Shadowboard: an Agent Architecture for enacting a sophisticated Digital Self

Steven Brady Goschnick

September, 2001

Submitted in total fulfilment of

the requirements of the degree of

Master Engineering Science

Department of Computer Science and Software Engineering

University of Melbourne

AUSTRALIA

Abstract

In recent years many people have built Personal Assistant Agents, Information Agents and the like, and have simply added them to the operating system as auxiliary applications, without regard to architecture. This thesis argues that an agent architecture, one designed as a sophisticated representation of an individual user, should be embedded deep in the device system software, with at least equal status to the GUI – the graphical user interface. A sophisticated model of the user is then built, drawing upon contemporary Analytical Psychology – the Psychology of Subselves. The Shadowboard Agent architecture is then built upon that user model, drawing both structural and computational implications from the underlying psychology. An XML DTD file named Shadowboard.dtd is declared as a practical manifestation of the semantics of Shadowboard. An implementation of the Shadowboard system is mapped out, via a planned conversion of two existing integrated systems: SlimWinX, an event-driven GUI system; and XSpaces, an object-oriented tuplespace system with Blackboard-like features. The decision making mechanism passes logic terms and contraints between the various sub-agent components (some of which take on the role of Constraint Solvers), giving this agent system some characteristics of a Generalised Constraint Solver. A Shadowboard agent (built using the system) consists of a central controlling autonomous agent named the Aware Ego Agent, and any number of sub-agents, which collectively form an integrated but singular whole agent modelled on the user called the Digital Self. One such whole-agent is defined in a file named DigitalSelf.xml – which conforms to the schema in Shadowboard.dtd - which offers a comprehensive and generic representation of a user's stance in a 24x7 network, in particular - the Internet. Numerous types of Shadowboard sub-agents are declared.

ACKNOWLEDGEMENTS

I deeply appreciate the help, support and encouragement of the following people leading into and during various stages of this research. I thank my supervisor Professor Leon Sterling for his guidance, support and faith in me throughout the course of this thesis. For his "suspending disbelief" when needed, and for encouraging me with the Shadowboard direction of the thesis at a crucial time, when numerous others would likely have discouraged it. I also thank Leon for directing me toward several preparitory postgraduate subjects in the department including *Advanced AI* and *Intelligent Software Agents* – the second, which he taught at the time, being the most enjoyable subject I have formally studied to date. I thank Professor Rao Kotagiri for his guidance, for taking me seriously in the early days of this research, and for his reminder of the privilege of fulltime research and the obligations to society it carries.

I thank Peter Gerrand for suggesting I do this Research Masters in the first place and in directing me to Leon Sterling and Peter Rule. I thank Peter Rule at Ericsson Australia for opening his door to me on several occasions, and for supporting my application for a SPIRT grant – the success of which made this thesis possible.

Leon's humility, calm and tolerance were evident from our first meeting, when I was concluding a demonstration of my early SlimWinX system on his Apple MAC and tried to extract my disk. Unaccustomed to that model of the MAC I inadvertantly pressed the 'off' button, positioned immediately near the floppy disk drive. His response to my shutting down of his desktop system was: *"Mmmm. That's a bad piece of MAC interface design I hadn't thought about until now."*

With regard to those aspects of Psychology drawn upon within this thesis, I would particularly like to thank three people: graphic designer and calligrapher Rodney Saulwick for handing me a copy of Jung's book *Man and His Symbols* with the words *"you're ready for this"*; Dr Hal Stone for forwarding me a copy of his earliest book *Embracing Heaven and Earth* – and Mary King for delivering it and Hal and Sidra's brand of Psychology when I needed them most.

I thank Professor Liz Sonenberg for her encouragement, support and her fast, insightful comments on several drafts of this thesis, and also for the opportunity and pleasure in co-teaching *Interactive System Design* as a subject with her in 1998 and 1999. She remains an inspiration to me as a scholar and a teacher. Also in the Department of Information Systems, I thank my colleagues at the IDEA Lab, Dr Steve Howard for his helpful and concise comments on the structure of this thesis, and Connor Graham for some useful comments on its contents.

My colleagues within the Intelligent Agent Lab (past and present) - particularly Clinton Heinze, Chris Wright, Adrian Pearce, Simon Goss, Andrew Cassin, Seng Wai Loke, David Kinny, Maya Hristozova, David Morley, Sharon Xiaoying Gao, Bohdan Durnota, Wayne Wobcke and Emma Norling – I thank for the various degrees of generousity in sharing ideas about agents, via the conversations, the seminars, the papers, the reflections and occasionally as sounding boards. In this context too, I thank both Leon Sterling and Liz Sonenberg as instigators and co-directors of the Intelligent Agent Lab, for the agent community it has encompassed and for the precession of international Agent luminaries that they have enticed to the lab as visitors, including: Bob Kowalski, Michael Wooldridge, Frank Dignum, Klaus Fischer, and others - all influential, all inspiring in their own ways.

More personally and over the long term, I thank my parents, my four brothers and long time good friend Lyn Phillips, for giving me a solid foundation in life. I thank my children and very dear friends Lauren and Tim for much indeed: for hope, for the pleasure they bring, for keeping me well-grounded throughout good times and bad, and for keeping my sense of humour always close at hand. Last and most of all I thank Christine Sun my partner in life for her unfailing love, faith, encouragement and her support of me in doing this whole thing the way it should be done - *in pursuing the dream*.

Table of Contents

List of Figures

List of Tables

Preface

Preliminary ideas on the Shadowboard Architecture, including drawing analogies from Analytical Psychology and the intention to base its decision making mechanism on a Generalised Constraint Solver, were published in the Proceedings of PRIMA 2000 [31], and as a Technical Report in the Department of Computer Science and Software Engineering, the University of Melbourne [32].

The material in Chapters 5, 6 and 7 are original contributions, except only for the previous development on the systems SlimWinX and XSpaces by the author - which are documented in private documents cited in this thesis.

To the best of my knowledge, this thesis contains no material previously published or written by another person except where references have been made in the text of this thesis.

None of the work presented in this thesis has been submitted for the award of any other qualification at college, institute or university.

The work in this thesis has not been published elsewhere except as noted above.

1 INTRODUCTION

Rather than simply having a GUI (Graphic User Interface) interface to the computer, what we now want as users of computers connected to the Internet 24 hours, 7 days a week (24x7), is an *interface to the network*. We want to be notified of information that is important to us, but be spared the distractions of the mundane. We want to be accurately *represented* in our global community as we sleep, and be able to take in information and process it subliminally from the net, as we go about our physical lives away from the wire. We want software to take the initiative and make decisions on our behalf, without waking us from our sleep or distracting us from the things we prefer to consciously focus on - but only those decisions that can be clearly made on our behalf such as *"No I can't go to the Melbourne concert as I'll be in Sydney that day."*

In the past few years there have been many people building intelligent software agents – e.g. Personal Assistant Agents which may help a user by filtering and responding to email; Information Agents, which may retrieve specific information from the Internet, made useful to the user by retrieving only information that fits a user's preferences. The developers of such agents have simply been adding them to the workstation operating system as auxiliary applications, without regard to architecture [45]. This thesis takes the view that an agent architecture, one suitably designed to include user interface functionality, should be embedded deep in the workstation system software. The concept is to build upon that architecture a central controlling agent, together with any number of sub-agents. These sub-agents may or may not have accompanying visual personifications within the GUI - i.e. visually displayed agents within the genres known as *Believable Agents* (a.k.a. anthropomorphic agents), *Social Agents* (communicate with humans and/or other agents) and *Emotional Agents* (express emotional indicators via multimedia techniques, as a part of their communication, and may try to sense the user's emotional state) [19].

This central controlling agent, together with the various subordinate sub-agents, form a singular *whole agent*, which is a sophisticated representation of the user, to

some degree. Hereafter in this thesis, this sophisticated representation of the user is termed the *Digital Self*.

At the workstation interface, there is a one-to-one relationship between an individual user and his/her Digital Self, which in turn embodies and manages the *user interface to the network.* The term *network* used here refers to the rest of the user's community, usually the Internet in total, although it could be some lesser network. The network connection of most interest here is the so-called 24x7 - a continuous connection, 24 hours per day, 7 days per week. Many of the sub-agents will perform tasks that involve the user's interaction with servers out on the network, during the user's presence but also in his/her absence. The Digital Self agent needs to hold a sophisticated model of the user, one capable of acting as a *proxy* for the user as an individual in 24x7, and for that it is reliant on the underlying agent architecture.

There are several well developed agent frameworks built upon abstractions drawn from psychology. In an overview of Agents as *rational systems*, Wooldridge and Jennings [82] put in context the use of mentalistic notions drawn from psychology, as legitimate *abstractions* for use within computer science:

> *'Much of computer science is concerned with good abstraction mechanisms, since these allow system developers to manage complexity with greater ease: witness procedural abstraction, abstract data types, and most recently objects. So why not use the intentional stance as an abstracting tool in computer science.'*

Indeed, a commonly used phrase to describe software agents to object-oriented practitioners new to the agent–oriented paradigm is that they are '*objects with attitude*'. Dynamic Objects and Active Objects are two forms of non-deterministic systems drawn from the Object Oriented Programming (OOP) paradigm that have some aspects in common with the Agent Oriented Paradigm.

The motivations for building an agent architecture into the base level of the human-computer-interface (HCI) include:

- To help the user manage his/her workstation or set-top box as an entity continuously connected to a network and open to ongoing change.

- To subliminally gather information and knowledge appropriate to the user's interests in 24x7 and present them to the user at convenient or the most appropriate times.

- To accommodate active objects and mobile agents, taking up some form of residence within or relationship with the local workstation system.

- To accommodate and manage numerous and varying numbers of Personal Assistant Agents and other forms of agent within the local workstation or device, which are often acting as various proxies for the user.

- To allow continual management and presentation of the user interface as a sophisticated *personification* of the individual user to the outside world.

However, any agent or sub-agent within such a system should not be seen simply as a surrogate of the user for situations when the user is not there. Nor should the architectural framework be seen as just a house to hold "digital butlers" in the manner that Negroponte [52] first perceived personal assistants which actively service a user's needs and goals. The Digital Self *augments* the user's skills and abilities, in a synergy of human and software based qualities, increasing the individual's effectiveness, efficiency and influence in an empowering manner. The closer the model of the user gets to the real user, the greater will be that synergy.

In the past, when a computer was viewed as a passive element in our reality, we got definitions of *interactive systems* that box computers into a passive role, like this one by Newman and Lamming [54]: *'the most crucial property of any interactive system is its* ***support for*** *human activity'*. Recent advances in interactive systems in the form of wearable devices, such as the work done at the MIT Media Lab [5], augment the user's reality in a demonstrable and active way. One example of augmented reality involves overlaying extra context-sensitive information - it might be the brightness level of the subject they are currently looking at - into the view of the human wearer of a small screen, which sits only inches forward of the eyes. It is placed at such a point, that the information on the screen appears in focus within their view of the real scene before them. From the user's perspective, augmented reality done well is as if they have a sixth sense, or more. Such wearable devices demonstrate a symbiotic

relationship between man and machine, *empowering* the human user rather than just *supporting* the human user.

With such wearable computers, the conceptual jump from viewing computers as *'supporting human activity'* to viewing computers as *'empowering the user'* is not a difficult one to make, given the placement of extra situational information for the user's immediate benefit. But such a jump has been a larger one that many will not make when looking at workstations running standard GUI interfaces. A contention of this thesis is that the embedding of a sophisticated agent architecture at the base of the workstation or device interface, will benefit the user to such a degree that most people will then readily agree, that the computer empowers the individual, whether it be a desktop computer, a set-top box, a PDA or otherwise. They will feel more effective, more responsive, and represented in a more integrated way as they interact with their immediate communities and the larger world via global networks connected in 24x7 time.

1.1 Contributions

This thesis makes the following contributions to the relationship between the user, the user interface and a supporting cast of agents, in the 24x7 mode of computer usage:

- A new agent architecture called Shadowboard is drawn from a contemporary stream of Analytical Psychology called the *Psychology of Subselves* - Analytical Psychology, as a model has not previously been cited in Agent research. Apart from analogy, the Psychology of Subselves is also used to draw structural and computational implications for the resulting agent system. The structure is presented visually, textually and as an XML DTD file (a schema) named *Shadowboard.xml*. The Shadowboard architecture is purpose-built to enact a Digital Self in the user's network interface.

- A blueprint for a complex Digital Self is specified in an XML file named *DigitalSelf.xml*, and a generic comprehensive list of *Shadowboard Agent Types* is developed, which encompass the numerous roles and parts that an individual user is likely to want in his/her 24x7 Digital Self.

- An implementation of a Shadowboard system is detailed, one that takes a Generalised Constraint Solver approach to decision making. It is derived from an earlier event-handling system, which moved event messages between active objects as currency. The implementation of Shadowboard upon it, replaces the currency from *event messages* to *goals* (expressed in predicate logic terms) and *constraints*, and replaces the receiving objects with sub-agents and groups of sub-agents called *envelope-of-capability*.

1.2 Overview of the Thesis

Beyond this introductory chapter the thesis is organised as follows:

- Chapter 2 covers technical background of server-side technologies and programming paradigms which have direct relevance to the thesis – in particular with a focus on dynamic and non-deterministic behaviour. The backgrounds are generally not in much detail, as the intention in this document is to give just enough detail to sustain the explanations in the following sections, but point elsewhere for fuller treatments. Chapter 2 is broken down into: Section 2.1 A definition of Dynamic Systems; Section 2.2 which emphasises the Internet's contribution to dynamic systems; Section 2.3 an overview of the Object Oriented Paradigm (OOP); Section 2.4 an overview of some relevant Distributed Systems with a focus on contemporary variants of the Blackboard and Tuplespace models; Section 2.5 is a brief look at Logic Programming and Constraint Logic Programming; Section 2.6 is a brief but panoramic view of the *Agent Oriented Programming* paradigm – as it probably represents the most non-deterministic paradigm for development of dynamic systems to date.

- Chapter 3 introduces some Human Computer Interface (HCI) issues, architectures and client-side object models; both as a lead-in to *user models* and *models of mind* discussed in Chapter 4, and as background to the interface aspects of Shadowboard – the agent architecture resulting from this research.

- Chapter 4 is a brief coverage of 100 years of Analytical Psychology from Freud and Jung, to contemporary concepts of the Psychology of Subselves, with a view to it as a model of the user, upon which the Shadowboard architecture is based.

- Chapter 5 presents the *Shadowboard Architecture*, putting technical terms and concepts in place of the psychological terms and concepts presented in Chapter 4. It also includes a tangible manifestation of the architecture, represented structurally both in graphical form and in an XML document definition file: *Shadowboard.dtd*.

- Chapter 6 outlines an implementation strategy for building a Shadowboard system upon XSpaces and SlimWinX - earlier works by the author, introduced in Chapters 2 and 3, with further detail in the early half of Chapter 6. Computational implications from the underlying psychology are used to guide the implementation details. Logic programming and constraint solver concepts are utilised to provide the basic computation mechanism within the overall system.

- In Chapter 7 we take the achieved Shadowboard architecture from Chapter 5, assume an implementation of a working system by following the prescription in Chapter 6, then re-address the concept of building a Digital Self as introduced in Chapter 1. An extensive list of possible subselves is compiled, which are then rendered in an XML file named DigitalSelf.xml, one which conforms to the Shadowboard.dtd. Other issues are covered at the end of this chapter including: compliance to the W3C DOM (Document Object Model) standard as a future research option; and a method for training the sub-agents within Shadowboard, based on CLS – the Concept Learning Scheme.

- In Chapter 8 we discuss some contemporary issues in agency and multi-agent systems (MAS) including *autonomy* and *trust*, and draw some conclusions with the hindsight of the earlier chapters. We outline future areas for research. We claim that a Digital Self based on a Shadowboard system will significantly improve the symbiotic relationship between man and machine. With regard to social implications, we note the potential for increased self-awareness and growth of the user, within the context of modelling and then building an operational Digital Self. We also look to the potential economic benefits of Shadowboard.

2 BACKGROUND

2.1 Dynamic Systems

With regard to dynamic systems in general, Luenberger [47] gives us this definition of the word *dynamic*:

> *"The term dynamic refers to phenomena that produce time-changing patterns, the characteristics of the pattern at one time being interrelated with those at other times. The term is nearly synonymous with time-evolution or pattern of change. It refers to the unfolding of events in a continuing evolutionary process."*

For Luenberger, a mathematician, *dynamic* has a dual meaning: the first is the use of the term dynamic to describe the phenomena in the world around us; the second, is that part of mathematical science which is used for representation and analysis of such phenomena. Most commonly, the mathematical tools used to deal with dynamic phenomena are either *differential equations* or *difference* equations. The choice of one over the other depends on whether the phenomena under study are advantageously viewed as occurring in either continuous or discrete time, respectively.

To the computer scientist there is a third meaning of *dynamic*: it refers to those aspects of computer systems (themselves, making up more and more of the world around us) which behave in a manner beyond a pre-canned or compiled way. To the object-oriented specialist, dynamic behaviour is seen in dynamic binding of objects and the instantiation of objects from object factories at runtime. Active Objects are seen as dynamic, in that they are busy doing things in themselves, regardless of other processes making calls upon their methods. To the user interface specialist, it is seen in adaptive intelligent interfaces. To the software agent specialist, dynamic behaviour is seen in the reactive, proactive and deliberative behaviour of software agents, which at its zenith, can be non-deterministic, particularly in the absence of a human user. The growth and evolution of the Internet presents many other areas of interest to computer science where *dynamic behaviour* is manifesting itself in new ways. The

treatment of dynamic systems in mathematics, gives us a basic foundation for tools to deal with these new dynamic behaviours.

In the realm of mathematical systems as one moves focus from the *analysis* of a dynamic system to that of influencing the *behaviour* of a system, by either control or design, one gains much by employing Control Theory.

Control theorists, as with computer scientists, are interested in inputs and outputs of complex systems. A modern approach used in control theory is to describe the behaviour of the entire system in a *state-space* description.

A *state vector* u(k) of n variables $x_1(k)$, $x_2(k)$, … $x_n(k)$ in discrete time systems represents a complete description of the system, at time k. The state vector serves as a kind of running collection of initial conditions, in a system in which one is interested in determining future behaviour. The *state space* is an n-dimensional space in which the state vector is defined. The evolution of a dynamic system can be visualised by the movement of the *state vector* within *state space*. In computer science, the concept of a *tuplespace* grew upon the foundation of *state space* and *state vectors*.

We cover the concept of a tuplespace and some recent systems built upon it further down. However before that, we will introduce a few synergies at work in some areas of study relevant to this one, most of which are related to the Internet and the World Wide Web (www).

2.2 Internet+Bandwidth = Pervasive Dynamic Systems

Within the era of commonly available commodity priced network infrastructure, object-oriented technologies have become available on the server-side of computing in the form of distributed component systems, such as: CORBA [14], DCOM/COM+ [16] and Enterprise Java Beans [21].

