  </simulations>
</simulator>
```

Figure 6.20: More Complex Simulator Configuration

University of Pretoria etd – Peer, E S (2005)

implemented. In addition, several benchmark functions have been defined for testing optimisation algorithms. Neural network and coevolutionary game (based on Blondie 24, refer to Section 2.7) frameworks are currently being worked on by other students as part of their studies. No significant contributions have been received from parties outside of the University of Pretoria, but it has not yet been very widely advertised either. Further, nothing has been implemented in the fuzzy systems paradigm, mainly because nobody in the CIRG@UP is currently focusing on research in that field. The framework has been offered as a platform for implementing assignments for postgraduate courses and has received a fair amount of interest from those students. Table 6.2 lists the names of significant contributors[2], crediting them with the parts of CILib that they have been primarily responsible for.

Table 6.2: CILib Contributors

Names	**Contributions**
Barla-Szabo, D.	LFPSO
Engelbrecht, A. P.	Benchmark Functions, PSO Additions
Kroon, J.	Nonlinear Mapping Problems [71], Domain Visitor
Naicker, C.	NichePSO, Benchmark Functions, EC Framework
Pampara, G.	Ant System Framework, Containers
Papaconstantis, E.	Coevolutionary Games Framework
Peer, E. S.	CILib Core, Benchmark Functions, PSO Framework
Van der Stockt, S.	Neural Network Framework
Van Niekerk, F	Cooperative Algorithms

6.4 Limitations

CILib successfully meets many of the goals identified at the start of this chapter. The use of design patterns and the XML object factory provide for a very flexible framework, where classes can be composed at will to produce any permutation permitted by the design. Experimentation is facilitated by the simulator, which provides for making

[2]http://sourceforge.net/project/memberlist.php?group_id=72233

University of Pretoria etd – Peer, E S (2005)

measurements of any interesting property during the execution of an algorithm. The domain system presented in Section 6.2.1 ensures that algorithms can use efficient types wherever possible, trading off the OO neatness of a polymorphic type system in favour of better performance, with a view to make the design cleaner as better compilers become available. A clean separation between algorithms, problems and measurements enables algorithms to be separated out and used in real world applications, not only within the research framework. In addition, the open source development model and having multiple people working on the same code base has forced improvements on the design, to make it accommodate their needs, and contributed towards numerous bug fixes.

That said, the CILib design is by no means perfect and continuous refactoring will be necessary as the framework grows to support more. Further, although CILib has generated numerous collaborative opportunities internally, it has yet to prove itself to a wider audience. A lack of documentation, which this dissertation hopes to alleviate, has also contributed to a steep learning curve for those wishing to use the software. Also, it has been difficult to convince some contributors of the benefits of unit testing (refer to Section 5.5), particularly when the correct outcomes for stochastic processes are not known *a priori*. Thus, there is lack of test cases for much of the implementation. Already, test cases for certain benchmark functions have proven their worth, where an error, which was discovered by a unit test, would have resulted in incorrect simulation results.

The following is a non-exhaustive list of some more specific limitations that have been identified:

- **Expensive fitness evaluations:** To accommodate discrete optimisation problems in CILib, the return value of benchmark functions was altered from a primitive `double` value to a `java.lang.Object` type. This means that every evaluation of an objective function typically results in an new instance of `java.lang.Number` being created. In addition to the extra object creation, the use of objective functions in tight loops places a severe strain on the garbage collector, since large amounts of memory will be consumed and need to be reclaimed. The mutable polymorphic type system presented in Section 6.2.1 may provide an efficient solution for this problem, since the same object used in the previous evaluation of an individual's position during a previous iteration can be reused by passing it as a reference to an objective function.

University of Pretoria etd – Peer, E S (2005)

- **Loose configuration file format:** The configuration file format was designed with hand crafting the document in mind. So, instead of having tags with consistent names and attributes with values corresponding to property names, it was decided to shorten the format by having the element name itself refer to the property name. In retrospect, it would have been better to follow an approach that can be validated against a static schema, which would have made writing the GUI tools discussed in the next chapter simpler. For example, instead of implementing a custom schema validator that needs to introspect the source code to perform its work, it would have been possible to make use of the XMLBeans[3] framework, capable of mapping an XML document directly onto Java objects.

- **Scalability:** The simulator spawns a new thread of execution for each sample. The motivation for this was that Unix tools such as GNU `awk`[4], which can be used for processing results, operate most conveniently on data presented in columns for each measurement of each sample. Since text files are most naturally written in rows, executing experiments sequentially would mean that information for subsequent columns would not be available. By running the experiments in parallel, it was hoped that all the information required for a given row would become available at roughly the same time, avoiding the need to buffer a large quantity of measurement results, which can quickly grow to hundreds of megabytes in size. Unfortunately, because of this, the simulator can not scale to large numbers of samples. The extra scheduling overhead and larger footprint required for keeping multiple executing algorithms in memory at the same time can become prohibitive. The implicit assumption that this memory overhead would be less than buffering the results also does not always hold, particularly if one thread of execution becomes starved of CPU time, in which case the buffering overhead is incurred anyway. The following chapter presents a solution to this problem, by storing the results in a structured database, as well as being able to scale experiments up to a cluster of workstations. Alternatively, the simulator could trivially be changed to write results in rows, requiring post processing for tools like `awk`, or results could be temporarily buffered to disk so that simulations can be run sequentially.

[3] http://xmlbeans.apache.org/

[4] http://www.gnu.org/software/gawk/gawk.html

University of Pretoria etd – Peer, E S (2005)

In spite of these and other limitations, CILib is already useful in its current state and has the potential to become an important collaborative resource in the future.

University of Pretoria etd – Peer, E S (2005)

Chapter 7

CiClops - Collaborative Laboratory

> *"I abhor averages. I like the individual case. A man may have six meals one day and none the next, making an average of three meals per day, but that is not a good way to live." — Louis D. Brandeis*

CiClops (Computational Intelligence Collaborative Laboratory Of Pantological Software), still in its early stages of development, was initially designed to address the scalability limitations of the CILib simulator discussed in the previous chapter, by storing simulation results in a structured database and distributing simulation workloads over a cluster of workstations. Further, CiClops is intended to facilitate empirical studies by maintaining a repository of past simulation data and providing statistical analysis tools. The following high level goals have been identified for CiClops:

- **Scalability:** The CiClops framework should support an arbitrary number of samples per experiment and enable those experiments to be clustered over multiple workstations.

- **Simulation repository:** CI simulations can be very computationally intensive, sometimes requiring days to complete an experiment, even scaled across a cluster of machines. Complete simulation results should be stored in a shared repository, so that existing simulation data can be used as a basis for future comparisons without the need to perform expensive re-computations. Further, the simulation data should keep track of its dependencies on code and data sets, so that if any dependencies change then the results can be recalculated to ensure their correctness.

University of Pretoria etd – Peer, E S (2005)

- **Statistical analysis tools:** The majority of researchers (90% in one study [16]) apply inappropriate parametric tests without first considering whether the assumptions on which those tests are based are satisfied [65]. Further, it has been empirically shown that these assumptions typically do not hold [81, 20]. Thus, CiClops should implement and provide decision support for sound statistical hypothesis testing, so that researchers without the necessary statistical background can reliably perform statistical testing without making errors. It would also be convenient if built-in tools could be used for visualising data in various ways.

- **Ease of use:** CiClops should provide an intuitive GUI, which facilitates experimentation with different parameters and algorithmic configurations.

- **Security:** A granular permission system is required to ensure that, while simulation results and configurations should be sharable, they can also be kept private whenever necessary. For example, it may be desirable to keep results private while working on a competitive publication. In addition, a full audit trail should be maintained to discourage misbehaviour and ensure the integrity of results in circumstances where permissions are permissive. Further, since the services provided by CiClops may have a salable value, only authorised users should be granted any access at all.

- **Revenue stream:** As discussed in Section 4.5, means of turning CiClops into a revenue generating resource should be investigated.