Meanwhile, on the client-side, user-centric interactive graphic applications have been evolving too, driven by the desire for usability and flexibility, and primarily delivered through event-driven, object-oriented GUI technologies. Although the

prevalent interactive systems in the marketplace are proprietary - e.g. Microsoft Windows; Apple Mac OS - the world wide web browser has fostered another source for object-oriented (OO) standards affecting interaction on the client-side. Quickly evolving versions of web browser technology from several suppliers, has culminated in the Document Object Model (DOM) specification from the Internet W3C organisation [83] - an *open* client-side object standard.

As the network bandwidth generally available increases several orders of magnitude (broadband), and local storage capacity of workstation and set-top boxes continues to increase, the advantages and disadvantages of technologies coming from either end of the Client - Server spectrum, will most likely merge in the following manner:

- the disadvantages of both will be minimised,
- the advantages of both will accumulate,

and pervasive dynamic systems will be upon the general user community.

This network will consist of *environments* (global state spaces) in which information and objects are *placed and retrieved*, and through which *active objects* and *intelligent software agents* will navigate and negotiate. This view of the emerging global network is an underlying assumption of this thesis, and the Shadowboard architecture and its implementation are aimed at positively contributing to such a globally shared network space.

What follows are those various technical aspects of server-side objects, client-side objects and their representations, and of agent-oriented concepts and technologies, which have some relevance to this thesis.

2.3 The Object Oriented Paradigm

The object-oriented paradigm of programming (OOP) employs abstraction of objects to solve problems in reusable ways. The driving force behind the paradigm was to increase programmer productivity. In an object-oriented language one must be able to instantiate *objects* of a particular type, in order to build reusable software components. The *type* of an object is termed a *class* of objects and the programmer can

make any number of such classes of his/her own, beyond the classes that come packaged with the language. A *class* is like a pattern or a mould used to instantiate multiple individual objects of like kind. They are alike in terms of their common capabilities and behaviours which are accessible via method calls, but they differ in terms of their state which is generally held in the internal data of each individual object - except for *static* data types, which hold information that is common to the whole class.

It is worthwhile here to look at specific example code of two significant OOP languages - C++ and Java - as they exemplify many common features of OOP languages, but also highlight several significant differenences in implementing the OOP paradigm. The Java code below in Figure 2.1 performs two functions with respect to the class *Person*: it is both a *declaration* and a *definition* of the class.

```
class Person
{
      public Person (String family, String first)
      {
        familyName     =  family;
        firstName      =  first;
      }
      public void ChangeFamilyName (String newName)
      {
        previousName = familyName;
        familyName   = newName;
        previousName = " ";
      }
      public void print()
      {
        System.out.println(firstName + " " + familyName);
      }

      protected String familyName;
      protected String firstName;
      protected String previousName;
}
```

Figure 2.1 - Declaration and definition of an object class called Person in Java.

The internal state of an object is held in its data *attributes* - for example, *familyName, firstName* and *previousName* in the above code fragment, are all data attributes of type String (itself, an array of type char) belonging to the class Person. Apart from being declared as either of the implicit data types such as - int (integer), byte, float (real), char (character), etc - attributes may alternatively be objects themselves, of some other class definition.

2.3.1 Object-Oriented Programming (OOP)

In addition to the basic ability to define classes of new objects, there are three other properties programming language must have to be called object-oriented:

1. Encapsulation
2. Inheritance
3. Polymorphism

As these concepts are central to the OOP paradigm they are each covered in some detail with examples below.

2.3.1.1 Encapsulation

Encapsulation - also called *data-hiding* - is the way in which OOP allows objects to become black-boxes during operation. The binding together of *data*, together with algorithms within *methods* for manipulating that data, into an object, is termed *encapsulation*. Changing the internal state of an object is done by invoking a method by *messaging* the object, with a call to the object in question. Eg.

```
Person person1 = new Person("Smith", "Jill");
person1.ChangeFamilyName("Glockenspiel");
```

The first line of code in this code fragment defines a new *object* called **person1** of *class* **Person**, (as defined in Figure 2.1) which calls the *constructor* with the name Jill Smith. The second line is a *message* to the object **person1**, asking to change Jill Smith's family name to *'Glockenspiel'* by invoking the *method* **ChangeFamilyName**. The public methods represent the public interface to the class, they are *public* in the sense that

they can be invoked outside of the scope of the class by sending messages to their methods.

Data-hiding is considered to be one of the advances of OOP over the earlier paradigm of *modular programming*. It is interesting to compare the basic notion of a program in OOP with one as defined in modular programming, as they both involve the level of containment of *data* and *algorithm*.

- Modular (procedural) programming: *data* + *algorithm* = *program*.
- OOP: *data* + *algorithm* = *object*; *object messaging object* = *program*.

From the perspective of data encapsulation, the difference between OOP and earlier modular programming languages such as PASCAL, is an issue of granularity.

2.3.1.2 Inheritance

Inheritance is the general concept of extending a base class, to some new class of objects with some capabilities evolved beyond those of the base parent class. The new class uses what is there if appropriate, overloads (replaces) what is not useful, and incorporates additions where there was no previous but needed ability. E.g.

```
class Student extends Person
{
       public Student(String family, String first, int level)
      {
         super(family, first);
         currentLevel = level;
      }
      protected int currentLevel;
          // 1,2 or 3 for primary, secondary and tertiary.
}
```

Figure 2.2 - Java inheritance: class Student inherits all from Person and extends.

A *super class* is the *parent class* of another class in Java terminology. In the above example code, the code - `super(family, first)` - refers to the constructor of the parent class. I.e. `Person(family, first)`.

A sub-class is the *evolved (child) class* of a parent class. *Evolved objects* are useable anywhere that the *parent object* may be used, but the converse cannot be true.

Note that the *Student* class has an additional data attribute - *currentLevel*, which represents a person's level of education - over and above the parent class Person.

As a point of comparison between Java and C++ which we will draw upon later – referring back to the Java code in Figure 2.1 - to enact the same object that is declared and defined there in a single file, requires two files in the C++ language: one for the *class declaration* (Figure 2.3); and a second for the *class definition* (Figure 2.4).

```
#include <string.h>
class Person
{
  private:
       char * familyName;
       char * firstName;
       char * previousName;
  public:
Person (char * family, char * first);          // the Constructor
       void ChangeFamilyName (char * newName);
       void print();
       ~Person();                              // the Destructor
};
```

Figure 2.3 – *Declaration* of an object class called Person in C++, in interface file *person.h*.

```
#include <person.h>
Person::Person (char * family, char * first)
{
       familyName = "\n";
       firstName  = "\n";
       previousName      = "\n";
}
void Person::ChangeFamilyName (Char * newName)
{
       strncpy(previousName, familyName);
       strncpy(familyName, newName);
}
void Person::print()
```



```
{
printf("%s %s", firstName, familyName);
}
Person::~Person() // Destructor - releases memory.
{
      delete familyName;
      delete firstName;
      delete previousName;
}
```

Figure 2.4 - The definition of an object class called Person in C++, in a file person.cpp.

This clear separation of *declaration* and *definition* for all classes in C++ is necessary for the multiple inheritance feature of the language. Java is only capable of single inheritance and so does not generally separate definition from declaration. For example in C++, if another class is declared in a separate file - Employee.h - like so:

```
class Employee
{
   private:
      int currentLevel;
      int employeeNo ;
      float salary;
   public:
      void setEmployeeNo(int employeeID);
      void setSalaryLevel(float annualSalary);
};
```

Figure 2.5 - The declaration of an object class called Employee in C++.

and the class is defined in a file Employee.cpp elsewhere, then a class called *Student* may inherit from both *Person* and *Employee*, as follows:

```
#include <Student.h>
#include <Employee.h>

class Student public Person, public Employee
{
   public:
      Student(char * family, char * first, int level);
   private:
      protected int currentLevel;
};
```

Figure 2.6 - Multiple Inheritance in C++.

This Student object defined in C++ inherits the attributes and methods of both parent classes, so that statements such as:

```
student1.ChangeFamilyName("Glockenspiel");
student1.setSalaryLevel(632.67);
```

are normal and legal C++. To get a Student class with this same sort of dual functionality of both Person and Employee in Java, is best done via a Java *Interface* construct. A Java Interface behaves like a class in which all methods are *abstract* and must be defined within the class that subscribes to that interface.

Abstract Classes in Java:

The higher one advances up the class hierarchy, the more general becomes the code. During code development the movement of common code by the programmer, back in the direction of the parent class, is essential in good object oriented programming. At some point of generalisation in refining the code, the original ancestor class becomes so generic that it can be considered a mould for other classes, rather than a mould for actual objects. The Java language has a construct for such generic classes called an *abstract class*, for which it uses the keyword *abstract* to identify them.

The keyword *abstract* can be placed in front of method names and in front of the class name. Abstract classes have at least one abstract method i.e. Abstract classes can have some real (implemented) methods. An abstract method acts as a place holder for a method that is to be implemented by the programmer in the evolved classes. Subscribing to an abstract class via inheritance, is a *promise* that all non-abstract classes that derive from this abstract class will implement a real method of the same name with the same number and type of parameters. The promise is enforced by the compiler.

The Java Interface construct:

A Java *interface* behaves like a special abstract class, one in which *all methods* are abstract. Unlike an abstract class, one can add an interface to a class that is already a part of a different inheritance family - in an attempt to overcome some of the disadvantages of not having Multiple Inheritance in Java. An interface declares a whole set of methods that a class *must* implement if it subscribes to that interface.

The implementation code is in each and every class that subscribes to the interface, not in the interface itself. The following Java example is the way in which a programmer would use an interface to give class Student features of both an Employee and a Student:

```
public interface Employee
{
      public void setEmployeeNo(int employeeID);
      public void setSalaryLevel(float annualSalary);
}

class Student extends Person implements Employee // Similar to Multiple
      Inheritance.
{
      public Student(String family, String first, int level)
      {  super(family, first);
         currentLevel = level;
      }
      public void setEmployeeNo(int empNo)
      {  employeeNo = empNo;
      }
      public void setSalaryLevel(float aSalary)
      {  salary = aSalary;
      }
      protected int currentLevel;
      private int employeeNo ;
      private float salary;
}
```

Figure 2.7 - Java Interface example, simulating multiple inheritance.

Note that the extra methods *setEmployeeNo* and *setSalaryLevel*, plus the extra data attributes associated with Employee functionality, have to be coded afresh in every new Java class that subscribes to the Employee interface.

2.3.1.3 Polymorphism and Dynamic Binding

OOP languages like C++ and Java allow a programmer to send an identical message to dissimilar but related objects, letting the individual objects to react in their own way. The objects are related, in that they both inherit from a common base class. The messages are identical only in that they have identical names, and identical number

and types of parameters - called the *signature* of the method. This language feature is termed *polymorphism*. For example, the method *drawYourself()* for two different objects (*circle1* and *square1*) can be called individually or via a common line of code within the *for* statement in the code fragment below:

```
circle1.drawYourself();

square1.drawYourself();

shapesArray[0] = circle1;

shapesArray[1] = square1;

for (int i=0; i < 2; i++ )

{

   shapesArray[i].drawYourself();

}
```

Polymorphism, also known as *dynamic binding*, may be achieved within a language by either *early binding* or *late binding*:

- **Late Binding** enacts polymorphism by *overloading* a particular method. For example, further down in Figure 2.8 there is a definition of a new class called *Graduate*, which inherits from class Student, and in doing so it inherits the method *print()* further up the ancestor chain, in Person as defined in Figure 2.1. The print() method is redefined within *Graduate*, effectively *overloading* the method - it has the same name and same number of parameters - none in this case. The redefinition of a method in a subclass effectively hides the method(s) of the same name and same signature in the ancestor classes. Late binding is so named, as the appropriate overlaid method for a given object in a given piece of code, is determined and bound *at runtime* time rather than at compile time. Late binding is the default in the Java language, but it is the special case in the C++ language, where the programmer's intention is made clear to the compiler via the *virtual* keyword being placed before the method name in the declarative code.

- **Early Binding** occurs when a particular overlaid method is related to a given object in the code, at compile-time rather than at runtime time. Early binding is the default in C++, but in Java the programmer's intention for early binding needs to be made clear by including the *final* keyword.

It is the inheritance and polymorphism features of an OOP language that set the stage for *dynamic objects*. Dynamic objects became possible within the OOP paradigm, due to late-binding advances in compiler technologies - and late binding was necessary to implement polymorphism.

2.3.1.4 Behaviour, State and Identity

An object's *behaviour* is defined by the messages (method calls) it accepts.

An object's *state* is the sum total of its internally held data, representing things such as how it looks and what it knows. State is not fixed, but is the result of instantiation followed by the sequence of messages that have been made upon the object thus far.

Identity includes more than inherited capability and current state, it also includes the object's relative relationships to other objects at runtime.

2.3.1.5 Class Relationships

There are four primary types of relationships between classes of objects:

- The ***Is-a*** relationship is defined by inheritance.
- The ***Has-a*** relationship is the containment of one object within another as an internal data attribute.
- The ***Uses*** relationship is one in which an object of one class uses an object of another class. *Has-a* is a special case of *Uses*. A *Uses* relationship exists wherever a method of class A messages (calls) an object of class B, or if it creates, receives or returns objects of class B.
- ***Create-a*** relationship, is dynamic instantiation - where a method of class A creates objects of class B at runtime. *Create-a* is also a special case of *Uses*.

The earlier example in Figure 2.2 - `class Student extends Person` - is an example of an *Is-a* relationship between classes.

The following example - code building upon that in figure 2.2 - contains an example of both an *Is-a* and a *Has-a* relationship:

```
class Graduate extends Student
{
      public Graduate(String family, String first,
         Qualification firstQual)
      {
      super(family, first, 3);
         latestQualification = firstQual;
      }
      public void print()
      {
               System.out.println(firstName + " " + familyName + ", "
                 + latestQualification);
      }
      protected Qualification latestQualification;
}
```

Figure 2.8 - Overloading the print() method in class Graduate.

Class *Graduate* ***is-a*** *Student* and it ***has-a*** *Qualification,* where class Qualification is defined in the following code:

```
class Qualification
{
        public Qualification(int qualWhen, String qualWhere, String qualWhat)
        {       yearOfGraduation = qualWhen;
                placeOfQualification = qualWhere;
                nameOfQualification = qualWhat;
        }
        public String toString()
        {
                return (nameOfQualification + " " + placeOfQualification);
        }
        protected int  yearOfGraduation;
        protected String  placeOfQualification;
        protected String  nameOfQualification;
}
```

Figure 2.9 - A hierarchically unrelated class, Qualification.

An example Java application program which uses these classes *Person, Student, Graduate* and *Qualification,* follows:


```
import java.util.*;
public class TopGroup
{       public static void main (String[] args)
        {
                Qualification patties_qual = new Qualification(1982, "UCLA",
                                                                "BA(Rock)");
                Qualification kellys_qual  = new Qualification(1990, "Melbourne",
                                                                "BA(Ballad)");
                Student [] TopGroupMembers = new Student[3];
                TopGroupMembers[0] = new Student("Jackson", "Michael", 2);
                TopGroupMembers[1] = new Graduate("Smith", "Pattie", patties_qual);
                TopGroupMembers[2] = new Graduate("Kelly", "Paul", kellys_qual);
                int i;
                for (i =0; i < 3 ; i++) TopGroupMembers[i].print();
                TopGroupMembers[1].ChangeFamilyName("Code-Smith");
                for (int k =0; k < 3 ; k++) TopGroupMembers[k].print();
        }
}
```

Executing the program prints out the following lines of output to the screen:

Michael Jackson

Pattie Smith, BA(Rock) UCLA

Paul Kelly, BA(Ballad) Melbourne

Michael Jackson

Pattie Code-Smith, BA(Rock) UCLA

Paul Kelly, BA(Ballad) Melbourne

Note that the print() method cited in this code, dynamically binds to the appropriate print(), depending on which object is in the array element TopGroupMembers[k] it is dealing with. So in some cases it is printing out just first name and surname, while in other cases it is printing out first name, surname and qualifications. This is an example of polymorphism through late binding in the Java language.

2.3.2 Active Objects

An *Active Object* is instantiated from a class like any other object in an OOP language, but in addition, Active Objects are busy in themselves - they do not simply wait for their member methods to be called externally before setting about on some activity. Nonetheless, member methods can still be called from outside of the Active Object.

Systems with multiple Active Objects are most easily programmed in parallel programming languages such as Linda language derivatives [85], or languages which have a multithreading capability built into them such as Java. However in C++, those programs which internally enact an *event model* are also capable of running multiple Active Objects.

This property of *continuous execution* - particularly suited to 24x7 operation - is a feature that Active Objects share with *Software Agents*: where it is termed *pro-activeness*. Software Agents have additional facets which include: *autonomy*, *reactivity*, *social ability* and others, which are covered in Section 2.6. In a SlimWinX user interface, a system described further down in Sections 2.4.9 and 3.3, there are buttons with continuous animation playing on top of them (built into the system, not as simply animated .GIF files), which are effectively simple Active Objects. See Fig. 2.10 below : the button with a globe on top of it has a continuous revolving image of the world upon it, but the button remains receptive to a user mouse clicks, which will call methods in the Active Object. The SlimWinX system, though written in the C++ language, enacts its own *event-handling model* thereby allowing Active Objects to be programmed in a straight forward manner.

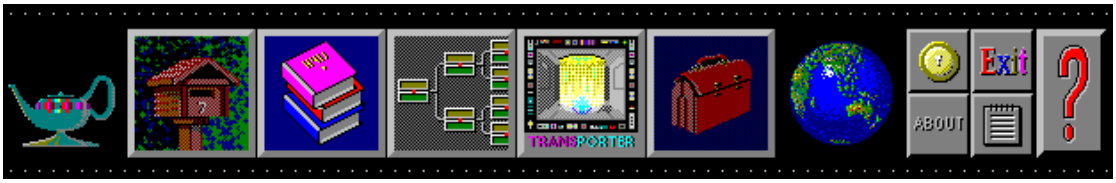

Figure 2.10: The 'globe' in this toolbar is a clickable Button which is also an Active Object.

2.4 Distributed Systems

The creation of computer networks in the 1970s spawned *distributed* applications and *concurrent* and *parallel* programming languages. It also spawned a level of software termed *middleware*, loosely encompassing the code between that implemented on the backend such as database systems and file systems on servers; and that code directly concerned with the client-side interface or client-side file system and peripherals. More lately, *middleware frameworks* have appeared upon which distributed

applications can be methodically implemented - notable *CORBA*, *DCOM* and in the last few years *Enterprise JavaBeans*.

In the mid 1990's the World Wide Web (WWW) delivered a universal hyper-linked information space, rendered largely in machine and human readable ASCII files (HTML). The WWW is rapidly evolving into a global knowledge web, also known as the semantic web using XML-based markup, peppered with *distributed systems*.

While the primary client interface remains the web-browser for the time being, the backend servers are being increasingly engineered as sophisticated distributed systems, which use an expanding number of markup languages and other mobile code, to facilitate communication and dynamic interactions with users and the client interface. A sizable part of this activity is focused on increasing both flexibility and variation at the client side of computing: a multitude of screen sizes and types from handheld computers and phones, to conventional desktop and set-top boxes engineered and marketed as Interactive Television (ITV) systems; stationary and mobile users; single user activities to computer supported co-operative work (CSCW) group activities.

This section presents a background to a few of the technologies and systems being used to enable distributed systems in the Internet arena. It also introduces the author's own *SlimWinX* and the *XSpace* systems, in the context of other more developed and more widely known systems including *JavaSpaces* from Sun Microsystems Inc, *TSpaces* from IBM, and the earlier *Tuplespace* model from the *Linda* language. Later, in Chapter 6, proposed extensions and modifications to XSpace and SlimWinX are outlined, as part of the implementation strategy for building a sophisticated agent called *Shadowboard*.

2.4.1 Java as Mobile Code

The widespread distribution and use of the two main web browsers - *Netscape Navigator* and *Microsoft Internet Explorer* - for using the World Wide Web, popularised two forms of mobile code: *JavaScript* embedded in client-side HTML, and *Java Applets* downloaded and referenced within HTML files. JavaScript is an *object-aware* scripting language, covered more extensively in Section 3.4. Java, as a

fully-featured OOP language, has continued to evolve into a mobile code system, able to move code and state across a network, and able to execute on any machine which is currently running a compatible *Java Virtual Machine* (JVM), inside or outside a web browser - depending on client-side settings.

Java compiles to *bytecode* - the native machine code of a virtual machine - which can then run inside a JVM within any operating system that can host a JVM - which includes most mainstream systems. See Figure 2.11 for an overview of the path that a Java applet takes, from source code in development to execution on a client machine.

Not surprisingly Java has become a popular language for the development of Mobile Agents - see Section 2.6.3.8.

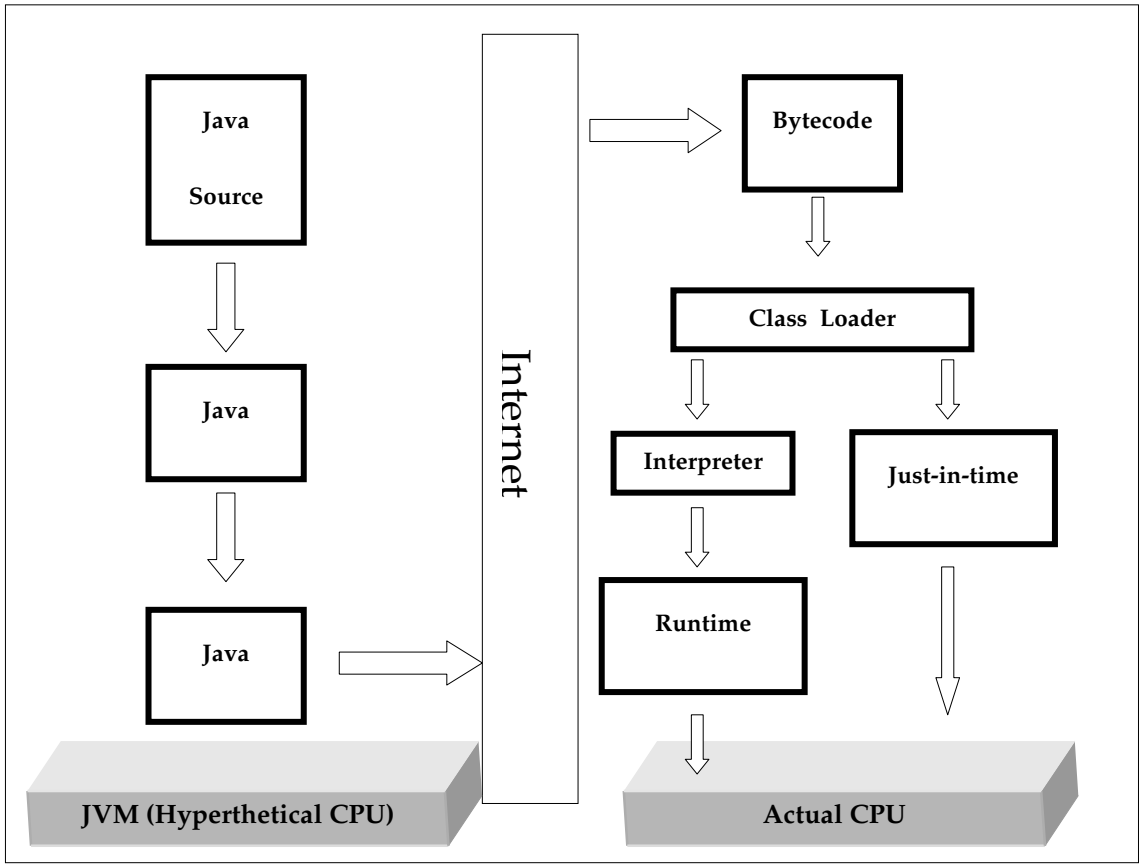

Figure 2.11 - Java Applet flow from source code to client machine execution.

2.4.2 Delegation Event Model in Java

Java is the first mainstream object-oriented language built from the ground up with distributed systems in mind. A consequential feature of the language is that it has a

built-in event model (since JDK V1.1) - the *Delegation Event Model* - which is well suited to distributed systems.

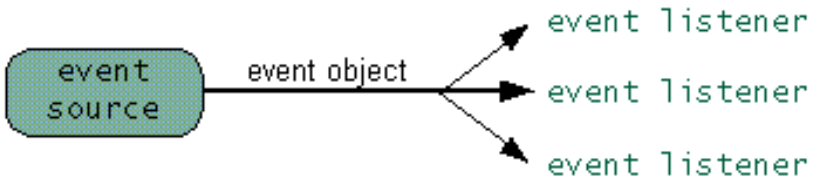

Figure 2.12 – Delegation Event Model in Java.

The Delegation Event Model consists of three conceptual entities (see Fig. 2.12 above):

- Event Types
- Event Source (which has listener registry and processing. E.g. a component Button)
- Event Listeners (which are the event handler methods).

Every element of communication between the GUI and the program logic is an *event.* An application class registers an interest in a particular *event type* (e.g. *ActionEvent*) emanating from a particular component (e.g. a *Button*), by asking that component (an *event source*) to add the application classes *event listener* (handler methods), to its list of interested listeners. Hence, a component such as a Button in Java has a built-in capability to maintain a linked-list registry of interested listeners. *Events* are objects which are reported only to the *registered listeners*.

The passing of *Event* objects does not have to involve a GUI at all. For example further down in Section 2.4.8, there is some coverage of the *notify* command in JavaSpaces, which lets an event received within the JavaSpace notify all manner of interested Java methods on distributed machines, in a *broadcast* sense.

2.4.3 Reflection and Garbage Collection in Java

Aside from encapsulation, inheritance and polymorphism, Java has other advanced features that bolster support for programming dynamic systems in distributed environments, including: *Reflection*; and built-in *garbage collection* - i.e. the non-deterministic destruction of objects.

A great advantage that comes from *single inheritance* in an OOP is that all object classes from whatever source ultimately inherit from a single parent. In the case of Java that base class is named the *Object* class. This single root to all other classes enables *reflection* (introspection) - the ability for a program to analyse the capabilities of classes at runtime. The *runtime type identification* (RTTI) of an object can be accessed by instantiating a reflective object of class *Class*, via the object one has an interest in. E.g. Assuming that it's the Student class of the earlier code in Fig 2.2, then:

```
Student s1;
Class myclass1 = e.getClass();
```

A Class object has special methods which give access to the inner details that all classes inherit.

`myclass1.getName();` would return the string "Student".

`Object o1 = myclass1.newInstance();` would dynamically create a new Student object.

```
String s2 = "Graduate";
Object o2 = Class.forName(s2).newInstance();
```
would create a new Graduate object, via a string.

For analysing classes at runtime, the Reflection mechanism in Java additionally allows a program to get:

- the data attributes (Fields) of a class;
- the methods of a class and the type and value of its parameters;
- the Constructor of a class.

In a distributed environment for both mobile and remote objects, the Reflective mechanism in Java is invaluable for dynamic programming. To compare that with C++, which is a multiple inheritance language, different classes of objects coded by different people over a distributed environment, are not guaranteed to have the same base class. If such multi-sourced C++ classes are to inter-cooperate, then the participants must subscribe to a scheme that does one of two things:

1. Agree upon some standard base class or classes.
2. Or else abide by some system outside of the C++ language itself.

CORBA is a major example of the second approach to distributed OOP, one that C++ programmers may enlist to enact a distributed processing system, which is covered briefly in the next section.

The lifetime of an object in Java is begun when it is instantiated, and deallocated through automatic garbage collection, non-deterministically. While instantiation in C++ is quite dynamic, deallocation is done implicitly by the programmer within the object's *delete* destructor. This contrast between the two languages, is again one in favour of Java in a distributed environment.

2.4.4 CORBA - Common Object Request Broker Architecture

CORBA was designed to allow components to discover each other and interoperate over an *object bus*. It also provides a number of bus-related services including: creating and removing objects; defining relationships between objects; putting objects into persistent stores.

CORBA has effectively taken the multiple inheritance aspect of C++ out of the language, and placed that capability into the encompassing operating system. The Object Request Broker (ORB) is the system software that does this work. In addition to enabling distributed programming, CORBA enables programs written in pre-OOP imperative language to have object-oriented abilities including: data encapsulation; polymorphism and multiple inheritance. As such CORBA allows both new and legacy applications to be built into new distributed applications.

CORBA defines *interfaces* to objects and services via IDL - the Interface Definition Language - in a manner that is independent of the operating system and the programming language. The CORBA IDL is purely declarative with language-neutral data types. IDL specified methods can be written in any language that provides CORBA bindings, which includes: C, C++, Java, COBOL, Ada and Smalltalk.

CORBA IDL is a subset of C++ but with additional keywords to support distributed concepts.

CORBA messages can be polymorphic: a message does not simply invoke an implicit remote procedure call; instead it invokes a method on a *target* object, hence the same message will have different effects, depending on which object receives the message.

The ORB has an *Interface Repository* - runtime metadata for describing all the functions that a server provides, in machine readable versions of the IDL-defined interfaces. I.e. The services that an ORB provides are themselves described in the same IDL that other user objects are described in. CORBA provides both static and dynamic interfaces to its services. ORB services include: a *Naming Service* to locate components by name; a *Persistence Service*; an *Event Service* to register an interest in specific events; a *Transaction Service* to provide two-phase commit coordination amongst recoverable objects; and many other services. Because the ORB services are themselves packaged with IDL-specified interfaces, any number of new services may be added to the ORB with minimal disruption.

Most aspects of CORBA, including a Naming Service and a Transaction Service, are available in a proprietary all-Java technology called *Enterprise Java Beans (EJB)* – with the exception of *easy assimilation* of legacy systems into the distributed object fabric. EJB itself is a *container object* with these numerous services built-in.

2.4.5 Blackboard Systems

The Blackboard model of problem solving arose from abstracting features of HEARSAY-II (an early speech recognition system) and HASP (an ocean surveillance system interpreting continuous sonar) between 1971 and 1976 [59]. The application of the blackboard model to two such diverse applications demonstrated that it was a

generic *problem solving model*. A problem solving model is a framework for organising knowledge together with a strategy for applying the knowledge to achieve solutions and partial solutions. All possible full and partial solutions to a problem represent the *solution space*.

The problem state data is held in a global database called the *Blackboard*. *Knowledge Sources* can be viewed as a group of specialists (domain dependent), each logically independent of the others yet apparently cooperating towards a solution, mediated by the Blackboard. See Figure 2.13 below.

Knowledge sources are capable of improving the current state of the solution and do so incrementally by either forward or backward reasoning methods. Over time, the solution space is organised into one or more hierarchies which are application dependent. Each level in the hierarchy represents partial solutions.

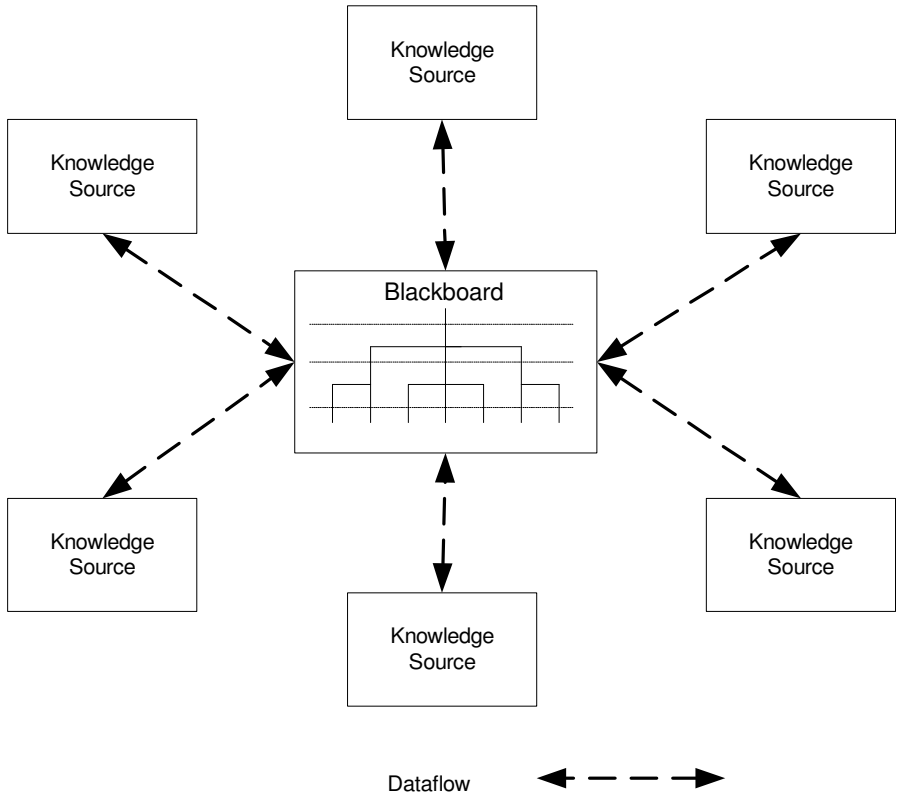

Figure 2.13 - Dataflow in a Blackboard System.

The Blackboard *model* does not have a concept of central control; instead, the *knowledge sources* change the shared Blackboard opportunistically in a dynamic

manner. A *model* is a conceptual entity, not a computational system. However a model usually does suggest behaviours that should be apparent in any computational system based upon it.

To tease out such behaviours in the Blackboard model, Penny Nii [59] gives an example that has multiple people solving a large jigsaw puzzle, the large pieces of which stick to a magnetic blackboard at the front of a room. Each person goes forward with a piece and places it into partial solutions of the complete puzzle, as he or she sees fit. The fellow puzzlers do not need to directly communicate since the progression towards the solution is mediated by the state of the puzzle on the blackboard, viewable by all. Each person is autonomous. The order in which they go up to the board is not predetermined nor controlled in the model.

Computational frameworks built upon the Blackboard model often introduce some form of control which monitors the progress of the solution space and applies some strategy regarding the choice of Knowledge Source interactions with the Blackboard. A control mechanism as such is a part of an engineered operational system for a given problem domain. Blackboard systems are well covered by Engelmore and Morgan [20].

2.4.6 Tuplespace Systems

A Tuplespace is a globally shared memory space that can be addressed associatively. It is a partial implementation of the Blackboard Model, one primarily designed for asynchronous communication rather than as a problem solving framework. The steady resolution of the Blackboard state data into hierarchies is not a feature of the Tuplespace model. A Tuplespace is simply a 'bag of tuples', and a tuple is simply a series or ordered vector of typed fields. The following table presents some tuples.

Tuple	**Template (an *antituple*)**	**A match?**
[11, 374, 640, "Globe", 12.5]	[11, 374, int, string, float]	Yes
[11, 374, 640, "Globe", 12]	[int, int, int, string, int]	Yes
[11, 374, 640, "Globe", 12]	[int, int, 640, "Globe2", int]	No
[30, 445, 383, "Globe", "Globe2",12]	[30, int, int, "Globe", string, int]	Yes

Table 2.1 - Four Tuples matched against Four Template(antituples) .

Tuples that share a given Tuplespace can be a vector of any length. In the table above, the first three tuples have five typed fields, while the fourth tuple has six typed fields. A Template (also known as an *antituple*) is used to address a tuple, associatively looking for a match rather than needing to address a tuple directly. Zero or more of the fields in the antituple can be replaced with *type* placeholders, called *formal fields*. The second column in Table 2.1 are templates (antituples). The third column indicates, whether or not the *template* in that row will match the *tuple* in that row.

The associative matching of the antituple against tuple, is very much like a *Query-By-Example* database query language, and the tuplespace is like a rudimentary database system.

The Tuplespace concept was proposed by Gelernter [28,29] as part of the Linda coordination language to enable the synchronisation of multiple processes via asynchronous communication. A process can create a tuple and place it in the Tuplespace where it remains persistent. Another process can retrieve the tuple, either a copy of it or remove it from the store destructively. The Linda language is a computational system that uses the Tuplespace to enact a parallel processing system. It provides the following minimal communication primitives to allow operations on the Tuplespace, from the host language:

Operator	Action
in	Takes a tuple from the tuplespace. Eg ***in(30, int i, int j, "Globe", string s1, int k);***
rd	Reads a tuple from the tuplespace, non-destructively. Eg ***rd(30, int l, int m, "Globe", string s2, int n);***
out	Writes a tuple to tuplespace. Eg ***out(11, 374, 640, "Globe", 12.0);***
eval	Evaluates expressions, including function calls, and places the results in a tuple. Eg ***eval(11, 300+74, "Globe", squareRoot(144.0));***
inp	A *blocking take*, meaning that the process will wait if the tuple is not yet in the Tuplespace to be taken.
rdp	A *blocking read*, meaning that the process will wait if the tuple is not yet in the Tuplespace to be read.

The simplicity of the Tuplespace concept enabled Linda to be easily implemented within numerous sequential languages (including Fortran, C, C++, Lisp, etc), enabling those language to be used for parallel processing.

The distinguishing features of the Tuplespace model are:

1. *Destination uncoupling:* The creator of a tuple requires no knowledge about the future use of that tuple, or its destination, so Tuplespace communication is fully anonymous.

2. *Space uncoupling:* By using an associative addressing scheme for tuples rather than a physical one, the Tuplespace is able to provide a globally shared data space to all processes, regardless of machine or platform boundaries.

3. *Time uncoupling:* Tuples have their own life span, independent of the processes that generated them, or any processes that may read them. This enables time-disjoint processes to communicate seamlessly.

4. *Type Flexible:* Lacking a schema a Tuplespace does not restrict the format of the tuples it stores or the type of data it contains.

Tuplespace systems are generally ranked above a pure message passing system since they include a data repository system and an associative addressing mechanism. As a data repository they rank below relational database systems as Tuplespaces do not usually include a transaction capability (enabling rollback), or a sophisticated query system.

2.4.7 TSpaces

The TSpaces system is an enhanced implementation of the Tuplespace concept, developed at the IBM Almaden Research Center. IBM Researchers were interested in being able to integrate small handheld and embedded devices, together with large servers and desktop machines, in a corporate-wide or global network. Wyckoff et al. [80] faced several pivotal problems that guided them towards the Tuplespace concept:

- Embedded processors and handheld devices are usually based on processors that are two or three generations behind the latest available chips, because they become comparatively inexpensive to make. This meant that the lowest common denominator in the network had both a small memory and small processor footprint.
- The system needed a flexible approach to data types, able to adapt to new data types.
- It had to provide a messaging service that supported anonymous communication - as machines may be turned off or may be busy.
- The system had to provide a flexible data repository for simple data, but a myriad of types of data.