The following section gives a general overview of the CiClops architecture and Section 7.2 reviews its underlying data model. The software component responsible for executing units of work on each node of a cluster is discussed in Section 7.3. Next, the CiClops client interface is covered in Section 7.4. Finally, the current status of CiClops is discussed in Section 7.5

7.1 Architectural Overview

As shown in Figure 7.1, CiClops is implemented using the J2EE framework (refer to Section 5.3) and consists of three essential components:

University of Pretoria etd – Peer, E S (2005)

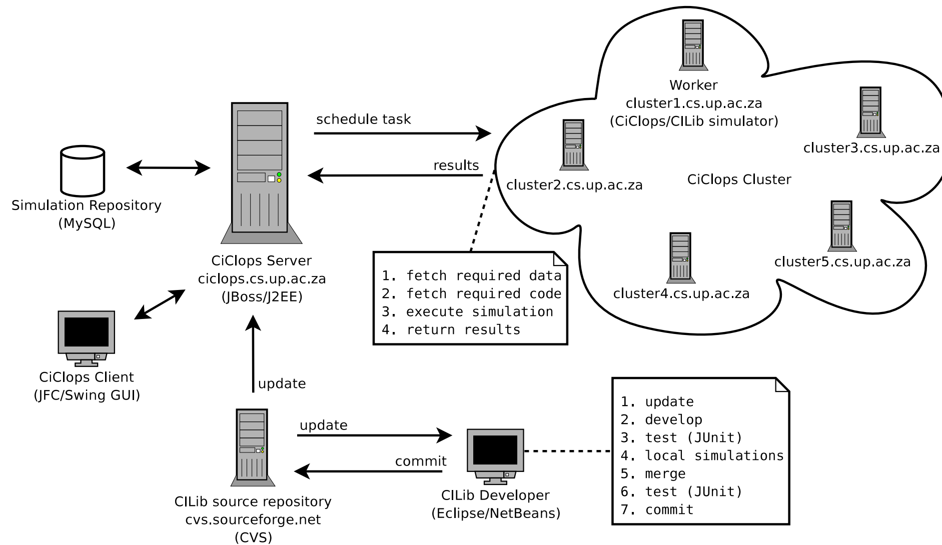


Figure 7.1: CiClops Overview

- **The CILib code base:** CILib forms the most important component, since it is used to conduct the actual simulations. The only change to CILib is the addition of a different simulator, which executes only a single sample at a time, sending the results to the CiClops server instead of writing them to a local file. Note, CiClops periodically (or at the express demand of a user) updates its version of CILib according the version stored in the CVS repository at SourceForge, so care should be taken by developers not to break it, which is why testing is emphasised in the diagram. The CVS code must be kept in a pristine state. Developers must ensure that they update their local version of the code, merge any conflicts with the CVS repository and run local test simulations before committing any changes. If sufficient unit tests are provided to perform proper regression testing, then few problems should be experienced with this approach. Alternatively, CiClops will need to implement different namespaces for code used by different developers, which would inhibit collaboration by spawning multiple versions of the code base.

- **A cluster of workstations:** Each cluster node, or worker, consists of a light weight stub which executes tasks, taking the form of CILib simulations, on behalf of the CiClops server. Workers always execute simulations using the latest available version of the CILib classes and any data sets by means of remote class loading

University of Pretoria etd – Peer, E S (2005)

and efficient local caching of data sets.

- **A central server and data store:** The CiClops server is implemented as a J2EE application and deployed on the open source JBoss application server. The back-end data store is a MySQL relational database, although the J2EE persistence framework makes this largely irrelevant to the application, affecting only the deployment descriptor, which is generated automatically using XDoclet (refer to Section 5.4). The server is responsible for configuring experiments, scheduling tasks on the cluster, archiving simulation results and performing statistical analysis on the results. The load balancing services provided by the J2EE container (refer to Section 5.3.2) means that CiClops can also be scaled up to multiple servers if and when the load of many workers becomes too high for one server to handle.

Finally, some kind of user interface is required to interact with the system. Presently, this is provided in the form of a rich JFC/Swing based GUI client (refer to Section 7.4), with a view to providing a web based front end in the future. Fortunately, this should not be difficult to accomplish, since all the CiClops application logic is executed on the server, lying within the application tier of the J2EE framework.

7.2 Data Model

The data model, or persistence tier, of CiClops is implemented exclusively using CMP entity beans (refer to Section 5.3.1). Figure 7.2 illustrates the object relational mapping employed by CiClops using private attributes, however, it should noted that those private fields do not physically exist and were provided for the sole purpose of making the diagram more readable, since they do at least exist conceptually.

The central concept in the data model is that of a simulation, which is characterised by its name, a description, an XML configuration for CILib and the number of times the experiment represented by this configuration should be repeated, or simply the number of samples. For each sample, a simulation stores the results for each measurement, which are serialised using a CILib domain (refer to Section 6.2.1) and then compressed to save on database space. The domain string is also stored with each measurement so that CiClops is able to deserialise it again, using the CILib domain classes via a *Proxy* (refer to Section 3.2.5) that makes use of the Java reflection API.

University of Pretoria etd – Peer, E S (2005)

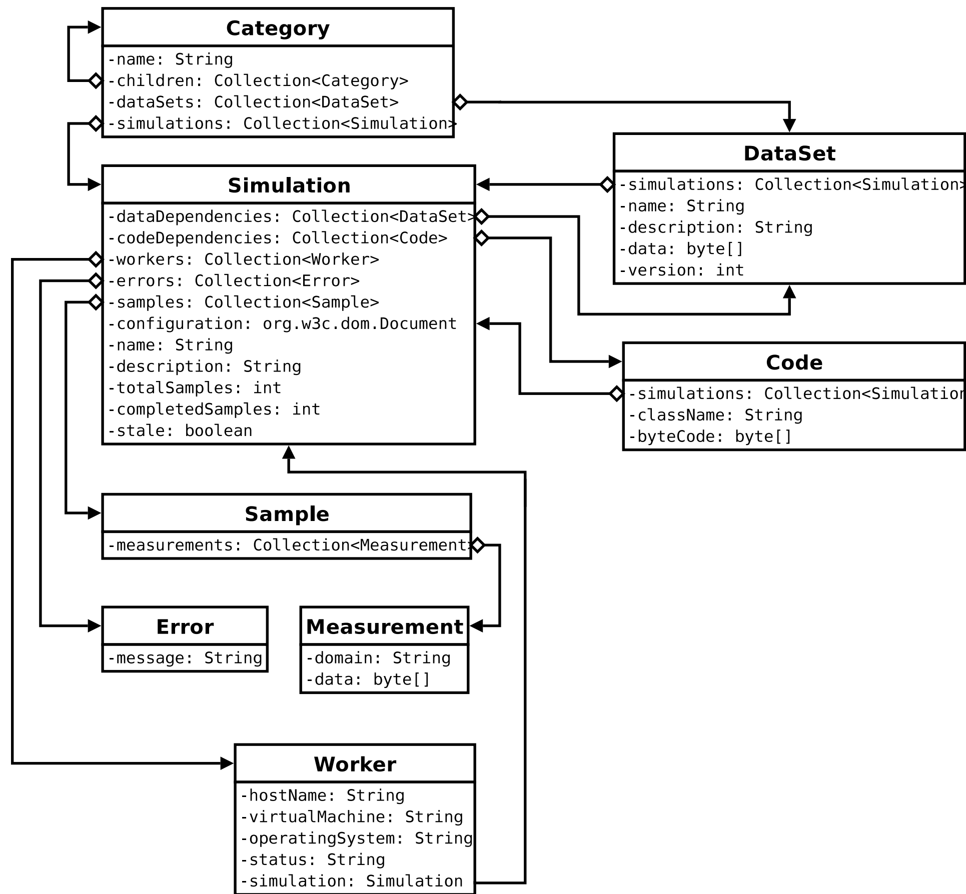


Figure 7.2: CiClops Data Model

Further, a simulation keeps track of its dependencies on particular CILib classes and data sets, or conversely, code and data set entities keep track of the simulations which are dependent on them. Whenever, a class or data set becomes modified they can *Iterate* (refer to Section 3.3.2) over their respective collections of simulations marking each simulation as stale and as a consequence a candidate for rescheduling whenever the cluster is idle. Fortunately, constraints on the data model like these can be isolated in the persistence tier and as such no error in application logic can ever cause a class or data set to become modified without their dependent simulations being marked as stale, particularly considering the transaction isolation provided by the container (refer to Section 5.3.4).

A simulation is scheduled over multiple workers and any errors, in terms of exceptions thrown, experienced by a worker are stored for that simulation to be later examined by

University of Pretoria etd – Peer, E S (2005)

the user. Finally, simulations and data sets are organised into a hierarchical name space, which is imposed by named categories.

7.3 Workers

The data model in the previous section implies that the smallest unit of work that can be scheduled to a worker is a single sample. Experiments should always be sampled at least 30 times [106], meaning that even a single experiment should be able to saturate a cluster of 30 workstations. Currently, the CIRG@UP has fewer than 30 dedicated machines at its disposal and the default number of samples is set to 100 to provide for more robust statistical analysis that may be accomplished using larger samples. Further, it is expected that many different experiments will be configured simultaneously, possibly even by multiple users, enabling CiClops to saturate even hundreds of cluster workstations with this simple scheduling policy. Further parallelism can only be achieved by implementing much more complex scheduling rules, which would require cluster aware algorithms in CILib and incur significantly higher network communication overheads. Responsibilities of workers include:

- **Remote class loading:** Most of the worker logic is implemented in the CiClops simulator, which is actually component of CILib. In fact, the worker part of CiClops consists of little more than a remote class loader, which overrides the standard Java class loader, and a *Proxy* (refer to Section 3.2.5), which is used to fire up the simulator using the reflection API and pass it the XML configuration for a simulation. Thus, code that runs on the cluster is stored and executed from a central location, where it can be upgraded to add new features at any time, without ever modifying the configuration of a workstation.

- **Fetching and caching data sets:** Data sets used in simulations can be very large, and in order to save network bandwidth it makes sense to cache as many as possible data sets on the cluster workstations. Each worker checks the version of any locally cached data set against the server before every simulation and updates its local copy if the versions do not match. The CiClops simulator exposes data sets using the same `net.sourceforge.cilib.Problem.DataSet` interface as the

University of Pretoria etd – Peer, E S (2005)

standard CILib simulator does (refer to Section 6.2.2), so clients do not need to treat remotely loaded data sets any differently.

- **Serialisation and compression of results:** Measurements are serialised using the CILib domain classes (refer to Section 6.2.1) and compressed using the standard Java `java.util.zip.GZipOutputSteam` output stream *Decorator* (refer to Section 3.2.3), providing a relatively good trade off between compression ratio and speed, before being sent back to the CiClops server for storage. Compressing the results on the workstation means that compression load is also distributed across the cluster and further network resources are spared.

7.4 Client

The CiClops client, which as far a possible conforms to the MVC architectural pattern mentioned in Section 5.3.3, currently only supports the configuration of simulations, exporting of their results for external processing and monitoring of the cluster progress.

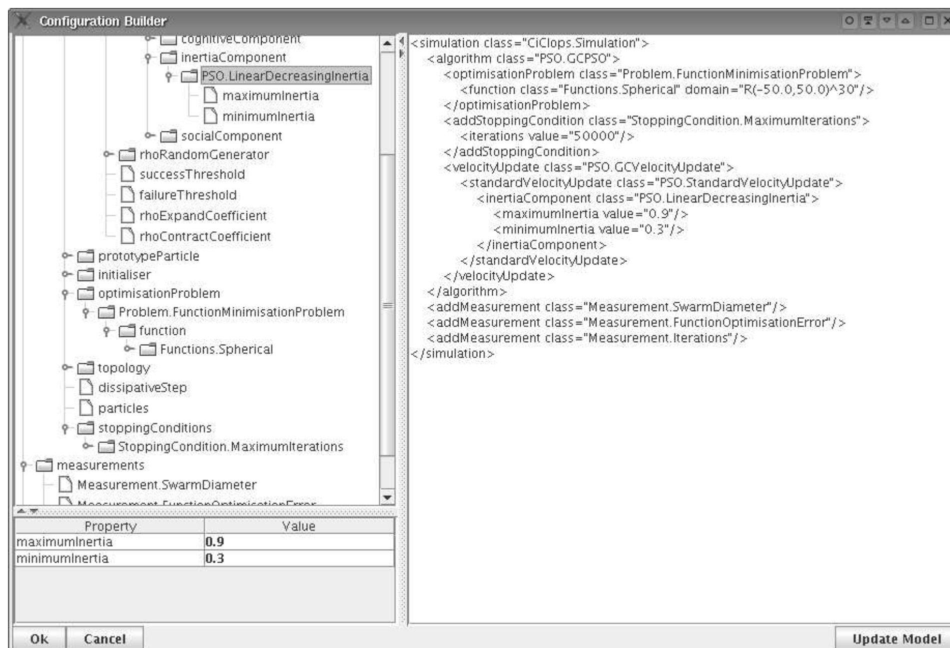


Figure 7.3: Configuring a CILib simulation using CiClops

Figure 7.3 is a screen-shot of the CiClops client being used to build an XML con-

University of Pretoria etd – Peer, E S (2005)

figuration for a CILib simulation. The user may choose to edit the XML configuration directly, however, the hierarchical view of classes and the property editor promote discoverability of CILib features, which the user would otherwise have had to consult the CILib API documentation to learn about. Since the textual, hierarchical and property views all make use of the same model, a combination of these mechanisms can be used simultaneously to edit the configuration. The XML document is validated by the CiClops server against a dynamic schema (refer to Section 5.1.2) which reflects the classes stored in the database.

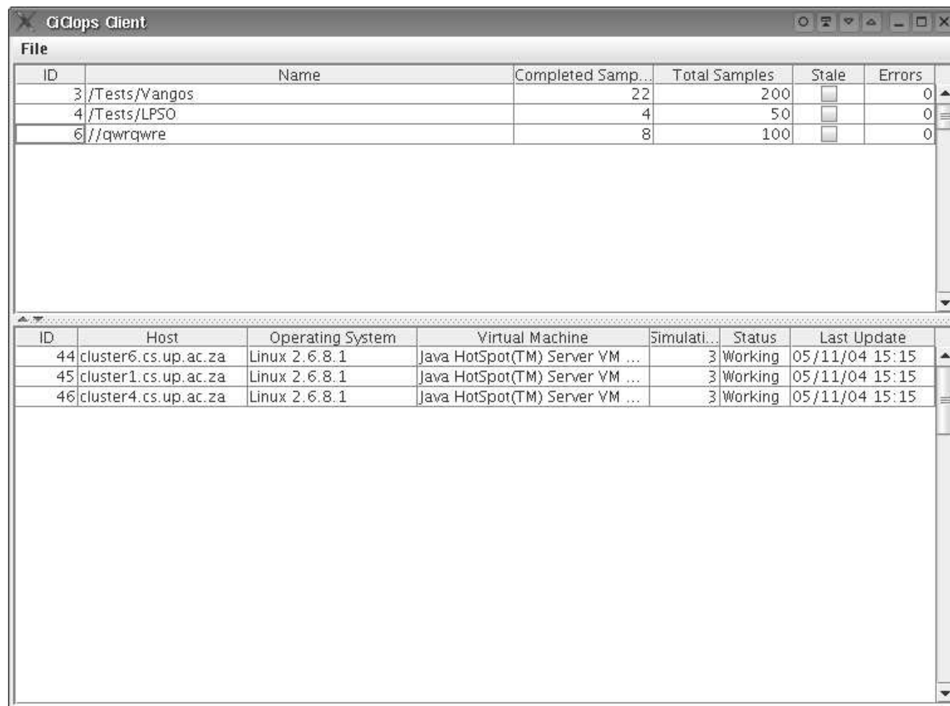


Figure 7.4: CiClops monitoring CILib simulations

Figure 7.4 is another screen-shot taken of the CiClops cluster monitoring view. The figure shows three test simulations (indicated in the top pane) being executed on a small cluster of workstations (indicated in the bottom pane).

University of Pretoria etd – Peer, E S (2005)

7.5 Status

As stated at the beginning of the chapter, CiClops is at an early stage of its development. Custodianship of the source code has recently been handed over to the CIRG@UP and the group as a whole will be continuing its development.

Many of the design goals have already been met, including solving the CILib scalability issue, maintenance of a simulation data repository and the provision of an easy to use mechanism for configuring simulations, by means of a hierarchical GUI builder.

The J2EE declarative security model using XDoclet tags (refer to Sections 5.3.4 and 5.4) presents some challenges. For example, the fact that the security permissions do not appear anywhere, except in the deployment descriptor, means that there is no way for a GUI client to query the security model in order to determine whether or not to present a specific option to a user, without resorting to custom security code. The use of code annotations, which can be queried using the reflection API as provided in the recent Java 1.5 release, for declaring security permissions may provide a solution to this problem, but still needs to be investigated.

Further, statistical analysis methods still need to be adequately investigated. Instead of implementing all the required functionality in-house, it may be better to draw on other software such as the tools available from the Java numerics project[1].

Finally, the CIRG@UP still needs to decide how best to market CiClops to the broader research community, while maximising collaborative and profit opportunities.

[1]http://math.nist.gov/javanumerics/

University of Pretoria etd – Peer, E S (2005)

Chapter 8

Conclusion

> *"Ask her to wait a moment - I am almost done." — Carl Friedrich Gauss (1777-1855), while working, when informed that his wife was dying.*

This chapter briefly summarises this work in Section 8.1 and provides some ideas for future research in Section 8.2.

8.1 Summary

First, this dissertation examined the computational intelligence field, distinguishing between types of problems and the algorithms that can be used to solve them. The complexities introduced by hybrid algorithms were also explored as well as commonalities such as stopping criteria, measurements and the representation of problems.

Design patterns capture the experiential knowledge of expert designers as reusable patterns. Software based on these patterns benefits from more flexible designs that are more able to support new features, often by merely composing classes in different ways. Further, open source software was explored as a mechanism to facilitate collaboration and improved peer review.

CILib demonstrates how design patterns can be applied to provide a flexible computational intelligence framework. Scientific experimentation is facilitated by this flexibility and a simulator governed by an XML configuration file, which enables any algorithm, in any configuration, to be executed on any suitable problem while measuring any number of properties. The improved peer review of open source software and the liberal

University of Pretoria etd – Peer, E S (2005)

use of unit testing should result in CILib becoming a very reliable platform. The fact that CILib is open source software also provides effective incentives for collaboration, including reputation rewards and sharing of development resources.

CiClops was introduced as a platform that primarily addresses the scalability limitations of CILib, a task greatly facilitated by the services provided by J2EE containers. Further, the benefits of a shared simulation repository and the need for statistical analysis tools that provide decision support for their proper use were discussed.

Thus, the combination of CILib and CiClops adequately addresses most of the problems set out in Section 1.2:

- **Duplication of effort:** CILib being open source means that any collaborator is made aware of what others are doing, through a common code base. Further, CiClops provides a common repository of past simulation data so that expensive simulations do no need to be executed more than once.

- **Failure to take latest developments into account:** Once again, the shared open source code base means that once a new idea is implemented, it is immediately available to everyone. That is, any specialist implementing a specific feature immediately makes the platform more general.

- **Insufficient testing on problems:** CiClops enables new experiments to be configured with ease, reducing the effort required to set up more tests. Further, past simulation data can be reused in comparisons and simulations can be executed rapidly on a parallel cluster of workstations.

- **Poor parameter choices:** Good parameter choices for algorithms can be communicated as default values in CILib. Further, CiClop's simulation repository improves awareness of better parameter choices.

- **Conflicting results:** Unit testing, a clean pattern based design and the open source peer review should all contribute to error free software.

- **Invalid statistical inference:** This issue is addressed as an item of future work in Section 8.2

Finally, a number of business models were suggested for exploiting the software for financial gain.

University of Pretoria etd – Peer, E S (2005)

8.2 Future work

The following potential avenues of research have been inspired by this work:

- **The role of open source in collaborative research:** Open source clearly has benefits for collaborative software development. Its role should be studied further, to identify and quantify critical success factors when using open source as a means to facilitate collaborative research, so that these factors may be applied to other projects. If and when CILib becomes successful as a collaborative tool beyond the borders of the CIRG@UP, it can be analysed as a case study to achieve this goal.

- **PSO Taxonomy and characterisation of optimisation problems:** A solid foundation for performing empirical studies is provided by the combination of CILib and CiClops. In this light, the original goal of creating a PSO taxonomy and empirically testing PSOs should be revisited. Further, a method of characterising optimisation problems should be investigated to determine the type of problems for which a particular optimisation algorithm is best suited.

- **MathML for benchmark functions:** Benchmark functions in CILib are implemented using a separate class for each function, resulting in a very large number of classes and no way to define new functions without resorting to writing code. MathML, an XML grammar for defining mathematical expressions, [9] should be investigated as an alternative. A primary concern will be the efficiency of this approach, since benchmark functions are typically executed in tight loops. One possibility worth investigating is compiling MathML function descriptions directly into Java byte code at run time so that they become the equivalent of classes.

- **Statistical analysis tools:** Tools for hypothesis testing need to be implemented in CiClops in consultation with a domain expert on statistics. There is a fair amount of disagreement within the research community regarding the appropriateness of parametric tests when their assumptions are not satisfied [124]. The robustness of parametric tests when their assumptions are not met needs to be properly investigated. Further, alternatives such as Monte Carlo simulations, non-parametric tests, robust procedures, data transformations and re-sampling techniques should also be investigated [124].

University of Pretoria etd – Peer, E S (2005)

- **Aspect Oriented Programming:** The attribute oriented functionality provided by XDoclet (refer to Section 5.4) is a subset of the broader Aspect Oriented Programming (AOP) paradigm [68, 32]. AOP groups together related pieces of code, or aspects, which can be applied across multiple classes by means of source code annotations. For example, the persistence logic provided by a J2EE container is an aspect which can be applied to entity beans by means of XDoclet tags. AOP should be investigated as a means to further improve the design of CILib and CiClops.

- **Mining simulation data:** CiClops has the potential to generate large volumes of simulation data. Data mining [119, 50] techniques should be investigated to determine trends in simulation data. In cases where the underlying data mining algorithms are based on CI techniques, as many are, an interesting question of whether CI techniques be applied recursively to make sense of CI simulation results can be answered.

- **Improved testing and development methodologies:** Unit testing and traditional development methodologies break down for experimental research code. Agile methodologies, such as extreme programming [11], should be studied as inspiration for composing new development methodologies. Further, robust testing mechanisms for stochastic processes should be investigated.

University of Pretoria etd – Peer, E S (2005)

Bibliography

[1] The Java HotSpot Virtual Machine, v1.4.1, Technical White Paper, 2002. http://java.sun.com/products/hotspot/.

[2] OMG Unified Modeling Language, Version 1.5, An Adopted Formal Specification of the Object Management Group, 2003. http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf.

[3] Using Open Source Software in the South African Government, A proposed strategy compiled by the government information technology officers' council, 2003. http://www.oss.gov.za/docs/OSS_Strategy_v3.pdf.

[4] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2003.

[5] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.

[6] R. C. Arkin. *Behaviour-Based Robotics (Intelligent Robotics and Autonomous Agents)*. Bradford Books, 1998.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[8] M. Ashnault, Z. Dean, T. Garben, P. R. Allen, J. J. Bambara, and S. Smith. *J2EE Unleashed*. Sams, 2001.

[9] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Huner, P. Ion, M. Kholhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical

University of Pretoria etd – Peer, E S (2005)

Markup Language (MathML) Version 2.0, W3C Recommendation, Oct. 2003. http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[10] R. Battiti. First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method. *Neural Computation*, 4:141–166, 1992.

[11] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[12] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[13] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes, W3C Recommendation, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[14] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[15] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[16] S. J. Breckler. Application of covariance structure modeling in psychology. *Psychological Bulletin*, 107:260–273, 1990.

[17] R. Brits, A. P. Engelbrecht, and F. van den Bergh. Scalability of Niche PSO. In *IEEE Swarm Intelligence Symposium*, pages 228–234, 2003.