They concluded that: *"Tuplespace is a good starting point for a distributed communication system because it provides communication, synchronisation and a simple data repository in one framework."*

TSpaces has the following operators and capabilities in common with most Tuplespace systems:

Operator	**Action**
take (in)	Read and remove a tuple from the Tuplespace.
read (rd)	Non-destructive read of a tuple in the Tuplespace
write (out)	Create and place a tuple in the Tuplespace
waittoread (rdp)	A *blocking read*, meaning that the process will wait if the tuple is not yet in the Tuplespace to be read.
waittotake (inp)	A *blocking take*, meaning that the process will wait if the tuple is not yet in the Tuplespace to be taken.

TSpaces has the following extra operators:

Operator	**Action**
scan	Reads multi-tuples if more than one matches the antituple, returning a vector of tuples.
consumingscan	The multi-tuple version of *take*.
rhonda	A pier-to-pier messaging operator, which has two arguments: one is a tuple, the second is a matching template. It will swap the tuple with another process executing a similar rhonda command which has a different tuple, but the same template in the second argument of rhonda.

In addition to these extra operators TSpaces allows the programmer to define new operators *dynamically* which are downloaded into the TSpaces server.

T Spaces is implemented in Java and it uses Java typing and the Java object model to give it variable typing. In addition *matching* is enhanced: TSpaces extends the Tuplespace notion of exact type equivalence, when matching tuple against template, to an object-oriented notion of equivalence: the type of a tuple in the TSpaces system must be a subclass of the type of the template. This allows the programmer to treat tuples as objects, addressable down at the level of task-specific subclasses of a generic class.

Unlike a conventional Tuplespace, TSpaces builds an index (in the database sense) on each *named* field in the tuple. By allowing *named fields* in a tuple, TSpaces has taken the concept of Tuplespace into the realm of database systems. TSpaces makes use of these indexes in a Query facility much like one finds in database systems. The Query facility enables clients to retrieve tuples based on queries which only refer to values in specific named fields. Eg. ("Student", 1220) will select all tuples *of any length* of vector, or order of fields, which contain a field named 'Student' with the integer value 1220. This is in contrast to the antituple matching query, which enforces the antituple and the tuple in a match to be of the same length, in terms of number of fields. The TSpaces query that uses a named field is termed an *Index query*. In TSpaces the match style of query available in a conventional tuplespace system is called a *Match query*. TSpaces also has an 'And' and an 'Or' operator, which allows the user to combine queries into complex query trees.

TSpaces combines the concept of a Tuplespace with the dynamic flexibility of Java, together with advanced database concepts, into a flexible network middleware solution for distributed systems across disparate hardware platforms. It is scalable in the sense that a particular installation of TSpaces may have one of three database implementations, behind the scenes:

- A simple array-based tuple manager, in Java.
- A memory resident data manager that enacts hash and T-Tree indexing algorithms, in Java.
- A wrapper and interface to IBM's enterprise relational DBMS product, DB2.

TSpaces evolves the Tuplespace concept in the direction of distributed database systems.

2.4.8 JavaSpaces

JavaSpaces is Sun Microsystems technology, designed to enable *distributed persistence* for objects and facilitate distributed algorithms in a distributed processing sense. In particular it provides services to Sun's Jini technology [41]. It draws from the Tuplespace model in being able to store state, but it has a significant addition in that it allows arbitrary *user objects* to be stored in the Tuplespace, which may then be forwarded, in a stored-and-forwarded message sense, or accessed via associative matching in a similar fashion to a Tuplespace. JavaSpaces is a Java-centric technology requiring a 100 percent pure Java Virtual Machine (JVM) on the client.

As with TSpaces one of the primary design goals of JavaSpaces was to provide a simple yet flexible network service for a diverse range of hardware platforms including embedded processors and handheld devices, which were likely to have small memory and processor footprints. As with TSpaces, the designers of JavaSpaces also had a primary concern with asynchronous communication and device synchronisation, and that combination of design goals led them to a Tuplespace inspired solution. But there was another primary design goal apparently not held by the TSpaces designers - *to allow the flow of data, in order to distribute algorithms* - that strongly influenced the resulting JavaSpaces implementation.

In place of the *tuple*, JavaSpaces has an *entity* - a *typed group* of object references (the fields), expressed as a class in Java. The fields within an entity must all be references to serialisable objects. So, on the Java platform the JavaSpaces *entity* is an *object of objects*. This is not just a semantic difference when compared with a tuple of typed fields: a JavaSpaces *entity* has behaviours that go with it, accessible via the methods of the class that enact the entity. JavaSpaces enables code mobility between processors in a distributed system. An entity retrieved from a JavaSpaces space is enacted through the JVM on the client machine.

JavaSpaces has the following operators and capabilities in common with most Tuplespace systems:

Operator	Action
take (in)	Read and remove an entity from the JavaSpaces space.
read (rd)	Non-destructive read of an entity in the JavaSpaces space
write (out)	Create and place an entity in the JavaSpaces space.

JavaSpaces has the following extra operator:

Operator	Action
notify	In the absence of *blocking read* (Eg waittoread) and *blocking take* operators in JavaSpaces, it makes use of the *distributed event model* built into Java, to enable the JavaSpaces server to notify multiple interested clients, when a particular tuple (entity) arrives in the space. The registered clients which have an interest expressed their interest earlier via the *notify* operator.

JavaSpaces also adopts the associative matching functionality of a Tuplespace, using a template (antituple) to retrieve entities from a space via value-matching lookup. A *formal field* in an antituple in Tuplespace terminology is termed a *wildcard* field in the template in JavaSpaces.

JavaSpaces supports a simple transaction mechanism allowing for both multiple operations and access across multiple spaces to complete or to rollback if the transaction cannot be completed in full. An entry that is written (via *write*), is not visible outside its transaction until after the transaction is complete. The situation for *take* is similar to that for the *read* operation. A *read* may match any entity written during the same transaction, or any entity in the entire space.

JavaSpaces evolves the Tuplespace concept in the direction of distributed object persistence. However, where an object database system stores 'the' object in terms of

application operations upon it, with entities held in JavaSpaces, one works upon copies of objects.

Where IBM researchers added *named* fields to TSpaces to give it fuller database features, Sun researchers replaced the tuple with a *typed object* (an *entity*) that has behaviours via the onboard methods of the object. And *within* the entity they have objects whereas the TSpaces tuple may have *typed* fields, *objects* or *named* fields. In template matching, where TSpaces distinguishes between classes and subclasses of objects in a tuple (not matching the two, even though they may have the same number and order of fields), JavaSpaces will allow a match between types and subtypes (one in the tuple, the other in the template), in the polymorphic manner of the OOP paradigm.

An application built upon JavaSpaces that uses the space as more than a state store, is best analysed and designed as a *flow of objects*, into and out of one or more JavaSpaces.

2.4.9 XSpaces and SlimWinX

XSpaces is an integral part of *SlimWinX,* both of which were developed by the author in a period between 1992 and 1994 [33], implemented in the C++ language. XSpaces has been named retrospectively - it was previously called MetaTree. SlimWinX - covered more extensively in Section 3.3 - was developed as a flexible *dynamic object system* with a GUI interface for DOS-based systems, for the development of simulation and role playing games. The first game concept developed using SlimWinX was called *Branch-Out* - a multicultural adventure game [34]. The game involved the player taking the role of a sleuth who traveled around the world, able to *beam down* to numerous locations, in the manner of role playing games like the well known Carmen San Diego series. In order for the system to incorporate large amounts of graphics, text and a multicultural database of almanac proportions, all within a 640K RAM system (DOS), a dynamic system of object swapping to disk was developed by the author. It automatically removed the least-used currently-not-needed objects from memory, and dynamically instantiated the objects again as needed, via a permanently available tuplespace-like system named *XSpaces*.

Though not influenced by the Tuplespace and Linda concepts during design and development, XSpaces has significant similarities to the Tuplespace concept in that tuples can be placed and retrieved dynamically from a state space within XSpaces. I.e. SlimWinX uses the *time uncoupling* feature of the tuplespace model to overcome the severe memory constraints on the targeted platform. However, SlimWinX does not currently use a network resident version of XSpace to achieve the memory savings. Instead XSpaces runs on the client machine and uses the local hard disk as dynamic storage at the tuple/object level - each tuple becomes an object within SlimWinX, as happens in JavaSpaces. However, given that SlimWinX incorporates its own object-oriented event model, network events could be easily added to the primary *get-event* / *dispatch-event* loop, which would then enable XSpaces to be located on processors other than those running the SlimWinX system.

A significant divergence from the Tuplespace concept is that the state space in XSpaces is held in a unified *structure* (hierarchical), which represents relationships between the objects - each tuple becomes an object within the accessing application. A small sub-hierarchy from a much larger XSpace is represented in the figure below:

```
PANEL,PDress_Keypad1,1,0,11,11,0,0,640,480,4,0
{
        PANE,pad83_0,1,0,32,47,10,9,KOORIM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_1,1,0,32,48,114,9,JEWISHM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_2,1,0,32,49,218,9,DUTCHF,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_3,1,0,32,50,322,9,BRITM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_4,1,0,32,51,426,9,SCOTF,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_5,1,0,32,52,530,9,FRENCHF,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_6,1,0,32,53,10,162,GERMANM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_7,1,0,32,54,114,162,VIETNAMF,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_8,1,0,32,55,218,162,SCOTM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_9,1,0,32,56,322,162,GREEK1,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_10,1,0,32,57,426,162,SIHKM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_11,1,0,32,58,530,162,CHINESEM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_12,1,0,32,59,10,315,GREEK2,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_13,1,0,32,60,114,315,MUSLEMF,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_14,1,0,32,61,218,315,ITALIANM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_15,1,0,32,62,322,315,INDIANF,(null),4700,271,0,0,116,100,34,4,6
```

```
        PANE,pad83_16,1,0,32,63,426,315,WESTERNM,(null),4700,271,0,0,116,100,34,4,6
        PANE,pad83_17,1,0,30,89,554,394,36,36,2,ESC32,ESC321,5000,270,0
        PANE,pad83_18,1,0,30,90,554,430,36,36,2,NOTE32,NOTE321,4F00,150,0
        PANE,pad83_19,1,0,30,91,591,394,40,72,4,HELP,HELP1,7800,40302,0
}
```

Figure 2.14 - Tuples that represent buttons in the keypad of a GUI application program.

XSpaces names each tuple just as JavaSpaces names each *entity*. Eg. *PDress_Keypad1* and *pad83_19* are the names of the first and the last tuple in the figure above. The full hierarchy of tuples for the interface and animations in the Branch-Out application are presented in Appendix A. The arrangement of the hierarchy within XSpaces is *dynamic* - the content and number of hierarchies and sub-hierarchies of the XSpace changes over time, in synchronisation with the changing relationships between objects within connecting applications. In this sense XSpaces is like the *Blackboard model*, with an evolving solution space in which sub-trees represent partial solutions in the complete space.

The XSpace is like both a single tuplespace, in which all tuples can be accessed equally (the hierarchy is ignored), and a whole series of related sub-tuplespaces - the siblings grouped together under a single parent tuple. Eg. In Figure 2.14 above, all the tuples, with the exception of the first one (the parent of the rest), form a sub-tuplespace which can be accessed as if it were a standalone tuplespace. Note that the underlying structure in the C++ implementation of XSpaces is both a tree structure and a double-linked list structure, with the functionalities usually associated with both sort of structures.

In addition to the tuplespace nature and Blackboard nature of XSpaces, it also has a persistent database behind certain *named* fields, in a similar manner to TSpaces. For example the tuples in the above figure with values in fields such as 'HELP' and 'NOTE32' are the names of graphics, which are held in an indexed database file, termed a GTO file (Graphics, Text and Objects). These files carry a dot extension of .GTO. Large text fields, such as Help messages, are also stored in GTO databases, with only the message-ID number appearing in the tuple. For these *complex* data

types a key is saved in the tuple. As with the presence of sub-tuplespaces within the branches of the total hierarchical tuplespace, XSpaces also may have these database files associated at the two levels: a globally accessed GTO file, and another at each group level within the tree structure.

XSpaces has the following operators and capabilities in common with most Tuplespace systems, but which function more like *read* and *write* in JavaSpaces, but not quite:

Operator	**Action**
get_object (rd/read)	Non-destructive read of an object in the XSpaces space
create_object (out/write)	Create and place an object in the XSpaces space.
cross_match (match)	An associative matching function.
(in/take)	*Read* and *remove* an object is currently only effected via a *drag-and-drop* operation in the GUI of the SlimWinX system.

Each tuple read from XSpaces (via *get_object*) becomes an object in the SlimWinX application code with the accompanying methods, much as the *read* in JavaSpaces creates an *entity* (which is really a Java object) with the associated functional algorithms.

XSpaces is also similar to the JavaSpaces system in a *naming service sense*, in that they both use an *object type* system in place of a *naming service* (which generally use name-value pairs). Where JavaSpaces simply uses the built-in Java object typing system, XSpaces relies on the *type_id* of objects being known to the client in advance. I.e. SlimWinX currently has more than 30 types of objects. [Note: With regard to the typing of objects retrieved from a tuplespace-like system, those written in Java have the advantage of the *introspection* functionality built into the basic Java object that in turn comes of single inheritance.]

SlimWinX uses the XSpace hierarchy of objects as a hierarchy independent of the one built-up within the application program in the C++ language. Rather than trying to track the actual C++ objects in and out of a persistent object system, SlimWinX tracks them in and out of the XSpace as the application creates and destroys C++ objects as it needs them. Within SlimWinX there is a mapping model between the two hierarchical representations of objects. It is a cross-referenced *individual-society model*, as follows:

Individual	**Society of objects**
Instantiated C++ Object	XSpace Hierarchy
*ego *(as seen from societal perspective)*	*self *(as seen from individual perspective)*

Figure 2.15 - The Individual-Society Model within SlimWinX.

The instantiated object in C++ is the *individual* and it has a reference called *ego* to the placeholder in the society of objects to which it belongs. The structure representing the *society of objects* is the tree structure from the XSpace, and each entity in the current society has a reference called *self* to the individual object instantiated in the application. If the individual object is deleted within C++, the *self* pointer gets set to *null*. The individual then lays dormant in the XSpace, represented by a lightweight tuple of information. If the application needs to enliven the individual object again, it makes a call upon the XSpace with a create_object call. At this point the entity within the society hierarchy has its *self* pointer set to the new objects *this* pointer, and the new objects *ego* pointer gets set to the entity placeholder in society tree. This cross-referenced *individual-society model* is a simple but powerful one in terms of what it can be harnessed to do.

XSpaces evolves the Tuplespace concept towards distributed hierarchical structures, in the spirit of the original Blackboard model with regard to constructing partial solutions, but ones which are constructed interactively by human users at the GUI. The dynamic nature in which the XSpaces tree can be modified makes it already highly suitable for *active objects* and 24x7 operation. Much further down, in Chapter 6

we cover the extensions necessary to XSpaces to accommodate the Shadowboard architecture, by enabling the state space to be computationally modified by the Digital Self.

2.5 Logic Programming

In the dynamic environment of a workstation connected 24x7 to a network of largely unknown computing entities awash with unknown components, objects, datasets and information spaces, a lot of the certainty that a programmer previously had on a single machine and within a fixed network, is gone. Such an incompleteness of knowledge at the time of programming, together with the high level of unforeseen situations that may arise, leaves the *top-down* design methodologies that go with object-oriented languages in a position of considerably less worth than they are on standalone machines and known intranets. For this reason, particularly with respect to *decision making* on the user's behalf - both in goal selection and plan selection - aspects of *logic programming* and *constraint programming* are introduced and employed within this thesis.

2.5.1 Horn Clauses, Predicates and Logic Programs

Rules, Facts, Queries and Goals

A logic program is the direct use of logic as programming language constructs, by using a set of logical clauses known as *Horn rules*, also called *Horn clauses*. A Horn rule is in the form:

$$A \leftarrow B_1 \,/\backslash\, B_2 \,/\backslash\, \ldots \,/\backslash\, B_n$$

and can also be written as:

$$A \leftarrow B_1, B_2, \ldots B_n$$

Horn clauses use symbols to represent the logic that we regularly find within everyday human language and thoughts. For example the propositions within a group of related English sentences:

I have enough time to go on a journey. (B_1)

I have enough money to go on a journey. (B_2)

I have the desire to go on a journey. (B_3)

I will go on a journey. (A)

And where these symbols have the following meaning:

implies.

/\ (,) conjunction (logical *and*) of propositions.

Then the horn rule above implies that: *I will go on a journey if I have enough time to go on a journey, and I have enough money to go on a journey, and I have the desire to go on a journey*. In its short form in English: *I will go on a journey if I have enough time, enough money and the desire.*

This is *reading the rule* from left to right, and is called a *procedural reading* of the rule. If one reads the rule from right to left as follows – *I have enough time, I have enough money, I have the desire to go on a journey, so I will go on a journey* – it is called a *declarative reading* of the rule. We understand and use that sort of logic in human language everyday in negotiating our way in the world.

By replacing the English with symbols that make up Horn clauses, we can arrive at a logic program in a very direct manner. Horn clauses represent logic in a specific and convenient form: there is only one proposition to the left of the *implies* symbol – which is called the *head* of the rule - and zero or more to the right – which is called the body.

A *fact* is a special case of a horn clause, one that has no propositions to the right. Eg.

I will go on a journey.

is a *fact*.

By converting logical English sentences into Horn clauses in this way, we arrive at a logic program. I.e. A logic program is a finite set of such statements. Putting the above sentences into a complete program looks like this:

```
B1.      (I have enough time to go on a journey)
```


B_2. (I have enough money to go on a journey)

B_3. (I have the desire to go on a journey)

$A \leftarrow B_1, B_2, B_3$. (I will go on a journey if I have enough time, enough money and the desire)

In this case, three *facts* followed by a *rule*. The order of the statements in the logic program is unimportant with regard to the information that can be derived from it.

In English a statement like *"I will go on a journey"* can be either *telling* or *asking*, depending on the context of the sentence. I.e. It can also be posed as a *query*: *I will go on a journey?* But in this case it is better expressed in English in a slightly different form of the sentence: *Will I go on a journey?*

While English often uses the positioning of pronouns and other subtle changes to emphasis the statement as *either* a *fact* or a *query*, in logic programs they are syntactically the same. However, they can be explicitly indicated as one or the other by ending the statement with either a *full stop* or a *question mark*. The *telling* aspect of the statement (a fact) is analogous to the *declarative* reading of a rule, while the *asking* aspect of the statement (a query) is analogous to the *procedural* reading of a rule.

Referring back to the four-line logic program above, we may pose the query *Will I go on a journey?* simply as:

A?

To which it would answer: 'Yes' or True.

But we would assert it as a fact like this:

A.

Facts, *queries* and *rules* as described above, make up the three basic statements of a logic program.

Also a *goal* can be a *fact* or a *query*. Program execution is initiated by a *goal*, which the system then reduces to a solution(s) if it can compute one or more, or halt when no solution is found. A basic step within that computation is a *reduction* – where the replaced goal is *reduced* and new goals are *derived*.