[18] Z. Budimlić. *Compiling Java for High Performance and the Internet*. PhD thesis, Rice University, Houston, Texas, 2001.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[20] N. Cliff. Answering ordinal questions with ordinal data using ordinal statistics. *Multivariate Behavioral Research*, 31:331–350, 1996.

[21] P. Coad and J. Nicola. *Object-Oriented Programming*. Pearson, 1993.

University of Pretoria etd – Peer, E S (2005)

[22] O. Cordon, F. Herrera, F. Hoffmann, and L. Magdelena. *Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases*. World Scientific, 2002.

[23] C. W. Cowell-Shah. Nine Language Performance Round-up: Benchmarking Math & File I/O, 2004. `http://osnews.com/story.php?news_id=5602`.

[24] L. N. de Castro and J. I. Timmis. *Artificial Immume Systems: A New Computational Intelligence Approach*. Springer-Verlag, 2002.

[25] O. P. Doederlein. The Tale of Java Performance. *Journal of Object Technology*, 2(5):17–40, 2003.

[26] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(1):29–41, 1996.

[27] R. Durbin and D. E. Rumelhart. Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks. *Neural Compuation*, 1:133–142, 1989.

[28] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

[29] R. C. Eberhart, P. Simpson, and R. Dobbins. *Computational Intelligence PC Tools*. AP Professional, 1996.

[30] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000.

[31] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley, 2002.

[32] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[33] D. Flanagan. *Java in a Nutshell*. O' Reilly, 4th edition, 2002.

[34] D. B. Fogel. *Blondie 24: Playing At The Edge of AI*. Morgan Kaufmann, 2001.

University of Pretoria etd – Peer, E S (2005)

[35] G. Fogel and D. W. Corne. *Evolutionary Computation in Bioinformatics*. Morgan Kaufmann, 2002.

[36] L. J. Fogel. Autonomous Automata. *Industrial Research*, 4:14–19, 1962.

[37] L. J. Fogel. *On the Organization of Intellect*. PhD thesis, University of California, Los Agneles, 1964.

[38] C. M. Fonseca and P. J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Genetic Algorithms: Proceedings of the Fifth International Conference*, pages 416–423, 1993.

[39] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[40] N. Franken and A. P. Engelbrecht. Comparing PSO Structures to Learn the Game of Checkers from Zero Knowlege. In *IEEE Congress of Evolutionary Computation*, 2003.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[42] S. Geman, E. Bienenstock, and R. Doursat. Neural Networks and the Bias/Variance Dilemma. (4):1–58, 1992.

[43] J. Ghosh and Y. Shin. Efficient Higher-Order Neural Networks for Classification and Function Approximation. *International Journal of Neural Systems*, 3:323–350, 1992.

[44] R. A. Ghosh. Cooking pot markets: an economic model for the trade in free goods and services on the Internet. *First Monday*, 3(3), 1998. `http://www.firstmonday.org/issues/issue3_3/ghosh/index.html`.

[45] J. C. Giarratano. *Expert Systems: Principles and Programming*. PWS Publishing, 3rd edition, 1998.

[46] F. Girosi, M. Jones, and T. Poggio. Regularization Theory and Neural Networks Architectures. *Neural Computation*, 7:219–269, 1995.

University of Pretoria etd – Peer, E S (2005)

[47] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1985.

[48] D. Green. *The Serendipity Machine*. Allen & Unwin, 2004.

[49] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design (Foundations of Computing)*. MIT Press, 1994.

[50] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[51] G. Hardin. The Tragedy of the Commons. *Science*, 162:1243–1248, 1968.

[52] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 1996.

[53] E. R. Harold and W. S. Means. *XML in a Nutshell*. O' Reilly, 3rd edition, 2004.

[54] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[55] D. Hillis. Co-evolving parasites improves simulation evolution as an optimization procedure. Artificial Life II, pages 313–324. Addison-Wesley, 1991.

[56] J. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM*, 3:297–314, 1962.

[57] S. Hommel. Code Conventions for the Java Programming Language, 1999. `http://java.sun.com/docs/codeconv/`.

[58] A. L. Hors, P. L. Hégaret, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification, W3C Recommendation, Apr. 2004. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`.

[59] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java Lanaguage Specification*. Addison-Wesley, 2nd edition, 2000.

[60] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.

University of Pretoria etd – Peer, E S (2005)

[61] J. Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 1931–1938, Washington D.C, USA, July 1999.

[62] J. Kennedy. Bare bones particle swarms. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 88–94, Indianapolis, USA, April 2003.

[63] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, volume IV, pages 1942–1948, Perth, Australia, 1995.

[64] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Proceedings of the IEEE Congress on Evolutionary Computation*, Honolulu, Hawaii USA, May 2002.

[65] H. J. Kesselman, C. Huberty, L. M. Lix, S. Olejnik, R. A. Cribbie, B. Donahue, R. K. Kowalchuk, L. L. Lowman, M. D. Petoskey, and J. C. Keselman. Statistical practices of education researchers: An analysis of their ANOVA, MANOVA, ANCOVA analyses. *Review of Educational Research*, 68:350–386, 1998.

[66] T. Kohonen. *Self-Organizing Maps*. Springer, 1995.

[67] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[68] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[69] Larson, Hostetler, and Edwards. *Calculus*. Heath, 1994.

[70] K. Lea. The Java is Faster than C++ and C++ Sucks. `http://kano.net/javabench/`.

[71] J. A. Lee, A. Lendasse, N. Donckers, and M. Verleysen. A robust nonlinear projection method. In *ESANN*, pages 13–20, 2000.

[72] J. P. Lewis and U. Neumann. Performance of Java versus C++, University of Southern California, 2003. `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`.

University of Pretoria etd – Peer, E S (2005)

[73] G. P. Liu, J. Yang, and J. F. Whidborne. *Multiobjective Optimisation & Control*. Research Studies Press, 2002.

[74] M. Lovbjerg, T. K. Rasmussen, and T. Krink. Hybrid Particle Swarm Optimiser with Breeding and Subpopulations. In *Genetic and Evolutionary Computation Conference*, 2001.

[75] E. H. Mamdani and S. Assilian. An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller. *International Journal of Man-Machine Studies*, 7:1–13, 1975.

[76] C. Mangione. Performance tests show Java as fast as C++, 1998. `http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html`.

[77] E. Mayr. *Animal Species and Evolution*. Belknap, 1963.

[78] R. Mendes, J. Kennedy, and J. Neves. Watch thy neighbor or how the swarm can learn from its environment. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 88–94, Indianapolis, USA, April 2003.

[79] L. Messerschmidt and A. P. Engelbrecht. Learning to Play Games using a PSO-based Competitive Learning Approach. In *Asia-Pcific Conference on Simulated Evolution and Learning*, 2002.

[80] S. J. Metsker. *Design Patterns C#*. Addison-Wesley, 2004.

[81] T. Micceri. The unicorn, the normal curve, and other improbable creatures. *Psychologica Bulletin*, 105:156–166, 1989.

[82] R. E. Michod. *Darwinian Dynamics*. Princeton University Press, 2000.

[83] P. B. Miltersen. MILP, ILP and TSP: Course notes for Search and Optimization, 2004. `http://www.daimi.au.dk/dSoegOpt/ilp.pdf`.

[84] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[85] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[86] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison-Wesley, 2003.

University of Pretoria etd – Peer, E S (2005)

[87] U. Paquet and A. P. Engelbrecht. A New Particle Swarm Optimiser for Linearly Constrained Optimisation. In *Congress on Evolutionary Computation*, 2003.

[88] W. Pedrycz. *Computational Intelligence: An Introduction*. CRC Press, 1998.

[89] E. S. Peer, A. P. Engelbrecht, and F. van den Bergh. CIRG@UP OptiBench: A statistically sound framework for benchmarking optimisation algorithms. In *IEEE Congress on Evolutionary Computation*, 2003.

[90] E. S. Peer, F. van den Bergh, and A. P. Engelbrecht. Using Neighbourhoods with the Guaranteed Convergence PSO. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 235–242, Indianapolis, USA, April 2003.

[91] M. A. Potter and K. A. de Jong. A Cooperative Coevolutionary Approach to Function Optimization. In *The Third Parallel Problem Solving from Nature*, pages 249–257, 1994.

[92] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an accidental revolutionary*. O' Reilly, 2nd edition, 2001.

[93] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, 1973.

[94] I. Rechenberg. *Evolutionsstrategie*. Frommann-Holzboog, 1994.

[95] K. Reinholtz. Java will be faster than C++. *ACM SIGPLAN Notices*, 35(2):25–28, 2000.

[96] R. G. Reynolds. An Introductions to Cultural Algorithms. In *Conference on Evolutionary Computing*, pages 131–139, 1994.

[97] R. M. Roberts. *Serendipity: Accidental Discoveries in Science*. Wiley, 1989.

[98] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. Wiley, 2000.

University of Pretoria etd – Peer, E S (2005)

[99] A. C. Schultz, L. E. Parker, and F. E. Schneider, editors. *Multi-Robot Systems: From Swarms to Intelligent Automata*, volume 2. Kluwer Academic Publishers, 2003.

[100] Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 69–73, Anchorage, Alaska, May 1998.

[101] Y. Shi and R. C. Eberhart. Empirical study of particle swarm optimisation. In *Proceedings of the IEEE International Congress on Evolutionary computation*, pages 101–106, 1999.

[102] J. A. Snyman. A New and Dynamic Method for Unconstrained Minimization. *Applied Mathematical Modelling*, 6:449–462, 1982.

[103] F. Solis and R. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6:19–30, 1981.

[104] D. Spiller and T. Wichmann. Basics of Open Source Software Markets and Business Models, Free/Libre Open Source Software: Survey and Study, 2002. `http://www.berlecon.de/studien/downloads/200207FLOSS_Basics.pdf`.

[105] R. M. Stallman, L. Lessig, and J. Gay. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, 2002.

[106] A. G. W. Steyn, C. F. Smit, S. H. C. du Toit, and C. Strasheim. *Modern Statistics in Practice*. J. L. van Schaik, 2nd edition, 1996.

[107] M. Su, T. A. Liu, and H. T. Chang. An Efficient Initialization Scheme for the Self-Organising Feature Map Algorithm. In *IEEE IJCNN*, 1999.