There is a single data structure used in logic programs called a *term*. Constants and variables are terms, and there are compound terms for example a *predicate* – detailed below. Terms are represented internally as a tree structure.

Predicates

Predicate logic is similar to the above described *propositional logic*. However, in predicate logic generalisations are made about whole groups of statements, and so predicate logic programs can become more interesting and multi-purpose.

Continuing with the *journey* theme, we could have a *predicate* called *destination* that represents a relationship between *place*, *cost* to get there, and *duration* - measured in consecutive days at the place.

```
destination(Place, Cost, Duration).
```

A *predicate* is a collection of clauses each of whose head has the same name (*functor*) and number of parameters (the *arity* of the predicate), for which the relationship holds. Here are some clauses in the collection represented by the predicate *destination*:

```
destination(brisbane, 800, 5).
destination(sydney, 500, 3).
destination(perth, 1000, 6).
```

A term starting with a capital letter, such as *Place*, is a variable, while one that does not, like *brisbane*, is a constant - an *individual*. Terms that do not have any variables are called *ground* and the others are called *nonground*. Within the computation in an executing logic program, *substitutions* are made between *variable* and *term* pairs, written like this: {Place=brisbane}. Returned solutions from a finishing logic program execution are often presented in these same curly brackets.

Unification

The computational heart of a logic language is based on *unification* of terms to achieve automated deduction. In any one pass of a logic program (recursion is a prominent feature of most logic languages), any given variable is assigned to just one individual via substitution. A *unifier* of two terms is a substitution that makes the two terms identical. In a logic language such as Prolog all data structures, such as

lists, records and trees, are represented internally as trees. Unification in Prolog then, is based on a tree unification algorithm. For a detailed reading on logic programming and the Prolog language in particular, see Sterling and Shapiro [70].

2.5.2 Constraint Logic Programming

Constraint logic programming evolved from logic programming. The essential modification happens in the unification process. Instead of unifying with an *equality* in a substitution, a test is done for constraint satisfaction (eg. X *greater-than* Y).

Continuing with the example theme of *journey* from Section 2.5.1, the following example logic program (Fig. 2.16 below) uses the earlier described predicate *destination.* It also uses some greater-than-or-equal-to *constraints* to build a useful general purpose *constraint logic program*.

```
journey(Place, Duration, Cost, Desire, Person)
                        ← destination(Place, Cost, Duration),
                           money(Person) >= Cost,
                           available_time(Person) >= Duration,
                           desire_to_travel(Person, Desire),
                           Desire = yes.
destination(brisbane, 800, 5).
destination(sydney, 500, 3).
destination(perth, 1000, 6).
destination(canberra, 300, 2).
destination(hobart, 350, 3).
destination(adelaide, 600, 4).
money(john, 1100).
money(jed, 700).
money(jill,400).
money(june, 800).
available_time(john, 2).
available_time(jed, 6).
available_time(jill, 4).
available_time(june, 5).
desire_to_travel(X, yes).
```

Figure 2.16 – Journey - a Constraint Logic Program.

Note: The last line in the program shows a single line that sets every individual's *desire_to_travel* to *yes* by placing a variable in a fact.

Constraint logic programs are little more than a specification of the relationships between objects – something they share with logic programs. It is then up to the language implementation to maintain those relationships. This gives them extraordinary flexibility in answering a range of goals/problems, when compared to a similarly sized program in an imperative language such as C or Pascal. For example, here are some queries and how they would be put to the above program called Journey:

Problem	***Query put to the Journey program***
Which people could go to Brisbane?	`journey(brisbane, Duration, Cost, Desire, Person)?`
What places could Jill travel to?	`journey(Place, Duration, Cost, Desire, jill)?`
What does it cost to go to Canberra?	`destination(canberra, Cost, Duration)?`
How much time does John have to travel?	`available_time(john, Time)?`

The Journey program may have been written specifically to answer just the first type of query, but it answers the other types for free. An equivalent imperative program would have to be programmed to handle each of those types of queries, from design time forward. For a detailed reading on constraint logic programming see Marriott and Stuckey [50].

2.6 The Agent Oriented Paradigm

There are currently many wide-ranging definitions of what constitutes an agent. The following is the definition used in this thesis whenever a specific definition is not given:

> *An agent is an entity situated in a changing environment, that continuously receives perceptual input and reacts by taking actions. Its choices of action are rational and are autonomously determined by calling upon internal mental state, consisting of beliefs, plans, goals and intentions.*

In short, agents are: persistent, autonomous, reactive, pro-active, flexible and social (capable of interacting).

How do software agents differ from other programs? Where a traditional program has *inputs* and *outputs*, an agent has *perceptions* and takes *actions*. Where the program holds facts, the agent holds both facts and beliefs and is therefore capable of execution with incomplete knowledge. Where the program usually stops or concludes when inputs cease, the agent runs continuously in an open system, autonomously, not just reacting to inputs. An agent has an internal model of its world (the environment it is situated in), and it expects that the actions it takes will change the environment in a manner that matches its own goals. It is not only the environment that is dynamic, the agent's own internal state is open to change, including its beliefs, goals, intentions, and how to achieve them. Figure 2.17 below represents the BDI Agent architecture introduced more fully down in Section 2.6.2.2

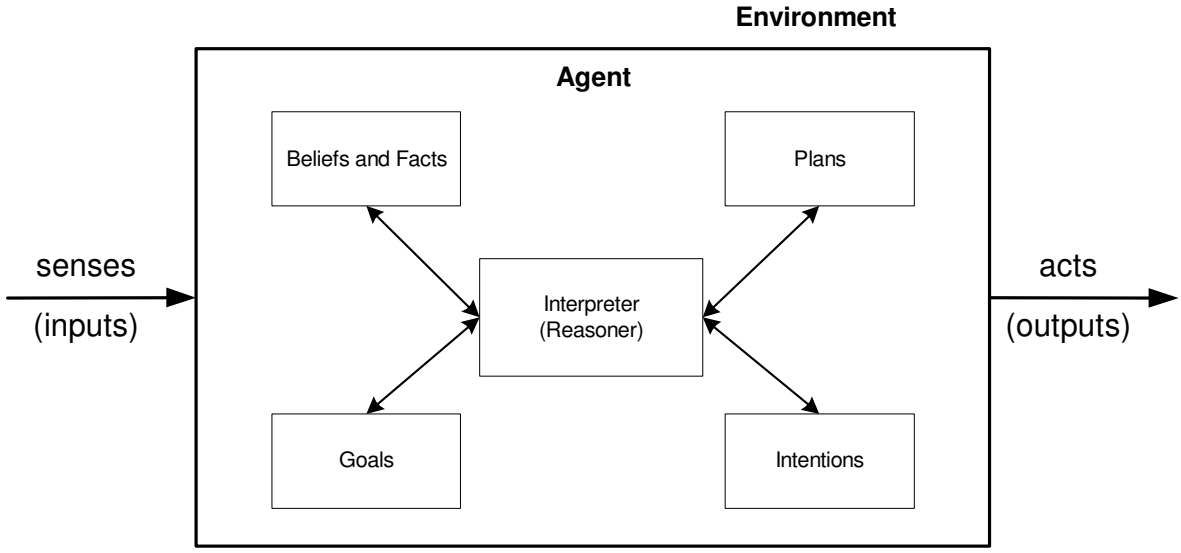

Figure 2.17 – BDI Agent Architecture, with Plans

The agent-oriented paradigm of programming has come to the fore in recent years for a number of reasons, including:

- It helps in modelling and building complex systems.

- An agent's ability to operate in dynamic and open environments. In particular: the Internet - the primary 24x7 environment; and also the environments in which robots operate – subsets of the physical world.

- An agent's autonomous abilities - particularly with respect to robots;

- An agent's interacting abilities - particularly with respect to software-based agents.

There have been many evolving streams of agent research. The BDI model represented in Fig. 2.17 is one of the higher order models. Other such high order models and the steps along the evolutionary paths in reaching them, are both interesting and instructive.

For the purposes of this introduction the numerous approaches to agents can be grouped within three influential categories of agents: *reactive agents*; *deliberative agents*; and *interacting agents*. There are also *hybrids* of those three types of agents. Another way of looking at these three categories of agents is to think about them with respect to three levels of awareness: *situated-awareness*, *self-awareness* and *social awareness*. Reactive agents have situated-awareness only. Deliberative agents generally have both situated-awareness and some self-awareness, while sophisticated interacting agents, such as BDI agents within a Multi-Agent System (MAS), have all three levels of awareness to some degree.

The following sections introduce various agent concepts within these three categories of agent.

2.6.1 Reactive Agents - Situated Aware

A reactive agent is situated in an environment that it senses and acts upon in an autonomous way. *Reactivity* is the sensing of changes in an agent's environment and then acting in a timely manner. A purely reactive agent does not need or have an internal representation of its environment – as such it does not have a world view. Such agents take in immediate local sensory information and they make decisions on relatively simple *situation-action* rules, which are often held in predicate logic.

The behaviour of reactive agents is effectively hardwired, which limits their flexibility. However, since they hold very little state, their advantage is that they can act more immediately than can deliberative agents. A robot that is programmed to simply change direction when it bumps into an object is operating in a reactive agent sense. It is this *situation-awareness* that gives reactive agents the ability to behave in a seemingly intelligent manner, without the need to deliberate significantly.

Firby [27] considers the reactive agent approach attractive in the making of a 'simple' synthetic (robot) vacuum cleaner, because the task contains a good deal of uncertainty, unpredictability and lack of knowledge. He envisages simple reactive strategies requiring only immediate local sensory data for the task. Firby goes on to describe the need for more sophisticated synthetic vacuum cleaner agents, which we will come back to further down in Section 2.6.4.3 on Hybrid Agents.

2.6.2 Deliberative Agents - Situated Aware and Self Aware

Deliberative Agents, also known as Intentional Agents, have a *world view* which considerably improves their degree of situation-awareness – i.e. they think global rather than think local. From an implementation point of view, holding a world view also alleviates the need for them to hold facts in predicate logic; facts may instead be held within a database or a linked-list. Changes to the agent's beliefs and known facts may then be accomplished by operators (eg. addList(aFact), deleteList(aFact)). Deliberative Agents have beliefs and goals, and they are also considered to be *rational agents*.

2.6.2.1 Rational Agents

Bratman et al [7] define a rational agent as one that *behaves* rationally in the sense that its behaviour is the production of actions that further the goals of the agent, based upon its concept of the world – its *world view*. Rational agents are committed to do what they plan to do. Additionally, a rational agent's adopted *plans* in any given situation should be consistent with each other and with the agent's *beliefs*. The adopted plans are drawn from a plan library, one constructed prior to agent execution. Most types of deliberative agents exhibit such rational agency.

Deliberation is done through the selection of a *goal*, selection of a *plan* that will be used to form an *intention*, selection of an intention, and then the execution (action) of the selected intention. The deliberation process produces intentions. All of these decisions are based on the beliefs of the agent. The process of selecting the plan is known as *means-end reasoning* – an ability to compute a sequence of actions that will achieve a particular goal.

2.6.2.2 BDI Agents and Plans

The BDI Agent architecture is one of the evolved forms of a deliberative agent. BDI calls upon the *mentalistic* notions of *Beliefs*, *Desires* and *Intentions* as abstractions to encapsulate the hidden complexity of the inner functioning of an *individual* agent. In an overview of Agents as *rational systems*, Wooldridge and Jennings [82] put in context the use of such *mentalistic* notions drawn from folk psychology, as legitimate *abstractions* for use within computer science:

> *"Much of computer science is concerned with good abstraction mechanisms, since these allow system developers to manage complexity with greater ease: witness procedural abstraction, abstract data types, and most recently objects. So why not use the intentional stance as an abstracting tool in computer science."*

As such, a BDI agent is an operational computation of a folk psychology. This allows the agent to exhibit behaviour consistent with a reasonable degree of *self awareness*. The object-oriented programmer may think of BDI Agents as *objects with attitude*. Intentional systems use these terms of attitude (Beliefs, Desires and Intentions) to select their own behaviour or to predict that of others in a competitive or collaborative situation.

Beliefs primarily represent the current state of the environment as perceived by the agent – although beliefs may include facts (static beliefs), they may also include a limited and distorted view of the actual situation. A BDI agent is able to modify its beliefs over time, as necessary to rationally portray an inner world view that matches the *percepts* it receives from the outer world. Although this notion of *belief* is significantly different from the notion of *knowledge* and may seem more limiting, it is this use of changeable facts (beliefs) over static facts alone, which allow agents to

operate in the absence of complete knowledge of a situation. An agent's beliefs also include belief in being able to do certain actions – a degree of *self belief*.

Desire is an unconstrained attitude. An agent's desires may conflict with what it believes to be possible, or with other desires and intentions. A *goal* is a *desire* that an agent adopts as a possible intention. To be rational, the BDI agent should believe goals to be achievable and multiple goals should be consistent. Operationally *goals* are little more than *rational desires*, however the term desire and its context gives BDI agency a better fit with psychology models of mind, albeit Folk Psychology, than most other agent architectures. Note: In Chapter 4, where we look at a number of psychologies, we make a connection between the BDI model and the Freudian model of the mind from Analytical Psychology, and there we make the claim that the Freudian model is the root source of the BDI.

An ***intention*** involves a commitment to act at some point in the near future, in order to achieve a particular goal. Operationally, intentions may be queued in a priority-based stack. Therefore a *goal* is a candidate for movement into the intention stack. Intentions can cause adoption of further sub-goals and hence further deliberation.

Plans*:* To take a BDI architecture and build it into a computational system requires the use of plans – precompiled recipes for achieving certain goals, each with a set of matching starting conditions. Plan selection is also known as *bounded rational choice*. Plans usually contain the distilled wisdom of an expert in a given field (human expertise). Typically, a *library of plans* is held by the agent. It is more accurately described as a *library of partial-plans*, as the exact combination of such plans that an agent will string together to satisfy a given goal in a given environment is a dynamic process. These partial-plans are not only incomplete, they can also contain sub-goals in addition to actions. An *intention* in an operational BDI agent is a commitment to a particular plan.

The following simple BDI interpreter algorithm was defined by Rao and Georgeff [60]:

```
BDI-interpreter
initialize-state();
do
      options := options-generator (event-queue, B, G, I);
      selected-options := deliberate (options, B, G, I);
      update-intentions (selected-options, I);
      execute(I);
      get-new-external-events();
      drop-successful-attitudes (B, G, I);
      drop-impossible-attitudes(B, G, I)
until quit
```

A BDI agent will execute the adopted plan until further deliberation is required. Further deliberation may be required in the following circumstances: the goal has been achieved; achieving the goal has become impossible; new events have occurred which require further deliberation.

2.6.2.3 Agent-0

In Shoham's [64] early proposal for a paradigm of Agent Oriented Programming (AOP), he identifies the mentalistic notions of *beliefs*, *capabilities*, *choices (decisions)* and *commitments*. And he recognises that various constraints are put upon the mental state of the agent, subjugating its autonomy. His approach to Agent Oriented Programming is not so much the specification of a well-rounded agent architecture, as a computational framework that Shoham could easily build, which he also outlines in the paper as Agent-0. He expresses a computation as consisting of *"agents informing, requesting, offering, accepting, rejecting, competing and assisting one another"*. Message passing between agents is a primary part of the Agent-0 framework.

Shoham acknowledges the body of research in AI leading to the BDI agent architecture. His *beliefs* can be equated to beliefs in BDI – beliefs about the agent's world, about itself and about other agent's mental state. However, his *capabilities* refer to the agent's self-knowledge (and knowledge about other agents) with respect to what it *can do* and *cannot do*, singled out from the belief modality, as being of particular interest to implementers of a computational framework. In BDI agents

such *capabilities* are referenced or called within plans, and as such are not given much prominence in the architectural overview.

Shoham further reduces the complexity of his framework by folding *choices* and *decisions* into *commitments*, leaving him just two modalities to implement in Agent-0 – *beliefs* and *commitment*. This renders his agents free of motivation (desire), other than what they get via *commitment* to others. He sees *commitment* as equating to obligation, and he sees *decision* as an obligation (commitment) to oneself.

Shoham muddies the modal waters of the earlier BDI architecture in his effort to get an expedient system defined, up and running. What is of interest here, apart from his call for an AOP paradigm, is that from the beginning Shoham's Agent-0 was envisaged as a multi-agent system (MAS), hence his added interest in an agent's *capability* and his early recognition that agents have constrained autonomy.

2.6.3 Interactive Agents - Situated, Self and Socially Aware

In his keynote talk at ICMAS-2000, within the context of multi-agent systems, Tom Malone [49] expressed the view that: *"People can be agents, but agents can't be people"*. He was discussing complex interactions of human-in-the-loop multi-agent systems and, in particular, where the responsibility for decisions taken lay. Increasingly one has systems in which people and agents are intermixed. Agents are participating in societies of users at a growing rate and with increasing sophistication in their agency. It is from this perspective – the growing mixture of people and intelligent software agents in processes and systems – that we have grouped together the following types of agents under the term *Interactive Agents*. Whether the *interaction* is between agents and humans, or agents with other agents, or agents mediating human-to-human communication, in each situation these agents are displaying some degree of *social awareness*.

2.6.3.1 Adaptive Agents, Interface Agents

Later, in Section 3.1 when discussing client-side objects and metaphors, we discuss a paper by Lewis [46] in which he considered aspects of human-agent interaction. His

definition of a software agent, a definition considerably simpler than what has been covered here in Section 2.6, is: "*an agent is a program that automates some stage of the human-information-processing cycle, one intended to automate repetitive, under-specified or complex processes*." If one takes an automation process as described by Lewis and adds some built-in initiative, some autonomy, and some ability to learn from the user, then this enhanced description of an agent is more generally known an *Adaptive Agent.*

Adaptive agents that deal with the actual interface between a human user and system or application programs are usually called *Interface Agents*. For example, an agent that adjusts aspects of the software interface and selectable options, to better facilitate a user's unfolding regular usage pattern, is an Interface Agent.

Such agents learn to anticipate user actions (automation of action) usually by learning the categorisation scheme used by the human user, or his/her general usage pattern observed over a period of time. These agents generally employ a machine learning scheme such as a Concept Learning algorithm to achieve this. Adaptive agents and Interface agents were preceded by a body of research in user interaction, going under subject titles such as: *adaptive systems*, *adaptive interfaces*, *intelligent user interfaces* and *intelligent tutoring systems*.

2.6.3.2 Personal Assistant Agents

More sophisticated Interface Agents were developed that are better thought of as highly personalised digital assistants, or *Personal Assistant Agents*. Personalisation refers to the ability of such an agent to adapt its behaviour to an individual's interactive style, information needs, location, personal workflow and other user preferences in specific repeated situations or near-fit situations. An early example of such agents was developed to *filter* email and Usenet news items, see Maes and Kozierok [48]. These types of agents are sometimes termed *filtering agents*. Maes and Kozierok also developed a meeting scheduler agent which automatically responded to outside requests for meetings on the user's behalf, based on prior user response patterns.