[108] P. N. Suganthan. Particle Swarm Optimiser with Neighbourhood Operator. In *IEEE Congress of Evolutionary Computation*, 1999.

[109] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, 2000.

University of Pretoria etd – Peer, E S (2005)

[110] T. Takagi and M. Sugeno. Fuzzy Identification of Systems and its Application to Modeling and Control. *IEEE Transactions of Systems, Man, and Cybernetics*, 15(1):116–132, 1985.

[111] W. Theunissen, A. Boake, and D. G. Kourie. A Preliminary Investigation of the Impact of Open Source Software on Telecommunication Software Development. In *Southern African Telecommunication Networks & Applications Conference (SANTAC)*, 2004.

[112] H. S. Thompson, D. Beech, M. Maloney, and N. Medelsohn. XML Schema Part 1: Structures, W3C Recommendation, Oct. 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

[113] F. van den Bergh. *An Analysis of Particle Swarm Optimizers*. PhD thesis, Department of Computer Science, University of Pretoria, South Africa, 2002.

[114] F. van den Bergh and A. Engelbrecht. A new locally convergent particle swarm optimizer. In *Proceedings of IEEE Conference on Systems, Man and Cybernetics*, Hammamet, Tunisia, Oct. 2002.

[115] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O' Reilly, 1999.

[116] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, Boston, USA, 1974.

[117] R. P. Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. PhD thesis, George Mason University, Virgina, 2003.

[118] P. M. Williams. Bayesian Regularization and Pruning Using a Laplace Prior. *Neural Computation*, 7:117–143, 1995.

[119] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[120] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, July 1995.

[121] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 4:67–82, 1997.

University of Pretoria etd – Peer, E S (2005)

[122] X. Xie, W. Zang, and Z. Yang. A dissipative particle swarm optimization. In *IEEE Congress on Evolutionary Computing*, 2002.

[123] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0, W3C Recommendation, Feb. 2004. `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[124] C. H. Yu. An Overview of Remedial Tools for the Violation of Parametric Test Assumptions in the SAS System. In *Western Users of SAS Software Conference*, 2002.

[125] L. A. Zadeh. Fuzzy Sets. *Information and control*, 8:338–353, 1965.

[126] L. A. Zadeh. The Concept of a Linguistic Variable and its Application to Approximate Reasoning - I. *Information Sciences*, 8:199–249, 1975.

[127] L. A. Zadeh. The Concept of a Linguistic Variable and its Application to Approximate Reasoning - II. *Information Sciences*, 8:301–357, 1975.

[128] L. A. Zadeh. The Concept of a Linguistic Variable and its Application to Approximate Reasoning - III. *Information Sciences*, 9:43–80, 1975.

[129] Y. Zhang and A. Kandel. *Compensatory Genetic Fuzzy Neural Networks and Their Applications*. World Scientific, 1998.

[130] H. Zimmermann, G. Tselentis, M. V. Someren, and G. Dounias. *Advances in Computational Intelligence and Learning: Methods and Applications*. Kluwer Academic Publishers, 2002.

University of Pretoria etd – Peer, E S (2005)

Appendix A

List of Acronyms and Abbreviations

Acronym: Abbreviated Coded Rendition Of Name Yielding Meaning

AFL: Academic Free License

AI: Artificial Intelligence

AIS: Artificial Immune System

AL: Artistic License

API: Application Programming Interface

ASL: Apache Software License

BMP: Bean Managed Persistence

CI: Computational Intelligence

CiClops: Computational Intelligence Collaborative Laboratory Of Pantological Software.

CILib: Computational Intelligence Library[1]

CIRG@UP: The Computational Intelligence Research Group at the University of Pretoria[2].

[1] http://cilib.sourceforge.net
[2] http://cirg.cs.up.ac.za

University of Pretoria etd – Peer, E S (2005)

CMP: Container Managed Persistence

CPL: Common Public License

CPU: Central Processing Unit

CVS: Concurrent Versioning System

DOM: Document Object Model

DPSO: Dissipative PSO

DTD: Document Type Definition

EC: Evolutionary Computing

EJB: Enterprise Java Bean

EP: Evolutionary Programming

ES: Evolutionary Strategies

GA: Genetic Algorithm

GC: Garbage Collection

GCC: GNU Compiler Collection

GNU: GNU's Not Unix

GoF: Gang of Four (Gamma, Helm, Johnson, Vlissides)

GPL: General Public License

GUI: Graphical User Interface

HTML: HyperText Markup Language

I/O: Input/Output

J2EE: Java 2 Enterprise Edition

JCP: Java Community Process

University of Pretoria etd – Peer, E S (2005)

JFC: Java Foundation Classes

JIT: Just In Time

JMS: Java Messaging Service

JNDI: Java Naming and Directory Interface

JVM: Java Virtual Machine

LGPL: Lesser General Public License

LVQ: Learning Vector Quantiser

NN: Neural Network

NP: Nondeterministic Polynomial-time

OMG: Object Management Group

OOP: Object Oriented Programming

OSD: Open Source Definition

OSI: Open Source Initiative

OSL: Open Software License

OSS: Open Source Software

PSO: Particle Swarm Optimiser

RPC: Remote Procudure Call

SAX: Simple API for XML

SDK: Software Developer Kit

SI: Swarm Intelligence

SOFM: Self-Organising Feature Map

SSE: Sum Squared Error

University of Pretoria etd – Peer, E S (2005)

TSP: Travelling Salesman Problem

UML: Unified Modelling Language

W3C: World Wide Web Consortium

XML: eXtensible Markup Language

University of Pretoria etd – Peer, E S (2005)

Appendix B

Unified Modelling Language

The notation used for class structure diagrams in this dissertation is based on the Object Management Group (OMG) Unified Modelling Language (UML) specification [2]. Diagrams were composed using the open source Dia[1] tool, which has some minor flaws in terms of formatting and strict conformance to the UML specification. Nonetheless, the diagrams still serve their intended purpose of effectively communicating class structure and relationships.

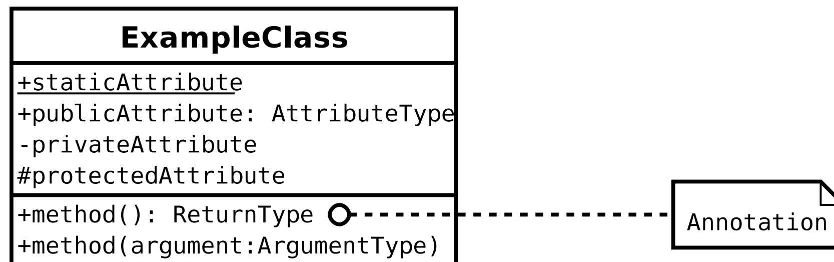


Figure B.1: Example UML Class

Figure B.1 illustrates how a class is represented in UML. The top rectangle contains the class name, the middle contains attributes, or fields, and the bottom contains methods, or operations. The prefix of a plus, minus or hash symbol in front of a class member indicates public, private and protected access modifiers respectively. Class scope, or static, members are underlined. In general, an identifier's type follows after its declaration, preceded by a colon. Method return types are declared to the right of the method

[1]http://www.gnome.org/projects/dia/

University of Pretoria etd – Peer, E S (2005)

definition, and method parameters are indicated within parentheses. While the class name must always be specified, method and attribute blocks may be omitted to simplify a diagram. Annotations are depicted by a piece of paper with a folded corner. Although not shown in the example, abstract operations and class names are indicated in italics.

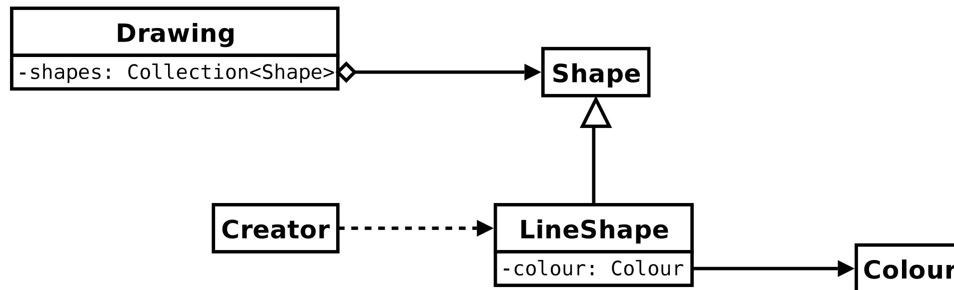


Figure B.2: UML Relationships

Figure B.2 shows the possible relationships between classes. Inheritance is indicated by a line with an open triangle pointing towards the base class. A line that starts with a diamond represents an aggregation relationship where the arrow points to the class that is aggregated. Acquaintance, or simply an object reference, is denoted by an arrow line without a diamond. Whenever possible, the starting point of aggregate or acquaintance arrows are aligned with the attributes taking part in the relationship. Finally, object instantiation is indicated by a dotted line with an arrow pointing from the creating class to the created class.

University of Pretoria etd – Peer, E S (2005)

Appendix C

The Open Source Definition

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

C.1 Free Redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

C.2 Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost, preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

University of Pretoria etd – Peer, E S (2005)

C.3 Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

C.4 Integrity of The Author's Source Code

The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

C.5 No Discrimination Against Persons or Groups

The license must not discriminate against any person or group of persons.

C.6 No Discrimination Against Fields of Endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

C.7 Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

C.8 License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom

University of Pretoria etd – Peer, E S (2005)

the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

C.9 License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

C.10 License Must Be Technology-Neutral

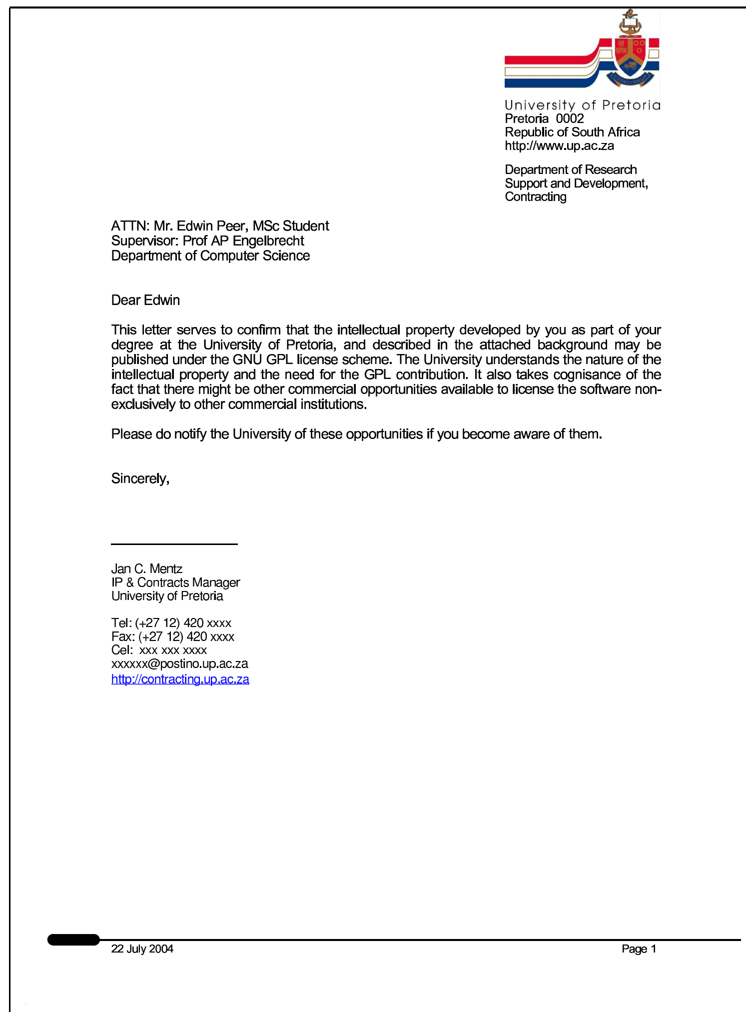
No provision of the license may be predicated on any individual technology or style of interface.

Copyright © 2004 by the Open Source Initiative.
Reproduced under the Open Software License 2.1 or Academic Free License 2.1.

University of Pretoria etd – Peer, E S (2005)

Appendix D

GPL Approval Letter



University of Pretoria
Pretoria 0002
Republic of South Africa
http://www.up.ac.za

Department of Research
Support and Development,
Contracting

ATTN: Mr. Edwin Peer, MSc Student
Supervisor: Prof AP Engelbrecht
Department of Computer Science

Dear Edwin

This letter serves to confirm that the intellectual property developed by you as part of your degree at the University of Pretoria, and described in the attached background may be published under the GNU GPL license scheme. The University understands the nature of the intellectual property and the need for the GPL contribution. It also takes cognisance of the fact that there might be other commercial opportunities available to license the software non-exclusively to other commercial institutions.

Please do notify the University of these opportunities if you become aware of them.

Sincerely,

Jan C. Mentz
IP & Contracts Manager
University of Pretoria

Tel: (+27 12) 420 xxxx
Fax: (+27 12) 420 xxxx
Cel: xxx xxx xxxx
xxxxxx@postino.up.ac.za
http://contracting.up.ac.za

22 July 2004 Page 1

University of Pretoria etd – Peer, E S (2005)

Appendix E

Popular Open Source Licenses

E.1 Academic Free License (AFL)

Version 2.1

This Academic Free License (the "License") applies to any original work of authorship (the "riginal Work") whose owner (the "Licensor") has placed the following notice immediately following the copyright notice for the Original Work:

```
Licensed under the Academic Free License version 2.1
```

1. Grant of Copyright License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license to do the following:

   - to reproduce the Original Work in copies;
   - to prepare derivative works ("Derivative Works") based upon the Original Work;
   - to distribute copies of the Original Work and Derivative Works to the public;
   - to perform the Original Work publicly; and
   - to display the Original Work publicly.

2. Grant of Patent License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, to make, use, sell and offer for sale the Original Work and Derivative Works.

University of Pretoria etd – Peer, E S (2005)

3. Grant of Source Code License. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor hereby agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work, and by publishing the address of that information repository in a notice immediately following the copyright notice that applies to the Original Work.

4. Exclusions From License Grant. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior written permission of the Licensor. Nothing in this License shall be deemed to grant any rights to trademarks, copyrights, patents, trade secrets or any other intellectual property of Licensor except as expressly stated herein. No patent license is granted to make, use, sell or offer to sell embodiments of any patent claims other than the licensed claims defined in Section 2. No right is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under different terms from this License any Original Work that Licensor otherwise would have a right to license.

5. This section intentionally omitted.

6. Attribution Rights. You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7. Warranty of Provenance and Disclaimer of Warranty. Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately proceeding sentence, the Original Work is provided

University of Pretoria etd – Peer, E S (2005)

under this License on an “AS IS” BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to Original Work is granted hereunder except under this disclaimer.

8. Limitation of Liability. Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to any person for any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to liability for death or personal injury resulting from Licensor's negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

9. Acceptance and Termination. If You distribute copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. Nothing else but this License (or another written agreement between Licensor and You) grants You permission to create Derivative Works based upon the Original Work or to exercise any of the rights granted in Section 1 herein, and any attempt to do so except under the terms of this License (or another written agreement between Licensor and You) is expressly prohibited by U.S. copyright law, the equivalent laws of other countries, and by international treaty. Therefore, by exercising any of the rights granted to You in Section 1 herein, You indicate Your acceptance of this License and all of its terms and conditions.

10. Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11. Jurisdiction, Venue and Governing Law. Any action or suit relating to this License may

University of Pretoria etd – Peer, E S (2005)

be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of the U.S. Copyright Act, 17 U.S.C. ?? 101 et seq., the equivalent laws of other countries, and international treaty. This section shall survive the termination of this License.

12. Attorneys Fees. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13. Miscellaneous. This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14. Definition of "You" in This License. "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15. Right to Use. You may use the Original Work in all ways not otherwise restricted or conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

This license is Copyright © 2003-2004 Lawrence E. Rosen. All rights reserved. Permission is hereby granted to copy and distribute this license without modification. This license may not be modified without the express written permission of its copyright owner.

University of Pretoria etd – Peer, E S (2005)

E.2 Apache Software License (ASL)

Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

University of Pretoria etd – Peer, E S (2005)

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

University of Pretoria etd – Peer, E S (2005)

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and

University of Pretoria etd – Peer, E S (2005)

customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[ ]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file

University of Pretoria etd – Peer, E S (2005)

format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the ''License'');
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an ''AS IS'' BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

E.3 Artistic License (AL)

Version 2.0beta4, October 2000

Copyright (C) 2000, Larry Wall.
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This copyright license states the terms under which a given free software Package may be copied, modified and/or redistributed, while the Originator(s) maintain some artistic control over the future development of that Package (at least as much artistic control as can be given under copyright law while still making the Package open source and free software).

This license is bound by copyright law, and thus it legally applies only to works which the copyright holder has permitted copying, distribution or modification under the terms of the Artistic License, Version 2.0.

You are reminded that You are always permitted to make arrangements wholly outside of a given copyright license directly with the copyright holder(s) of a given Package. If the terms of

University of Pretoria etd – Peer, E S (2005)

this license impede your ability to make full use of the Package, You are encouraged to contact the copyright holder(s) and seek a different licensing arrangement.

Definitions

"Package" refers to the collection of files distributed by the Originator(s), and derivatives of that collection of files created through textual modification.
"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways suggested by the Originator(s).
"Modified Version" refers to the Package, if it has been changed by You via textual modification of the source code, and such changes were not suggested by the Originator(s).
"Originator" refers to the author(s) and/or copyright holder(s) of the Standard Version of the Package.
"You" and "Your" refers to any person who would like to copy, distribute, or modify the Package.
"Distribution Fee" is any fee that You charge for providing a copy of this Package to another party. It does not refer to licensing fees.
"Freely Available" means that:
(a) no fee is charged for the right to use the item (though a Distribution Fee may be charged).
(b) recipients of the item may redistribute it under the same conditions they received it.
(c) If the item is a binary, object code, bytecode, the complete corresponding machine-readable source code is included with the item.

Permission for Use and Modification Without Redistribution

1. You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not redistribute the Modified Version to others outside of your company or organization.

Permissions for Redistribution of the Standard Version

2. You may make available verbatim copies of the source code of the Standard Version of this Package in any medium without restriction, either gratis or for a Distribution Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At Your discretion, such verbatim copies may or may not include compiled bytecode, object code or binary versions of the corresponding source code in the same medium.

University of Pretoria etd – Peer, E S (2005)

3. You may apply any bug fixes, portability changes, and other modifications made available from any of the Originator(s). The resulting modified Package will still be considered the Standard Version, and may be copied, modified and redistributed under the terms of the original license of the Standard Version as if it were the Standard Version.

Permissions for Redistribution of Modified Versions of the Package as Source

4. You may modify your copy of the source code of this Package in any way and distribute that Modified Version (either gratis or for a Distribution Fee, and with or without a corresponding binary, bytecode or object code version of the Modified Version) provided that You clearly indicate what changes You made to the Package, and provided that You do at least ONE of the following:

   (a) make the Modified Version available to the Originator(s) of the Standard Version, under the exact license of the Standard Version, so that the Originator(s) may include your modifications into the Standard Version (at their discretion).

   (b) modify any installation scripts and procedures so that installation of the Modified Version will never conflict with an installation of the Standard Version, include for each program installed by the Modified Version clear documentation describing how it differs from the Standard Version, and rename your Modified Version so that the name is substantially different from the Standard Version.

   (c) permit and encourage anyone who receives a copy of the Modified Version permission to make your modifications Freely Available in some specific way.

   If Your Modified Version is in turn derived from a Modified Version made by a third party, then You are still required to ensure that Your Modified Version complies with the requirements of this license.

Permissions for Redistribution of Non-Source Versions of Package

5. You may distribute binary, object code, bytecode or other non-source versions of the Standard Version of the Package, provided that you include complete instructions on where to get the source code of the Standard Version. Such instructions must be valid at the time of Your distribution. If these instructions, at any time while You are carrying our such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If You cease distribution within thirty days after You become

University of Pretoria etd – Peer, E S (2005)

aware that the instructions are invalid, then You do not forfeit any of Your rights under this license.

6. You may distribute binary, object code, bytecode or other non-source versions of a Modified Version provided that You do at least ONE of the following:

   (a) include a copy of the corresponding source code for the Modified Version under the terms indicated in (4).

   (b) ensure that the installation of Your non-source Modified Version does not conflict in any way with an installation of the Standard Version, include for each program installed by the Modified Version clear documentation describing how it differs from the Standard Version, and rename your Modified Version so that the name is substantially different from the Standard Version.

   (c) ensure that the Modified Version includes notification of the changes made from the Standard Version, and offer to provide machine-readable source code (under a license that permits making that source code Freely Available) of the Modified Version via mail order.

Permissions for Inclusion of the Package in Aggregate Works

7. You may aggregate this Package (either the Standard Version or Modified Version) with other packages and distribute the resulting aggregation provided that You do not charge a licensing fee for the Package. Distribution Fees are permitted, and licensing fees for other packages in the aggregation are permitted. Your permission to distribute Standard or Modified Versions of the Package is still subject to the other terms set forth in other sections of this license.

8. In addition to the permissions given elsewhere by this license, You are also permitted to link Modified and Standard Versions of this Package with other works and distribute the result without restriction, provided You have produced binary program(s) that do not overtly expose the interfaces of the Package. This includes permission to embed the Package in a larger work of your own without exposing a direct interface to the Package. This also includes permission to build stand-alone binary or bytecode versions of your scripts that require the Package, but do not otherwise give the casual user direct access to the Package itself.

Items That are Never Considered Part of a Modified Version Package

University of Pretoria etd – Peer, E S (2005)

9. Works (including, but not limited to, subroutines and scripts) that you have linked or aggregated with the Package that merely extend or make use of the Package, but are not intended to cause the Package to operate differently from the Standard Version, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not bound by the terms of the Package's license.

Acceptance of License and Disclaimer of Warranty

10. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, modify or distribute the Standard or Modified Versions of the Package. These actions are prohibited by copyright law if you do not accept this License. Therefore, by copying, modifying or distributing Standard and Modified Versions of the Package, you indicate your acceptance of the license of the Package.

11. Disclaimer of Warranty:

    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT UNLESS REQUIRED BY LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER OR CONTRIBUTOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

E.4 BSD License

Copyright (c) <YEAR>, <OWNER>. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

University of Pretoria etd – Peer, E S (2005)

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

E.5 Common Public License (CPL)

Version 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

   "Contribution" means:

University of Pretoria etd – Peer, E S (2005)

(a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

(b) in the case of each subsequent Contributor:

i. changes to the Program, and

ii. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

(a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

(b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

University of Pretoria etd – Peer, E S (2005)

(c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

(d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

(a) it complies with the terms and conditions of this Agreement; and

(b) its license agreement:

    i. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

    ii. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

    iii. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

    iv. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

(a) it must be made available under this Agreement; and

University of Pretoria etd – Peer, E S (2005)

(b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

   Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

   For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

University of Pretoria etd – Peer, E S (2005)

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

   EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

   If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

University of Pretoria etd – Peer, E S (2005)

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

E.6 GNU General Public License (GPL)

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

University of Pretoria etd – Peer, E S (2005)

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public

University of Pretoria etd – Peer, E S (2005)

License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is

University of Pretoria etd – Peer, E S (2005)

interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all

University of Pretoria etd – Peer, E S (2005)

modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program

University of Pretoria etd – Peer, E S (2005)

by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two

University of Pretoria etd – Peer, E S (2005)

goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

University of Pretoria etd – Peer, E S (2005)