Personal assistant agents are commonly devised to do specific tasks on the user's behalf, including *filtering* of information, *notification* of events or pending action

required, and they sometimes *act* for the user. Recent work that is evolving the concept of Personal Assistant agents to *user proxies*, is being implemented within a distributed agent integration architecture called Teamcore, by Tambe et al [76].

2.6.3.3 Information Agents

The Internet provides an ideal proving ground for agent-oriented software, a place to demonstrate the benefits of this paradigm shift in programming. The sheer volume and diversity of information that is accessible from the user screen, represents a media that is diametrically opposite to conventional mass media such as television. Where TV *pushes* information to the viewer, who sits passively and accepts what they get, the worldwide web (the general interface to the Internet) sits there statically, connected to hundreds of thousands of web-server, holding hundreds of millions of pages of information and interactive media, of which some dispersed minute subset will be of immediate interest to the user. With a WWW browser the user may manually click their way about in cyberspace, sporadically looking for trace elements of that subset, in an information *pull* fashion.

Search engines have dramatically eased the situation, with indexes built upon large sub-sets of the total information in cyberspace. However, various constraints upon them mean that the results obtainable from search engines are drawn from limited subsets, and linear rather than lateral in what they retrieve. For example, commercial constraints upon search engine companies have influenced many of them to enable information providers to buy priority positions under selected keywords. Such a commercial model is pushing search engines back in the direction of push media. An agent approach to finding appropriate subsets of information on the Internet is situated somewhere between *push* and *pull* technologies.

An interesting approach to raising the intelligence in finding required information, was taken by Etzioni [22]. He built *softbots* or *metacrawlers* which used numerous search engines as lower level tools which it called upon in succession, much as human users use them. His more developed softbots, such as ShopBot went further by being able to autonomously learn to extract product information from the web pages of online vendors.

The scope of finding all good information in a space as large as the Internet, has given rise to a category of agents called *Information Agents*. Given that they are intent on automating searches that a user wishes to forego personally – that they are filtering the Internet - it should be readily seen that Information Agents are a subset of Personal Assistant Agents. The scope of the task, and the positioning of it within AI and Computer Science in general, is well covered in a paper by Sterling [69]. In it he outlines the lessons learned from building three Information Agents: CiFi – a citation finder; SportsFinder – a retriever of sports results; and CASA – a classified advertisement search agent. The researchers identified three types of knowledge that substantially help in building effective Information Agents in any particular domain area:

- WWW authoring conventions
- Domain specific common sense knowledge
- Domain specific browsing knowledge.

2.6.3.4 Believable Agents, Emotional Agents

Early interface agents and personal assistant agents originally had UI components much like those seen in conventional application programs, eg. dialog boxes and drop-down menus. *Believable Agents* include a strand of personal assistant agents that have some personification of themselves, which is presented to the user. They typically have visual and audio components that help the human user believe they are dealing with a life-like intelligence.

In the same way that cartoon characters can succeed at being believable, Believable Agents attempt to get users to suspend their disbelief regarding the reality of the agent. Humans communicate through a number of different means: written language, speech, images, facial expression, gesture, tone, music, humour and other modes. Emotions as portrayed by facial expressions, gestures and tonal speech, have been used to correlate several of these modes of communication. A good survey of the approaches taken to make agents more believable by employing such human-like qualities, and the applications and environment they have been used in, is found in Elliott and Brzezinski [19].

Research into these human-like qualities tends to go down two main directions: deep reasoning about human-like qualities; and the manifestation or presentation of human-like qualities emanating from the computer, using multimedia techniques. The second stream alone, has little to do with autonomous behaviour of agents, and much to do with interface alone – so called *social interfaces*. For example, automated presentations of weather and news, such as the WWW-based animated face news reader Ananova [3], which ties together facial expression and lip movement with text-to-speech software, is not an intelligent agent as defined here in Section 2.6. However the combination of these outward expressions with deep reasoning about them, into making sophisticated believable agents, leads to the category of *Emotional Agents*. Emotional Agents use the portrayal of emotion to convey added meaning to what they are attempting to communicate to human users.

2.6.3.5 Multi-Agent Systems (MAS)

Since societal adoption of the Internet, very few computers are islands. Neither is it common for the single agent to be an island in the agent-oriented paradigm of computing. From early on in the reporting of agent research and development, multi-agent systems were often assumed to be the normal situation. As mentioned earlier in Section 2.6.2.3, Shoham's Agent-0 was envisaged as a multi-agent system from its inception. In a paper titled *Collaborative Interface Agents*, by Lashkari, Metral and Maes [45], the researchers allowed one personal assistant agent to learn from another user's personal assistant agent, based on finding two or more users with similar interests via similar personal preferences. I.e. They leveraged the benefit one finds in a society of like individuals, to overcome the shortfall in personal learning experience of an individual agent - learning from the experience of others.

The *means* in means-end reasoning for single agents are *plans*. In MAS the *means* are much more, in particular, they involve *relationships* with other agents, and the leveraged use of other agents to achieve one's goals. Goals are usually achieved via *cooperation* with other agents. Goals are often jointly held by cooperating agents, but not necessarily. Touched on in this section are some of the more significant and useful advances in MAS research, particularly with respect to cooperation and coordination of agents.

2.6.3.6 Teamwork and Collaboration

There are two streams of research that have focused on cooperation between agents in MAS systems: one is through *teamwork* amongst teams of agents; the other is through less structured *collaborations* between participating agents.

Teamwork

Like plans, *teams* are a part of the means to achieve goals. Teams are formal and often rigid representations of relationships between individual agents. In addition to reasoning about the world they are in, agents in a team can reason about the beliefs, plans and capabilities of others. This raises all sorts of design and implementation issues. There are several different approaches to having teams of agents, two of which are:

1. Tambe's [75] (ISIS/STEAM) team of agents have joint-intentions (Cohen and Levesque [12]), shared-plans (Grosz and Kraus [38]) and common goals.

2. Tidbar [77,78] has built a team-based framework on BDI in which the structure of an organisation is implicit in the design of a given MAS. Another difference from Tambe's work is that individuals in Tidbar's organisation do not need to have common goals, as long as any adopted goal is not opposite or in conflict with another member's goal.

The now annual Robocup Soccer competition (Noda et al [56]), running since 1997, was initiated to add competitive stimulation to the development of teams of agents – both robots and simulated in software - in an international context, in a popular human interest area. As a measure of the interest and success of the initiative, the fourth competition at RoboCup 2000 in Melbourne [63] attracted over 120 teams, involving more than 600 participants from 45 countries. The agent issues that continue to be the subject of team-based MAS research include: team structure; joint mental attitude; joint plans; team formation and dismantling; learning; roles and responsibilities; and dynamic modelling of the opposition. Members of teams have obligations to other members. Responsibilities and penalties for unfulfilled obligations are also areas of interest to agent learning in teamwork systems.

Teams can be viewed as a form of decentralised decision making: each member has an apriori understanding of the goals it will endeavour to achieve, and those it will leave to other team members, in a given situation.

Collaboration

Where *teams* have stark measures of success and failure - such as win or lose in a Robocup match - *collaborations* between cooperative agents, such as a coalition of information agents, are loosely coupled and have indirect benefits and indirect penaltities. Collaborations of agents are more suited to dynamic environments in which constraints and opportunities are much more fluid than in situations as well defined as a game like soccer; or in situations where there are teams of equals, each with specialist capability. In place of *obligations* driving a team member's behaviour, Castelfranchi, Dignum et al [10] explored the use of *norms* - standardized behaviour - as a constraint on agent behaviour in situations where collaboration is called for. Dignum et al [17,18] suggested that the *influences*, *norms* and *preferences* of an agent be possible mechanisms to form collaborations with other agents. The following table compares some aspects of *teams versus collaborations*.

	TEAM	COLLABORATION
Commitment	Obligation	Norms
Reward	Stark reward	Indirect reward
Penalty	Stark penalty	Indirect penalty
Goal	Team goals	Individual goals

Of importance to the success of both teams and collaborations of agents, is a mechanism that allows individual agents to reason about a new opportunity. Dignum et al [17] discuss a mechanism to allow agents to decommit from a previous commitment, but with a known penalty for doing so.

2.6.3.7 Roles and Autonomy

Relationships between agents in MAS systems are usually premised on the respective capability of the individual agents, and these capabilities are oftentimes formalised in the *roles* that agents take on in a given situation. Agent *roles* can thus be considered as a way of *typing* agents, in the way that objects are type cast by their attributes and capabilities. Cavedon and Sonenberg [11] go as far as expressing *role* as a basic modality in the agent model. They use a strict hierarchy of role to affect a built-in superior-subordinate delegation system. In their model, roles are predetermined and exploited at runtime. Goals are attached to roles. When an agent adopts a role, it automatically adopts the goals that go with it. In this sense, their role hierarchy is used to partially achieve decision making through automatic delegation: by prioritisng goals and then passing them to the team, decision making thereafter is based on the role hierarchy. Theirs is a rigid use of *role*.

Agent roles in looser collaborations or in more dynamic team structures require more flexibility in the way a role is delegated to an agent in a MAS. A *role model* for a team is one way to add such flexibility. An agent *role model* is a template for agent acquaintance and interaction. The need for such models becomes more evident, when an agent team is less formal and the team members less known than, say, in a Robocup team - eleven agents that play robot soccer together. The Robocup Federation, the organisers of Robocup Soccer, have recently instigated a competition called Robocup Rescue [43] – an attempt to bring together research in robotics and AI to support emergency rescue operation after a major disaster. An objective of Robocup Rescue is to apply technology created through Robocup activities, to socially important problems. In severe search and rescue situations such as the Kobe earthquake in Japan - the initial impetus for Robocup Rescue - collaborative teams are often formed spontaneously. A goal of Robocup Rescue is to achieve optimal collaboration with a team that has never worked together before. In such a scenario the *role model* stands to play a pivotal part in the success of such a collaboration.

Adopting a role within a MAS was seen by Castelfranchi [9] as a *social commitment* by the agent. Not surprising to us as humans in society, making a commitment to others constrains the autonomy of the individual. Likewise Castelfranchi notes that an agent socially committed to another, loses some of its autonomy. The degree of

autonomy an agent has within a MAS is still a vigorous area of agent research. For example, Tambe et al [76] introduce the term *adjustable autonomy* in a project called *electric elves*, in which the degree of autonomy an agent has can be reasoned about. And as discussed earlier in the previous section on collaboration, Dignum et al [17] are interested in a mechanism for an agent to decommit with a known penalty, in order that an agent may reason about newly unfolding opportunities. Their intent is clearly one of giving back some autonomy to overly constrained agents in earlier models.

2.6.3.8 Mobile Agents, Mobility

There are agents that stay on the one machine and interact with a multitude of other machines via communication, and then there are agents that physically move from one machine to another. An agent with this ability to move from the machine it began operation on, to another machine where it will continue to execute, or from where it will move on to yet another machine and so on – is termed a *Mobile Agent*. The CORBA Mobile Agent [15] is an example of a mobile agent specification, whose conforming agents can move their code and state across a heterogeneous IIOP network, under their own control. A CORBA mobile agent carries with it an itinerary of where to travel to next.

Generally, mobile agents operate in a secure part of each of the remote systems they visit. Telescript [74] and the Java language are two early examples of technologies that have been used to implement mobile agents. Java applets, which run across platforms via the Java Virtual Machine usually built into the web browser, are a simple form of mobile agent. Currently there is a significant convergence on Java-based technologies for mobile agents, as mentioned earlier in Section 2.4.1. The relatively recent provision of JavaSpaces by Sun Microsystems as detailed in Section 2.4.8, a system designed to aid the flow of objects across spontaneous networks, has further enhanced Java as an underlying technology for mobile agents.

Tom Malone in his aforementioned keynote address at ICMAS-2000 depicted an exponentially decreasing cost of information. He then predicted that it will be feasible some time in the not too distant future, to supply *all information to all points* - which would render mobile agents as we know them, unnecessary. Brooks et al [8] at

the same venue portrayed an intermediate concept useful within current day bandwidths. He suggested having an agent *loci* – a place where mobile agents can congregate in a MAS-enabled space - the agent equivalent of putting up a sign and broadcasting far and wide: *come here and let's haggle*.

2.6.3.9 Agent Communication Languages

Where static agents are the armchair travelers of the agent world, mobile agents are the backpackers. Static agents are appropriately more dependent on communication and protocols to get and do what they want and need. Static agents communicate in the form of declarative messages often in the form of *speech acts*. Speech act theory is based on the observation that language is not just used to state facts, but is usually driven by a *purpose* and is coloured by *attitudes,* e.g. believing, hoping, fearing. Speakers voice *speech acts* such as: requests; promises; threats. A *speaker* intends a *hearer* to receive the message and be spurred into *action* by it.

To achieve cooperation amongst heterogeneous agents requires communication in a common language. Understanding a common language involves understanding the vocabulary and how to effectively use that vocabulary to achieve one's purposes. An *ontology* is a common vocabulary with agreed upon meanings, addressing a specific subject domain. Ontologies are used in general purpose ACLs to reduce the occurrence of ambiguity and to more effectively target agents within specific domains of human knowledge.

An early and widely used ACL is KQML [25], which is a high level message-oriented communication language, one independent of content syntax and ontology. It is high level in the sense that it is above the *transport protocol layer* - the subject of other protocols such as TCP/IP; HTTP; IIOP; etc. KQML is message-oriented in that it has a *sender* and an expected *receiver*. Every KQML message represents a single speech act - eg. *ask, tell, request* – together with associated semantics via an ontology, and a protocol via a referenced language. The following KQML statement demonstrates many of the concepts of an ACL as introduced above:


```
(ask-one
        :sender fred
        :content (PRICE IBM ?price)
        :receiver stock-server
        :reply-with ibm-stock
        :language LPROLOG
        :ontology NYSE-TICKS)
```

Figure 2.18 – KQML Code fragment, one message.

The first line in figure 2.18 holds the performative *ask-one*, then follow keyword-value pairs, which represent the argument of the performative. The keywords – *sender*, *content*, *receiver*, *reply-with*, *language* and *ontology* – adequately introduce their function to the reader. The parameter after content – (PRICE IBM ?price) – in this example is an anti-tuple (also known as a Template) as described earlier in Section 2.4.6, one that is expressed and understood in the language LPROLOG, and is used to retrieve a matching tuple from the intended receiver *stock-server*. The content of a KQML statement is language independent, achieved by naming the content-language in the *content* parameter.

In addition to KQML there is an ACL called FIPA ACL [26] currently in the early stages of evolution, but conceived after and in the light of the specification of KQML. FIPA is the Foundation for Intelligent Physical Agents, a non-profit association (www.fipa.org).

2.6.3.10 Hybrid Reactive-Rational-Interactive Agents

Bratman, Israel and Pollack [7] point out that real agents are *resource bounded* – that an agent is operating within time and other constraints. Consequently, during the time spent in deliberation, chances are that their world has changed in important ways – possibly ways that undermine the assumptions on which current deliberation is proceeding. One aspect of their work is concerned with efficiency of the deliberation process. They outline a *compatibility filter* – a filtering process which drops sub-plans from ongoing deliberation, those which are incompatible with already adopted plans.

Bratman et al express the incompleteness of the agent's knowledge of unfolding events in the world, as the *knowledge boundedness* of the agent. They assert that a rational agent's current adopted plans (intentions) should therefore be revocable. That sometimes an agent should reconsider current intentions in the light of new events, triggered by some criteria of interest and preference being flagged. They proposed a *filter override mechanism* encoding the agents sensitivities to *problems* and *opportunities* that arise, which can affect the deliberation process. In their proposed architecture such new opportunities are dealt with by an *opportunity analyser*. Looking at the BDI-interpreter algorithm presented back in Section 2.6.2.2, access to new events in the world during the deliberation process is done via the following line of the algorithm:

```
get-new-external-events();
```

What Bratman et al have described, and what is available within the BDI-interpreter algorithm, is a degree of *reactivity* within an otherwise deliberative rational process. This property of BDI agents is sometimes described as *reactive planning*. It is in this sense that BDI implementations are usually *hybrid rational-reactive systems*. Going a step further and embedding BDI-like agents in a MAS framework, or into a system where agents and humans interact, and they can be considered to be hybrid *rational-reactive-interactive* agents (Sections 2.6.2.1, .2 and .3): so that the resultant agents may be considered *self-aware*, *situated-aware* and *socially-aware*.

3 HCI MODELS, METAPHORS AND ARCHITECTURES

Where distributed systems involving CORBA and object-oriented Tuplespace systems are usually concerned with *server-side* objects and information, *client-side* architectures dealing with the Human Computer Interface (HCI), have been centred around a *user interface model*, concentrating on the interaction between objects and the actions people take to do things with them. The modeling of a user's thought processes has been a central aspect of user-centric interactive systems. The following seven stage *action model* of people *doing things*, was proposed by Norman [58]:

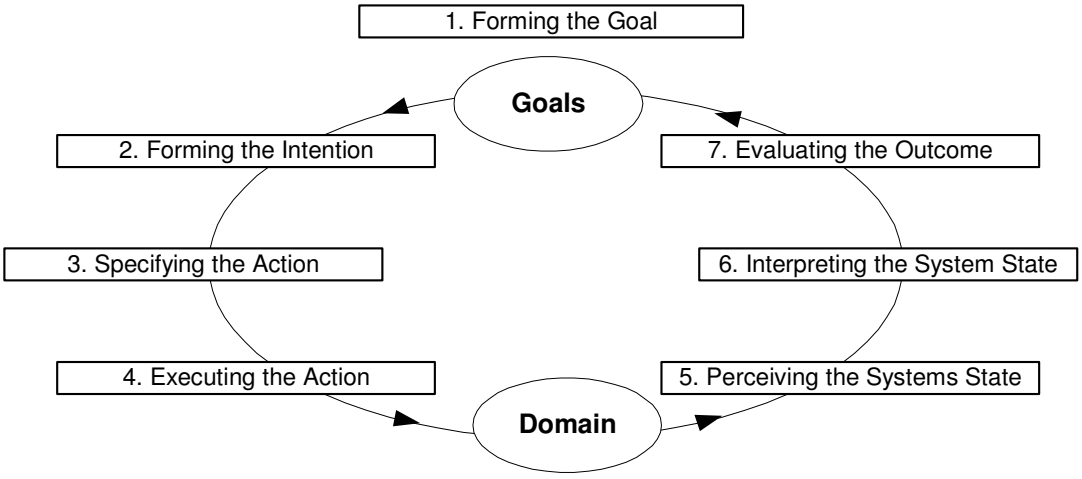

Figure 3.1 - Norman's 7-stage Action Model.

There is one stage for *goals*, three for *execution* and three for *evaluation*. He termed this model an *approximate model*, rather than a complete psychological theory, conceding that in practice the stages are not discrete, nor necessarily sequential, and most behaviour does not go through all stages. Nonetheless, the model is the basis of many formalisms in user interface design, and has been a significant shaping force in the design and development of many user input devices, real and virtual.

The interface between human and computer, is a place where human actions - originally developed and refined in the natural world - have been harnessed to manipulate virtual and abstract objects that are typically made visible as patterns of brightly coloured pixels on a flat screen. Abstraction and metaphor are used to achieve tasks on or via the computer. In Figure 3.2 below, Shneiderman captures this transition, of *objects and actions* in the natural world, to *objects and actions* in the interface of computer software, in his Object-Action Model [66].