```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail. If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

University of Pretoria etd – Peer, E S (2005)

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

E.7 GNU Lesser General Public License (LGPL)

Version 2.1, February 1999

Copyright © 1991, 1999 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages–typically libraries–of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

University of Pretoria etd – Peer, E S (2005)

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does

University of Pretoria etd – Peer, E S (2005)

the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

University of Pretoria etd – Peer, E S (2005)

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) The modified work must itself be a software library.

   (b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   (c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   (d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

University of Pretoria etd – Peer, E S (2005)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

University of Pretoria etd – Peer, E S (2005)

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   (a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which

University of Pretoria etd – Peer, E S (2005)

must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

(b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

(c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

(d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

(e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and dis-

University of Pretoria etd – Peer, E S (2005)

tribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

(a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

(b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example,

University of Pretoria etd – Peer, E S (2005)

if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the

University of Pretoria etd – Peer, E S (2005)

two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

University of Pretoria etd – Peer, E S (2005)


```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

E.8 MIT License

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute,

University of Pretoria etd – Peer, E S (2005)

sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

E.9 Mozilla Public License (MPL)

Version 1.1

1. Definitions.

   1.0.1. "Commercial Use" means distribution or otherwise making the Covered Code available to a third party.

   1.1. "Contributor" means each entity that creates or contributes to the creation of Modifications.

   1.2. "Contributor Version" means the combination of the Original Code, prior Modifications used by a Contributor, and the Modifications made by that particular Contributor.

   1.3. "Covered Code" means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

   1.4. "Electronic Distribution Mechanism" means a mechanism generally accepted in the software development community for the electronic transfer of data.

   1.5. "Executable" means Covered Code in any form other than Source Code.

   1.6. "Initial Developer" means the individual or entity identified as the Initial Developer in the Source Code notice required by Exhibit A.

University of Pretoria etd – Peer, E S (2005)

1.7. "Larger Work" means a work which combines Covered Code or portions thereof with code not governed by the terms of this License.

1.8. "License" means this document.

1.8.1. "Licensable" means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

1.9. "Modifications" means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

A. Any addition to or deletion from the contents of a file containing Original Code or previous Modifications.

B. Any new file that contains any part of the Original Code or previous Modifications.

1.10. "Original Code" means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

1.10.1. "Patent Claims" means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.11. "Source Code" means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor's choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. "You" (or "Your") means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2. Source Code License.

University of Pretoria etd – Peer, E S (2005)

2.1. The Initial Developer Grant. The Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

(a) under intellectual property rights (other than patent or trademark) Licensable by Initial Developer to use, reproduce, modify, display, perform, sublicense and distribute the Original Code (or portions thereof) with or without Modifications, and/or as part of a Larger Work; and

(b) under Patents Claims infringed by the making, using or selling of Original Code, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the Original Code (or portions thereof).

(c) the licenses granted in this Section 2.1(a) and (b) are effective on the date Initial Developer first distributes Original Code under the terms of this License.

(d) Notwithstanding Section 2.1(b) above, no patent license is granted: 1) for code that You delete from the Original Code; 2) separate from the Original Code; or 3) for infringements caused by: i) the modification of the Original Code or ii) the combination of the Original Code with other software or devices.

2.2. Contributor Grant.

Subject to third party intellectual property claims, each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license

(a) under intellectual property rights (other than patent or trademark) Licensable by Contributor, to use, reproduce, modify, display, perform, sublicense and distribute the Modifications created by such Contributor (or portions thereof) either on an unmodified basis, with other Modifications, as Covered Code and/or as part of a Larger Work; and

(b) under Patent Claims infringed by the making, using, or selling of Modifications made by that Contributor either alone and/or in combination with its Contributor Version (or portions of such combination), to make, use, sell, offer for sale, have made, and/or otherwise dispose of: 1) Modifications made by that Contributor (or portions thereof); and 2) the combination of Modifications made by that Contributor with its Contributor Version (or portions of such combination).

(c) the licenses granted in Sections 2.2(a) and 2.2(b) are effective on the date Contributor first makes Commercial Use of the Covered Code.

University of Pretoria etd – Peer, E S (2005)

(d) Notwithstanding Section 2.2(b) above, no patent license is granted: 1) for any code that Contributor has deleted from the Contributor Version; 2) separate from the Contributor Version; 3) for infringements caused by: i) third party modifications of Contributor Version or ii) the combination of Modifications made by that Contributor with other software (except as part of the Contributor Version) or other devices; or 4) under Patent Claims infringed by Covered Code in the absence of Modifications made by that Contributor.

3. Distribution Obligations.

3.1. Application of License.

The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2. The Source Code version of Covered Code may be distributed only under the terms of this License or a future version of this License released under Section 6.1, and You must include a copy of this License with every copy of the Source Code You distribute. You may not offer or impose any terms on any Source Code version that alters or restricts the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5.

3.2. Availability of Source Code.

Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available; and if made available via Electronic Distribution Mechanism, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular Modification has been made available to such recipients. You are responsible for ensuring that the Source Code version remains available even if the Electronic Distribution Mechanism is maintained by a third party.

3.3. Description of Modifications.

You must cause all Covered Code to which You contribute to contain a file documenting the changes You made to create that Covered Code and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Code provided by the Initial Developer and including the name of the Initial Developer in (a) the Source Code, and (b) in any notice in an Executable version

University of Pretoria etd – Peer, E S (2005)

or related documentation in which You describe the origin or ownership of the Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims.

If Contributor has knowledge that a license under a third party's intellectual property rights is required to exercise the rights granted by such Contributor under Sections 2.1 or 2.2, Contributor must include a text file with the Source Code distribution titled "LEGAL" which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If Contributor obtains such knowledge after the Modification is made available as described in Section 3.2, Contributor shall promptly modify the LEGAL file in all copies Contributor makes available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the Covered Code that new knowledge has been obtained.

(b) Contributor APIs.

If Contributor's Modifications include an application programming interface and Contributor has knowledge of patent licenses which are reasonably necessary to implement that API, Contributor must also include this information in the LEGAL file.

(c) Representations.

Contributor represents that, except as disclosed pursuant to Section 3.4(a) above, Contributor believes that Contributor's Modifications are Contributor's original creation(s) and/or Contributor has sufficient rights to grant the rights conveyed by this License.

3.5. Required Notices.

You must duplicate the notice in Exhibit A in each file of the Source Code. If it is not possible to put such notice in a particular Source Code file due to its structure, then You must include such notice in a location (such as a relevant directory) where a user would be likely to look for such a notice. If You created one or more Modification(s) You may add your name as a Contributor to the notice described in Exhibit A. You must also duplicate this License in any documentation for the Source Code where You describe recipients' rights or ownership rights relating to Covered Code. You may choose

University of Pretoria etd – Peer, E S (2005)

to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Code. However, You may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear than any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Distribution of Executable Versions.

You may distribute Covered Code in Executable form only if the requirements of Section 3.1-3.5 have been met for that Covered Code, and if You include a notice stating that the Source Code version of the Covered Code is available under the terms of this License, including a description of how and where You have fulfilled the obligations of Section 3.2. The notice must be conspicuously included in any notice in an Executable version, related documentation or collateral in which You describe recipients' rights relating to the Covered Code. You may distribute the Executable version of Covered Code or ownership rights under a license of Your choice, which may contain terms different from this License, provided that You are in compliance with the terms of this License and that the license for the Executable version does not attempt to limit or alter the recipient's rights in the Source Code version from the rights set forth in this License. If You distribute the Executable version under a different license You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or any Contributor. You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works.

You may create a Larger Work by combining Covered Code with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation.

   If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Code due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b)

University of Pretoria etd – Peer, E S (2005)

describe the limitations and the code they affect. Such description must be included in the LEGAL file described in Section 3.4 and must be included with all distributions of the Source Code. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Application of this License.

   This License applies to code to which the Initial Developer has attached the notice in Exhibit A and to related Covered Code.

6. Versions of the License.

   6.1. New Versions.

   Netscape Communications Corporation ("Netscape") may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number.

   6.2. Effect of New Versions.

   Once Covered Code has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Covered Code under the terms of any subsequent version of the License published by Netscape. No one other than Netscape has the right to modify the terms applicable to Covered Code created under this License.

   6.3. Derivative Works.

   If You create or use a modified version of this License (which you may only do in order to apply it to code which is not already Covered Code governed by this License), You must (a) rename Your license so that the phrases "Mozilla", "MOZILLAPL", "MOZPL", "Netscape", "MPL", "NPL" or any confusingly similar phrase do not appear in your license (except to note that your license differs from this License) and (b) otherwise make it clear that Your version of the license contains terms which differ from the Mozilla Public License and Netscape Public License. (Filling in the name of the Initial Developer, Original Code or Contributor in the notice described in Exhibit A shall not of themselves be deemed to be modifications of this License.)

7. DISCLAIMER OF WARRANTY.

   COVERED CODE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE

University of Pretoria etd – Peer, E S (2005)

IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED CODE IS WITH YOU. SHOULD ANY COVERED CODE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

8. TERMINATION.

   8.1. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Covered Code which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

   8.2. If You initiate litigation by asserting a patent infringement claim (excluding declatory judgment actions) against Initial Developer or a Contributor (the Initial Developer or Contributor against whom You file such action is referred to as "Participant") alleging that:

   (a) such Participant's Contributor Version directly or indirectly infringes any patent, then any and all rights granted by such Participant to You under Sections 2.1 and/or 2.2 of this License shall, upon 60 days notice from Participant terminate prospectively, unless if within 60 days after receipt of notice You either: (i) agree in writing to pay Participant a mutually agreeable reasonable royalty for Your past and future use of Modifications made by such Participant, or (ii) withdraw Your litigation claim with respect to the Contributor Version against such Participant. If within 60 days of notice, a reasonable royalty and payment arrangement are not mutually agreed upon in writing by the parties or the litigation claim is not withdrawn, the rights granted by Participant to You under Sections 2.1 and/or 2.2 automatically terminate at the expiration of the 60 day notice period specified above.

   (b) any software, hardware, or device, other than such Participant's Contributor Version, directly or indirectly infringes any patent, then any rights granted to You by

University of Pretoria etd – Peer, E S (2005)

such Participant under Sections 2.1(b) and 2.2(b) are revoked effective as of the date You first made, used, sold, distributed, or had made, Modifications made by that Participant.

8.3. If You assert a patent infringement claim against Participant alleging that such Participant's Contributor Version directly or indirectly infringes any patent where such claim is resolved (such as by license or settlement) prior to the initiation of patent infringement litigation, then the reasonable value of the licenses granted by such Participant under Sections 2.1 or 2.2 shall be taken into account in determining the amount or value of any payment or license.

8.4. In the event of termination under Sections 8.1 or 8.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or any distributor hereunder prior to termination shall survive termination.

9. LIMITATION OF LIABILITY.

   UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL YOU, THE INITIAL DEVELOPER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED CODE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

10. U.S. GOVERNMENT END USERS.

    The Covered Code is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995).

University of Pretoria etd – Peer, E S (2005)

Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein.

11. MISCELLANEOUS.

    This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by California law provisions (except to the extent applicable law, if any, provides otherwise), excluding its conflict-of-law provisions. With respect to disputes in which at least one party is a citizen of, or an entity chartered or registered to do business in the United States of America, any litigation relating to this License shall be subject to the jurisdiction of the Federal Courts of the Northern District of California, with venue lying in Santa Clara County, California, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

12. . RESPONSIBILITY FOR CLAIMS.

    As between Initial Developer and the Contributors, each party is responsible for claims and damages arising, directly or indirectly, out of its utilization of rights under this License and You agree to work with Initial Developer and Contributors to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability.

13. MULTIPLE-LICENSED CODE.

    Initial Developer may designate portions of the Covered Code as Multiple-Licensed. Multiple-Licensed means that the Initial Developer permits you to utilize portions of the Covered Code under Your choice of the NPL or the alternative licenses, if any, specified by the Initial Developer in the file described in Exhibit A.

EXHIBIT A - Mozilla Public License.

```
``The contents of this file are subject to the Mozilla Public
License Version 1.1 (the ``License''); you may not use this file
except in compliance with the License. You may obtain a copy of
```

University of Pretoria etd – Peer, E S (2005)

the License at

http://www.mozilla.org/MPL/

Software distributed under the License is distributed on an ‘‘AS
IS’’ basis, WITHOUT WARRANTY OF

ANY KIND, either express or implied. See the License for the
specificlanguage governing rights and limitations under the License.

The Original Code is ______________________________________.

The Initial Developer of the Original Code is

________________________. Portions created by

______________________ are Copyright (C) ______

_______________________. All Rights

Reserved.

Contributor(s): ______________________________________.

Alternatively, the contents of this file may be used under the
terms of the _____ license (the [___] License), in which case
the provisions of [______] License are applicable instead of
those above. If you wish to allow use of your version of this
file only under the terms of the [____] License and not to allow
others to use your version of this file under the MPL, indicate
your decision by deleting the provisions above and replace
them with the notice and other provisions required by the [___]
License. If you do not delete the provisions above, a recipient
may use your version of this file under either the MPL or the
[___] License.’’

University of Pretoria etd – Peer, E S (2005)

[NOTE: The text of this Exhibit A may differ slightly from the text of the notices in the Source Code files of the Original Code. You should use the text of this Exhibit A rather than the text found in the Original Code Source Code for Your Modifications.]

E.10 Open Software License (OSL)

Version 2.1

This Open Software License (the "License") applies to any original work of authorship (the "Original Work") whose owner (the "Licensor") has placed the following notice immediately following the copyright notice for the Original Work:

```
Licensed under the Open Software License version 2.1
```

1. Grant of Copyright License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license to do the following:
   - to reproduce the Original Work in copies;
   - to prepare derivative works ("Derivative Works") based upon the Original Work;
   - to distribute copies of the Original Work and Derivative Works to the public, with the proviso that copies of Original Work or Derivative Works that You distribute shall be licensed under the Open Software License;
   - to perform the Original Work publicly; and
   - to display the Original Work publicly.
2. Grant of Patent License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, to make, use, sell and offer for sale the Original Work and Derivative Works.
3. Grant of Source Code License. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor hereby agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original

University of Pretoria etd – Peer, E S (2005)

Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work, and by publishing the address of that information repository in a notice immediately following the copyright notice that applies to the Original Work.

4. Exclusions From License Grant. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior written permission of the Licensor. Nothing in this License shall be deemed to grant any rights to trademarks, copyrights, patents, trade secrets or any other intellectual property of Licensor except as expressly stated herein. No patent license is granted to make, use, sell or offer to sell embodiments of any patent claims other than the licensed claims defined in Section 2. No right is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under different terms from this License any Original Work that Licensor otherwise would have a right to license.

5. External Deployment. The term "External Deployment" means the use or distribution of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether the Original Work or Derivative Works are distributed to those persons or made available as an application intended for use over a computer network. As an express condition for the grants of license hereunder, You agree that any External Deployment by You of a Derivative Work shall be deemed a distribution and shall be licensed to all under the terms of this License, as prescribed in section 1(c) herein.

6. Attribution Rights. You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7. Warranty of Provenance and Disclaimer of Warranty. Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are

University of Pretoria etd – Peer, E S (2005)

owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately proceeding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to Original Work is granted hereunder except under this disclaimer.

8. Limitation of Liability. Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to any person for any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to liability for death or personal injury resulting from Licensor's negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

9. Acceptance and Termination. If You distribute copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. Nothing else but this License (or another written agreement between Licensor and You) grants You permission to create Derivative Works based upon the Original Work or to exercise any of the rights granted in Section 1 herein, and any attempt to do so except under the terms of this License (or another written agreement between Licensor and You) is expressly prohibited by U.S. copyright law, the equivalent laws of other countries, and by international treaty. Therefore, by exercising any of the rights granted to You in Section 1 herein, You indicate Your acceptance of this License and all of its terms and conditions. This License shall terminate immediately and you may no longer exercise any of the rights granted to You by this License upon Your failure to honor the proviso in Section 1(c) herein.

10. Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any

University of Pretoria etd – Peer, E S (2005)

licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11. Jurisdiction, Venue and Governing Law. Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of the U.S. Copyright Act, 17 U.S.C. ?? 101 et seq., the equivalent laws of other countries, and international treaty. This section shall survive the termination of this License.

12. Attorneys Fees. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13. Miscellaneous. This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14. Definition of "You" in This License. "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15. Right to Use. You may use the Original Work in all ways not otherwise restricted or conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

This license is Copyright © 2003-2004 Lawrence E. Rosen. All rights reserved. Permission is

University of Pretoria etd – Peer, E S (2005)

hereby granted to copy and distribute this license without modification. This license may not be modified without the express written permission of its copyright owner.