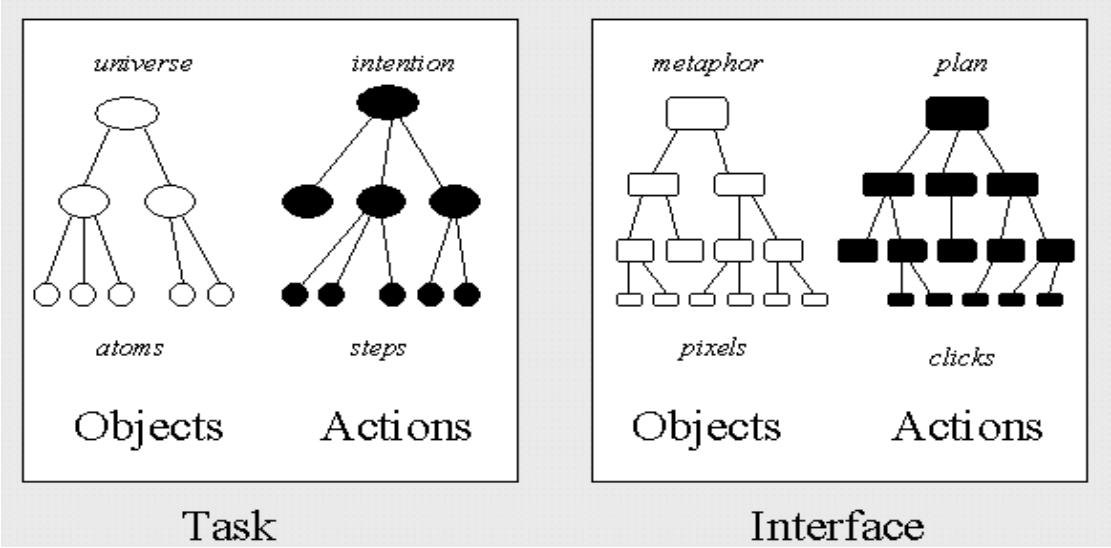

Figure 3.2 - Shneiderman's Object-Action Interface Model.

The following sections cover a number of models, architectures and frameworks that necessarily deal with the user-centric aspects of client-side computing of interest to our 24x7 interface to the network.

3.1 UI Metaphors and Devices

The *desktop* metaphor popularised by the Apple Macintosh and then Microsoft Windows 3.1 and evolutions of it, is the currently most widely used metaphor in client GUI systems at the primary interface level.

At a level less than the whole interface, a number of *Virtual Devices* and *Abstract Devices* have become prevalent in the client GUI. The following four definitions are provided to clarify similar but quite different terms used with respect to UI devices:

Physical Device - these input devices include common staples: Mouse, Keyboard and Joystick.

Virtual Device - simulating a physical device on the screen is a virtual device. Eg. A 3D-like image of a button on the screen, is a virtual device.

Abstract Device - some screen-based device that doesn't have a direct physical equivalent, for example, the tree representation of the file system.

Logical Device - is an abstract model of input, defining the point of view from the software. Eg. Multiple user actions such as: selecting a Quit menu; clicking an icon in a toolbar; typing an accelerator key combination such as Control+Q - will all quit a program, because the program sees them as paths to the same *logical device*.

The *abstract device* which is the *tree representation* of the file system in Windows Explorer, continues to be a dominant metaphor within the Windows desktop with respect to managing the lifecycle of files and the directory structure (new, delete, move, copy). However, once you step into the application programs themselves, the most highly used devices are *virtual devices* such as: Menu selection; Form fill-in and Dialog boxes, including tabbed dialog boxes.

Human-Agent Interaction Metaphor:

In an expose on the design of human-agent interaction, Lewis [46] declared that there were just two dominant classes of UI metaphor which between them included all others:

1. Environment
2. Agent

In Lewis's categorisation of metaphor, the desktop metaphor and metaphors involving navigation in information spaces (such as Windows Explorer, and Internet

Explorer) are all within the *Environment* class of metaphor. His definition of a software agent, is a program that automates some stage of the human-information-processing cycle, one intended to automate repetitive, under-specified or complex processes (a less rigorous definition of Intelligent Software Agent as we have seen in Section 2.6). With that definition in place, the agent metaphor of interaction treats the computer as an intermediary between the user and agents.

An example Lewis uses to differentiate the two metaphors, is a comparison of certain features in two different styles of text editing:

- *Direct-manipulation* in screen-based editor.
- *Search all occurrences*, in command-line editor;

Highlighting text within a document on-screen with the mouse, and then clicking the italics button on the toolbar, is an example of *Direct Manipulation* – itself, an example of interaction in the environment metaphor. A menu item found in all serious modern editors, is the *Search All* option, used to find all occurrences of a particular sub-string in the whole document, typically way beyond that portion displayed on the screen. *Search All* is an example of the agent UI metaphor.

Though the *direct manipulation* approach in particular within the environment metaphor is superior for one-off and other tasks, the *'search all occurrences'* menu command in modern screen-based editors - a carry-over feature from earlier command-line editors - retains its place in modern editors. So the *agent metaphor* of user-computer interface is superior for particular operations, particularly repetitive ones, even within the heart of products that are heavily designed around the environment metaphor.

Direct Manipulation in UI:

The example immediately above is one of many user operations that are collectively termed, *direct manipulation*. Another, that was introduced with GUI systems and the mouse, is the drag-and-drop operation, heavily used in current mainstream GUI systems. The number and type of direct-manipulation operations is increasing as new devices – physical, abstract and virtual devices – are employed to increase the flexibility and usability of the interface in realistic 3-D games and Virtual Reality

applications. In currently popular real-time strategy games such as Age of Empires [1] and StarCraft [68], hundreds of objects (simulated soldiers, vehicles, animals, buildings, etc) can easily and quickly be selected, inspected and manipulated, using direct manipulation techniques.

Shneiderman and Maes [67] attribute much of the success of direct manipulation interfaces to the *constraints on behaviour* introduced by the visual objects and actions they accept, thereby reducing the possible tasks and human effort in initiating them. In a point-and-click environment, the planning of tasks and executing the actions, is narrowed down to selecting a visible on-screen object – enforcing constraints on the user's *intention* and the system's *action*. Immediate feedback is available for the user to evaluate whether the execution was as intended or otherwise.

Referring back to Fig. 3.2 above - Shneiderman's Object-Action Interface model – as a reference for a further explanation of these acknowledged gains in usability: *direct manipulation* can be seen as a *bringing together* of the *task* domain on the left, to the *interface* domain on the right, by an appropriate design of the right visual objects and actions, good metaphors or models of representation, coupled with appropriate physical devices.

3.2 The MVC Architecture

A forerunner to the object-oriented paradigm was the Smalltalk-80 language which came with a complete graphical interface built into the environment in which all Smalltalk programs ran. The MVC (Model View Controller) user interface (UI) architecture first appeared as the model of interaction for the Smalltalk80 language [30] within that graphical interface. It was devised as a particular strategy for *representing*, *displaying*, and *controlling* information. According to Krasner and Pope [44], the strategy was chosen to satisfy two primary goals: to create a set of system components to support a highly interactive software development process; and to provide a general set of system components that make it possible for programmers to create portable interactive graphical applications.

Moving forward from Smalltalk to the more recent Java language introduced 15 years later, it also uses the MVC model to implement its *'pluggable look-and-feel'*;

whereby, you may switch the look-and-feel of an application written in Java, from say a Microsoft Windows style interface to an Apple MAC interface, on the one machine at the click of a menu item.

The MVC architecture consists of three main conceptual elements (see Figure 3.3), as follows:

Model stores structure and state data.

View deals with display of data.

Controller synchronizes data between model and view, schedules interactions, and enacts eve nts or messages.

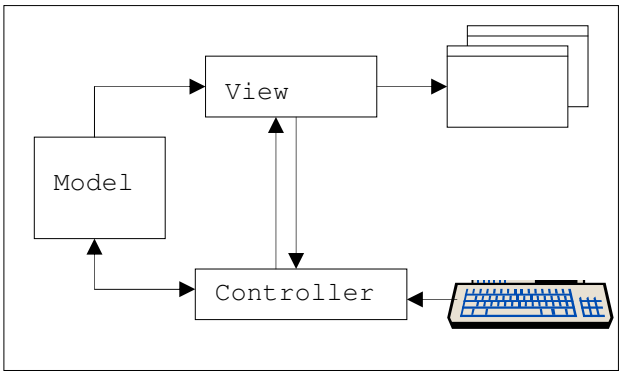

Figure 3.3 - The MVC Architecture.

MVC interaction within an event model goes as follows: The *Model* maintains a list of *views* that have registered with the Model for change notification. When a change is made to the model it notifies each registered *View*. Views then usually retrieve extra information from the Model, in the process of updating themselves. These notifications are done by *events*. Events are handled by the *Controller*.

As a demonstration of enacting MVC in Java, one that is simpler than describing the built-in pluggable look-and-feel functionality, we have set up a simple application - a calculator - which can be changed at any time by the user from a grid-layout of buttons, to a vertical column of buttons. This switch in screen layout of components,

allows for more screen real estate for other applications, as needed. See Figure 3.4 below.

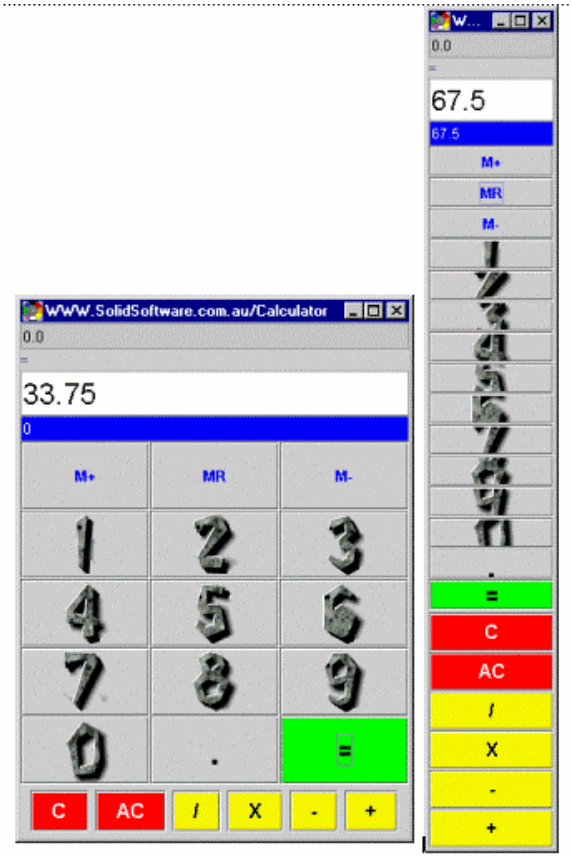

Figure 3.4 - Dynamically switching Layout Managers in a Java Application.

The built-in *Layout Managers* of Java are constraint-based patterns for the layout of on-screen components, dynamically arranging the inclusive components as the screen-size varies or as window size is changed by the user. They allow Java programs to have adaptive interfaces with respect to occupying screen real-estate. For example, the traditional grid layout of calculator buttons, as in the left-side of Figure 3.4, is controlled by the standard Grid Layout Manager. Typically, as the calculator's window changes size - either by the user or as enforced by a different client screen size - the buttons within the grid will dynamically resize to maintain the grid formation. However, within a Java application, it is possible to dynamically swap the Layout Managers themselves, causing an entirely different view of the interface objects to appear on screen, as we have done with the calculator on the right-hand-side of Figure 3.4. In MVC parlance, the Grid Layout Manager and the vertical Box Layout Manager, used respectively for the left and right screen images in

Figure 3.4, represent two different *Views*. The array of buttons and all that they entail independent of the Views, is represented by the *model*, while the event processing code that manages the switch between the two Layout Managers, is the *Controller*.

3.3 SlimWinX

As briefly introduced in Section 2.4.9 *SlimWinX* is a GUI interface for DOS-based systems which uses *XSpaces* to manage dynamic objects coming into and going from the memory of a SlimWinX application. SlimWinX has numerous standard window features such as multiple graphic windows, container windows, buttons (multi-stage icon buttons, including fully animated Active Objects), button toolbars, dialog boxes and a built-in object-oriented event-handling system. In addition, it has several advanced GUI features that aide direct manipulation programming, in particular:

- a *direct manipulation* interface including an extensive mouse *two-click* drag-and-drop facility;
- a context-sensitive *balloon help* system using a *help wand*;
- dynamic memory management (via XSpaces as described in detail in Section 2.4.9);
- semi-modal dialog boxes (pop-up windows);
- dynamic interactive objects;

expanded on below:

3.3.1 Two-click Drag-and-Drop

To enact a *drag-and-drop* sequence in SlimWinX, the user issues two separate clicks of the left-mouse-button, rather than holding down the button for the duration of a drag - as is the convention in Microsoft Windows and several other mainstream GUI systems:

1. First click (click+release) *grabs* the moveable object (normally an icon representing an object). I.e. The icon latches onto the cursor. See the *three-coin*

icon being dragged in Fig. 3.5 below. The cursor changes to a grabbing-hand image. As the cursor is moved about, the grabbed icon goes with it, visually centred beneath it. At this stage, you do not hold the mouse button down as you would under a MS Window's *drag*.

2. A second click (click+release) *drops* the moveable object at the current location. If that particular destination part of the screen can accept that particular type of object being dropped, then it will stay there, or even *go into it*, as per dropping the *three-coins icon* into the *mailbox button* in Fig. 3.5. If that destination cannot accept that object-type, then the image *floats* back to the spot from where it was first *grabbed*.

Figure 3.5 – Drag-and-Drop action onto a Mailbox Button, in SlimWinX application.

In Fig. 3.5, note that the small window (with its own small toolbar) to the upper-right of the *briefcase* button, *contains* the *contents* of the *briefcase.* The *briefcase* in SlimWinX is functionally equivalent to the *clipboard* in Microsoft Windows. Also note:

- that the three-coins icon has just been dragged from the contents of the briefcase, enroute to the mailbox;
- the picture of an *open* briefcase on the button in the down position (when its content window is closed, the button reverts to the up position and a graphic of a *closed* briefcase appears;

- a SlimWinX button can represent a *container object*, as the *mailbox* and *briefcase* buttons do in this example program;

- the use of different *cursors* during different functions for visual feedback to the user, i.e. *the hand* cursor on the coins is used only during a *drag* operation.

We designed and implemented the above *two-click* drag-and-drop action for two reasons, both revolving around the initial target markets of *education* and *entertainment* for *SlimWinX* technology:

1. The *MS Windows* convention of holding the mouse button down for the duration of the *drag operation*, is awkward and prone to inaccuracy for the very young - and for every user when a mouse or mouse-mat is not operating in consistently near-perfect condition, or during long distance drags, or if the user's train-of-thought gets interrupted during a drag.

2. It was designed to support three different physical input devices: *keyboard, mouse* and *joystick*. The two-click *drag-and-drop* sequence also suits keyboard and joystick button pushing.

However, the two-click drag-and-drop also suits other non-mouse control devices that are now being used in 24x7 network operation, such as the infrared remote control on set-top boxes.

3.3.2 Context-sensitive Balloon Help

Context-sensitive help delivers help messages that are tailored for just the local context of the situation at hand, reducing the need for a user to search through a comprehensive contents-list, or manually stepping through a help-search function. The *context-sensitive* help system within SlimWinX is accessed and implemented via the *Help Wand*.

In Figure 3.6 below, the *far right button* on the toolbar has a *question mark* graphic upon it, which is the *Help Wand* – the tool a user engages to get context-sensitive help within SlimWinX applications. The user can pick up the question mark graphic in the manner of a *drag-and-drop* icon, except that the cursor *becomes* the

Help Wand rather than just attaching to the icon graphic. With the question mark now functioning as the cursor, the user can click on any object visible on the current screen in order to bring up a *Balloon Help* message specific to that object. The term *balloon* is borrowed from the cartoon-like call-out text box which represents speech in the cartoon genre of the Visual Arts. SlimWinX places the balloon either, up, down, left or right of the chosen object, depending on constraints from the current screen layout.

Figure 3.6 – Use of the Help Wand to access the context-sensitive Help Balloon.

When the user has completed their interactive investigations with the Help Wand, they need to park it back on the button top from where it came, in order to regain normal UI operation - done with a single click.

3.3.3 Semi-Modal Dialog Boxes

SlimWinX has *modal* and *non-modal* dialog boxes. In UI terms a user is in a particular *mode* whenever they must cancel or complete what they are doing, before they can do something else. Modes force users to focus on specifics about an application, rather than let them go about the tasks they may want to in an adhoc order – such as requiring a user to enter a legitimate filename when saving a new file. As such, modes interfere with the conceptual model of how an application should work, and so they are used sparingly in GUI's. A *modal dialog box* provides a *serial dialog* with the user, which they must finish in a program-anticipated way, before the user can interact with any other window associated with the application. Conversely, the user can interact with *non-modal* dialog boxes and windows in parallel with other

windows and components in the same application, without needing to complete the current dialog.

When a dialog box is modal and when it becomes the *most recently* opened window on the screen, it has the *focus* - meaning that all user *events* (keyboard commands, mouse clicks) are then directed to that window for processing first - until it is closed. E.g. In systems such as MS Windows, you have to *Close* a modal dialog box before you are able to affect other windows and objects on the screen, outside that window.

SlimWinX modal dialog boxes are much more flexible than the standard modal dialog box - they are better described as *semi-modal*. These semi-modal dialog boxes allow certain user actions such as mouse-clicks, outside the current *modal* window. Such mouse clicks are processed by the current dialog box *first*, but then, *if appropriate* (i.e. *allowed* by the application programmer), may be processed by windows that were opened before the current dialog box.

An example is needed to best highlight the difference between modal and semi-modal. Referring back to Fig. 3.6 the briefcase window with its own smaller toolbar, is an example of the flexibility of semi-modal windows in *SlimWinX*:

- The briefcase window in this situation is a semi-modal dialog box. Keyboard and mouse events meant for the buttons in the toolbar within the dialog box, will reach their destination. However, clicks outside the briefcase window, though still processed by the dialog boxes event-handling code first, may alternatively activate things outside the current dialog box, for example: *dropping* an object into a container object (the mailbox), which lies outside the dialog box.

- To exit the semi-modal dialog box, *the user* needs to press the screen-button with *'ESC'* upon it, as you would the *Close* button in a MS Windows modal dialog box. However, it will also be closed down by *the application* itself, if an object is dropped on an appropriate external object (eg. the mailbox container button). To enact this behaviour, SlimWinX *generates* several new *events* from the single user click, which go into an *event queue* and get executed in the correct order as follows: A *close_modal* event is issued first, followed by a *drop_object* event.

The flexibility gained in SlimWinX by having such a *semi-modal* dialog box component, has significantly enhanced the *direct manipulation capability* of the GUI in the following ways:

- Implementation of the drag-and-drop feature itself was streamlined by having a semi-modal dialog box. I.e. Drag-and-drop was not an afterthought as it has been in most mainstream GUI's, but was an initial design feature of SlimWinX.

- Implementation of the context-sensitive balloon-help facility was streamlined by having a semi-modal dialog box. The help button is implemented as a *semi-modal dialog box with just one button* – the help button, which fully occupies the dialog box visually. The user is kept in *help mode* while that dialog box is open. The semi-modal dialog box enables mouse-clicks all over the application, particularly outside of the help button (a semi-modal dialog box), enabled context-sensitive event messages to be generated and acted upon, without closing down help mode. The dialog box can only be closed, by clicking on the help button a second time. I.e. Context-sensitive help is simply programmed as a *special-case* semi-modal dialog box. It is a special-case only in that every SlimWinX application must have it, and the context-sensitive event type is hard-coded.

- In addition, drag-and-drop direct manipulation programming by the application programmer, is simplified to the setting of just two parameters: one telling a particular *SlimWinX* object that it is *draggable* (eg. the three-coins icon in Figure 3.6 has such a flag set); another telling a particular container component that it can have objects *dropped upon it*. (eg. the mailbox container button, has such a flag set).

3.3.4 Dynamic Interactive Objects

The advanced features of SlimWinX outlined so far, are modifications of the way that conventional GUIs do similar things: drag-and-drop, modal dialog boxes; help systems. However, the *dynamic interactive object* feature of SlimWinX doesn't have equivalents in current mainstream GUI systems. The following three figures – Figures 3.7, 3.8 and 3.9 - represent snapshots from an interactive sequence of user

actions, as they move a dynamic-interactive-object from one fullscreen window, to another fullscreen window in the same application program.

Step 1 (Fig. 3.7): The user clicked the *right* mouse-button (rather than the left) to grab the Mailbox container button as seen in the earlier examples, from the toolbar of the current window, and is currently dropping it into the Briefcase container button.

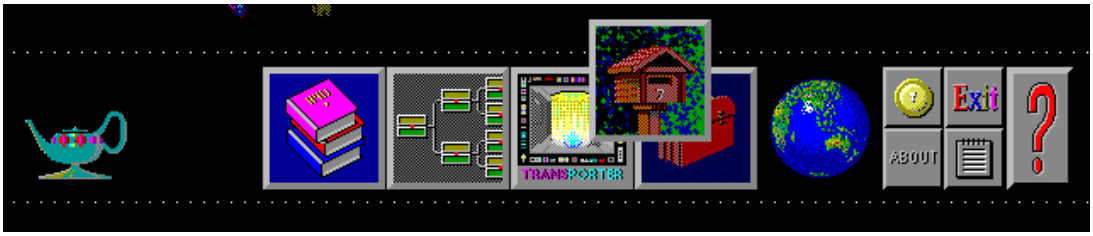

Figure 3.7 – A Dynamic Object is drag-and-dropped into the Briefcase container button.

Step2: The user then navigates to another fullscreen window in the program, by application specific means.

Step 3 (Fig. 3.8): The user now accesses the contents of the Briefcase button (via a smaller version of the button this time, in the 2x2 button cluster) from within the other fullscreen window. There are two objects in there: a candelabrum icon and the recently placed Mailbox container button. The user grabs the Mailbox by clicking it with the right mouse button (clicking the left button would open the mailbox).

Figure 3.8 – Accessing the Briefcase contents from within a different window.

Step 4: The user drops the mailbox button onto the toolbar of the new window – the toolbar has been flagged during development by the application programmer, to accept this type of object dropped upon it – where it can now be accessed like the other buttons in the toolbar.

Step 5 (Fig. 3.9): The user now accesses the contents of the newly placed Mailbox container button, having left-button clicked it, and is now in the process of grabbing the three-coin icon, which is the only object currently in the mailbox.

As outlined previously in the discussion on two-click drag-and-drop, any object within *SlimWinX* can have a parameter which says whether or not it can be *drag-and-dropped* - setting that parameter is all the application programmer needs to do to make *any* SlimWinX component draggable. As has been elaborated upon immediately above, such objects can be fully functioning parts of the application, dynamically taking their functionality with them. This makes for a very fluid UI, however, the SlimWinX application programmer has considerable control over where such objects can be dropped, as legitimate target object containers have to have the appropriate flag set.

Figure 3.9 – Step 5: After the drop the contents of the Mailbox are displayed intact.

Dynamic-interactive-objects are implemented in SlimWinX, courtesy of the XSpaces system. As discussed in Section 3.4.9 the XSpaces system implements the *in* or *take* operator common in tuplespace-based languages, interactively via the drag-and-drop operation more fully explained in this section. Remember that the XSpace of the application from which the above figures were taken, is represented in Appendix A. The operation discussed in the above three figures, would cause the rearrangement of the sub-trees in Appendix A into a slightly different tree configuration – which can be stored persistently between uses of a SlimWinX application.

Event Driven:

SlimWinX is an event driven system, which is running within a continual event loop with an event queue to cope with *get-dispatch* performance mismatch. The algorithm is as follows:

```
Initialize(...);
While (not quit)
{
        Get_event (event);    // Event types: Mouse; Keyboard, Internal, Idle
        Dispatch_event (event);
}
```

The handling of events is object-oriented, i.e. non-handled events get passed back to parent objects.

3.4 The Netscape Navigator V4 Object Model

JavaScript is a scripting language built into the web browser Netscape Navigator. A clone of JavaScript called JScript is built into the Microsoft Internet Explorer. In addition to being termed a scripting language, JavaScript is also described as an object-*aware* language, able to address and manipulate a number of pre-existing objects within the web browser window. The JavaScript programmer cannot make classes of their own as they are able to in an object-oriented language. The Object Model available to JavaScript is outlined in Figure 3.10. On the second level of the hierarchy there is a node, two from the right, which represents the *document object* - which is a runtime in-memory version of the HTML file currently occupying the window of the executing browser. The various elements of the HTML document are all available to the JavaScript language as instantiated objects, addressable in a hierarchical manner, for example the JavaScript statement:

```
parent.frame[0].document.forms[0].elements[3].options[2].text
```

references the text of the third option (numbering starts at 0), of the fourth element with the first form, within the first frame, of the current HTML file in the browser window in question.

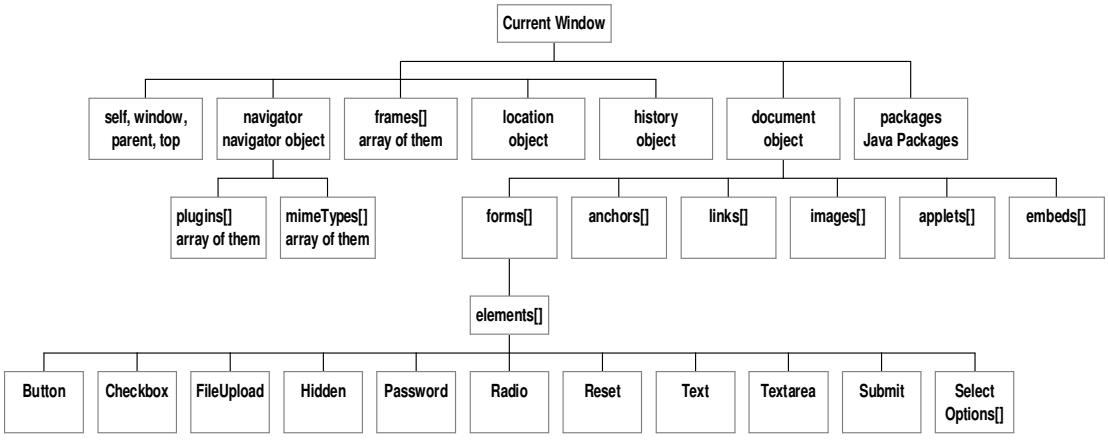

Figure 3.10 - The JavaScript V1.1 Object Hierarchy

The JavaScript language has a number of *event handlers* which bind user interactivity via the keyboard and mouse to the JavaScript coding, such as: onClick(), onMouseIn(), onMouseOut, and so on.

3.5 The W3C DOM, HTML and XML

The internal object model within the web browser from Microsoft (Internet Explorer), though used to display and enact the same HTML files as Netscape Navigator, is significantly different from that above in Figure 3.10. This variation between internal *object models* in different proprietary web browsers, led to discrepancies in the way the different browsers display the same HTML file. More significantly, the discrepancies in the way the scripting languages (JavaScript, JScript) in the browsers dealt with those objects, burdened JavaScript programmers with writing different multiple scripts in some situations, to get the same interactivity with the same HTML file. The collective disquiet arising from those discrepancies amongst the global community of web browser users and content developers, led the W3C organisation to develop an open standard *document object model*, called the W3C *Document Object Model* (DOM) [83].

In parallel to the specification of a DOM, in an ongoing effort to evolve and complement the HTML language, the W3C specified XML [23] (the eXtensible Markup Language) as a markup language that specifically dealt with the *structure* of information, rather than the *presentation* of information in the browser window. HTML had originally combined presentation and structure using a simple *hard-coded tags* approach to markup. The *presentation* aspect of markup was refined and extended within the Cascading Style Sheets (CSS1 and CSS2) revisions of the HTML V4 standard [39], and in other standards such as XSL [24]. As such, the DOM specification addresses both HTML and XML, and the W3C encourages all web browser and other software developers with an interest in XML and HTML documents, to program to the DOM specification.

The DOM then, is an application programming interface (API) for XML and HTML. It is an *interface* not an *implementation*, an interface in the object-oriented sense discussed in Chapter 2: proprietary applications *must implement* the specified

interfaces in order to claim compliance. Indeed, the DOM is specified in CORBA IDL. However, in the latest Level 2 of the DOM specification there are eight modules:

1. *Core*,
2. *HTML*,
3. *Views*,
4. *StyleSheets*,
5. *CSS*,
6. *Events*,
7. *Traversal*,
8. *Range*,

and not all of them have to be implemented in an application to achieve *DOM compliance*. The *DOM Core* represents the functionality in dealing with XML documents but also forms the basis for dealing with HTML documents and is divided into two subsections: *Fundamental Interfaces*; and the *Extended Interfaces* (not relevant to HTML documents). A compliant application must implement *at least*: the Fundamental Interfaces in Core; and *either* the HTML module or the Extended Interfaces in Core. All of the other six modules are optional. Hence, an XML Editor as an example DOM application program, need not deal with HTML at all to be W3C DOM compliant. This is significant as XML and DOM have far more relevance to client-based systems than just HTML and the browser.

The term *document* is used in DOM in a broad sense, as XML files also often hold data. The DOM specifies how XML and HTML documents are represented *as objects* so they may be used consistently in various object-oriented programs. The DOM is typically a hierarchical tree of nodes. These hierarchies do not represent data structure, they represent objects with *identity* and *functionality*.

Hierarchical and Flattened Views of the API

The Core DOM interfaces collectively represent two distinct approaches to accessing and manipulating a document: an object-oriented approach which deals with a hierarchy of inheritance; and a linked-list approach that deals with the nodes in the order they appear in the document. That is, an implementation of the DOM must be able to access the object version of a document as both a hierarchy of nodes and as a linked list of nodes.

In the definition of a DOM Document Node via the Node Interface defined in CORBA IDL, outlined in Figure 3.11 below, we have bolded the attributes and methods which are indicative of these two approaches to access and manipulation.

```
interface Node {
        // NodeType
        const unsigned short   ELEMENT_NODE = 1;
        const unsigned short   ATTRIBUTE_NODE = 2;
        const unsigned short   TEXT_NODE = 3;
        const unsigned short   CDATA_SECTION_NODE = 4;
        const unsigned short   ENTITY_REFERENCE_NODE = 5;
        const unsigned short   ENTITY_NODE = 6;
        const unsigned short   PROCESSING_INSTRUCTION_NODE = 7;
        const unsigned short   COMMENT_NODE = 8;
        const unsigned short   DOCUMENT_NODE = 9;
        const unsigned short   DOCUMENT_TYPE_NODE = 10;
        const unsigned short   DOCUMENT_FRAGMENT_NODE = 11;
        const unsigned short   NOTATION_NODE = 12;

        readonly attribute DOMString  nodeName;
                 attribute DOMString  nodeValue;
                              // raises(DOMException) on setting
                              // raises(DOMException) on retrieval
        readonly attribute unsigned short    nodeType;
        readonly attribute Node       parentNode;
        readonly attribute NodeList   childNodes;
        readonly attribute Node       firstChild;
        readonly attribute Node       lastChild;
        readonly attribute Node       previousSibling;
        readonly attribute Node       nextSibling;
        readonly attribute NamedNodeMap attributes;
        readonly attribute Document   ownerDocument;

        Node       insertBefore (in Node newChild, in Node refChild)
                                              raises(DOMException);
        Node       replaceChild (in Node newChild, in Node oldChild)
                                              raises(DOMException);
        Node       removeChild (in Node oldChild)    raises(DOMException);
        Node       appendChild (in Node newChild)    raises(DOMException);
        boolean    hasChildNodes ();
        Node       cloneNode (in boolean deep);
        boolean    supports (in DOMString feature, in DOMString version);

        readonly attribute DOMString  namespaceURI;
        attribute DOMString           prefix;
```

```
        readonly attribute DOMString  localName;
};
```

Figure 3.11 – DOM Document Node Interface in IDL.

The intended functionality built around these attributes and methods of the Node interface are better understood by looking at the interfaces defined in the optional *Traversal* module, in particular, the *TreeWalker Interface* represented here in Fig. 3.12 and the *NodeIterator* Interface in Fig 3.13 below.

The methods are in bold font:

```
// Introduced in DOM Level 2:
interface TreeWalker {
  readonly attribute long            whatToShow;
  // Constants for whatToShow
  const unsigned long       SHOW_ALL                      = 0x0000FFFF;
  const unsigned long       SHOW_ELEMENT                  = 0x00000001;
  const unsigned long       SHOW_ATTRIBUTE                = 0x00000002;
  const unsigned long       SHOW_TEXT                     = 0x00000004;
  const unsigned long       SHOW_CDATA_SECTION            = 0x00000008;
  const unsigned long       SHOW_ENTITY_REFERENCE         = 0x00000010;
  const unsigned long       SHOW_ENTITY                   = 0x00000020;
  const unsigned long       SHOW_PROCESSING_INSTRUCTION   = 0x00000040;
  const unsigned long       SHOW_COMMENT                  = 0x00000080;
  const unsigned long       SHOW_DOCUMENT                 = 0x00000100;
  const unsigned long       SHOW_DOCUMENT_TYPE            = 0x00000200;
  const unsigned long       SHOW_DOCUMENT_FRAGMENT        = 0x00000400;
  const unsigned long       SHOW_NOTATION                 = 0x00000800;

  readonly attribute NodeFilter       filter;
  readonly attribute boolean          expandEntityReferences;
           attribute Node             currentNode;
  Node               parentNode();
  Node               firstChild();
  Node               lastChild();
  Node               previousSibling();
  Node               nextSibling();
  Node               previousNode();
  Node               nextNode();
};
```

Figure 3.12 – TreeWalker Interface of the DOM Traversal Module, expressed in IDL.


```
// Introduced in DOM Level 2:
interface NodeIterator {
  readonly attribute long             whatToShow;
  // Constants for whatToShow
  const unsigned long       SHOW_ALL                          = 0x0000FFFF;
  const unsigned long       SHOW_ELEMENT                      = 0x00000001;
  const unsigned long       SHOW_ATTRIBUTE                    = 0x00000002;
  const unsigned long       SHOW_TEXT                         = 0x00000004;
  const unsigned long       SHOW_CDATA_SECTION                = 0x00000008;
  const unsigned long       SHOW_ENTITY_REFERENCE             = 0x00000010;
  const unsigned long       SHOW_ENTITY                       = 0x00000020;
  const unsigned long       SHOW_PROCESSING_INSTRUCTION       = 0x00000040;
  const unsigned long       SHOW_COMMENT                      = 0x00000080;
  const unsigned long       SHOW_DOCUMENT                     = 0x00000100;
  const unsigned long       SHOW_DOCUMENT_TYPE                = 0x00000200;
  const unsigned long       SHOW_DOCUMENT_FRAGMENT            = 0x00000400;
  const unsigned long       SHOW_NOTATION                     = 0x00000800;

  readonly attribute NodeFilter       filter;
  readonly attribute boolean          expandEntityReferences;
  Node               nextNode();
  Node               previousNode();
};
```

Figure 3.13 – NodeIterator Interface of the DOM Traversal Module, expressed in IDL.

In DOM, Node lists and node trees are *live* – meaning that the structure of the document can be dynamically altered within the compliant application program, and the other functions in the API are required to gracefully cope with those dynamic changes.

You may recall from Section 2.4.9 that the XSpaces system can be accessed as both a hierarchy and as a double-linked list, and that the dynamic-interactive-objects of SlimWinX, as described in Section 3.3, are enacted in the code by dynamically altering the tree structure of a SlimWinX application, *live*. There is a considerable likeness of attributes and methods between a DOM interface and some of the similar implemented methods within XSpaces – to a degree that it would be feasible to make XSpaces WC3 DOM compliant with little effort.

The architects of DOM clearly have multiple presentation systems in mind for XML, evident in the *Stylesheets module*, which defines base interfaces that are to be used to

represent "*any type of style sheet*". This is opposed to the *CSS module*, which is succinctly designed for HTML and XML web documents*: "Cascading Style Sheets is a declarative syntax for defining presentation rules, properties and ancillary constructs used to format and render Web documents."* [83]

The W3C DOM, initially inspired by HTML's universality, represents the first cross-platform, cross-language open standard that addresses *client-side objects* – an object domain usually black-boxed within proprietary GUI systems such as Microsoft Windows and MAC OS. (note: An earlier attempt was made with the proposed OpenDoc [13] standard by OMG, but which has since been abandoned.)

`DOM Event Model`

Another interesting addition to the DOM specification at Level 2 of the document, an aspect of acute interest to client-side object systems, is the specification of a generic *event model* in the *Event Module*. The primary goal of the Event Module interfaces is the design of a generic event system that:

> "*… allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. Additionally, the specification will provide standard sets of events for user interface control and document mutation notifications, including defined contextual information for each of these event sets*."

The description earlier, of an ability to modify the tree-structure of a DOM document *live*, is termed a *mutation*. An event that caused the change of the tree structure in an application – such as the drag-and-drop operation of a dynamic-interactive-object with the right mouse button in SlimWinX, described in Section 3.3 – is termed a *Mutation Event* in the DOM Event Module specification.

The terminology used in the DOM Event Model includes:

UI events

> User interface events. These events are generated by user interaction through an external device (mouse, keyboard, etc.)

UI Logical events

Device independent user interface events such as focus change messages or element triggering notifications.

Mutation events

Events caused by any action which modifies the structure of the document.

Capturing

The process by which an event can be handled by one of the event's target's ancestors before being handled by the event's target.

Bubbling

The process by which an event propagates upward through its ancestors after being handled by the event's target.

Cancelable

A designation for events which indicates that upon handling the event the client may choose to prevent the DOM implementation from processing any default action associated with the event.

To introduce here something of the flavour of the specification of the standard *set of events* to deal with general user interface control, the MouseEvent Interface is shown below in Figure 3.14:

```
// Introduced in DOM Level 2:
interface MouseEvent : UIEvent {
  readonly attribute long             screenX;
  readonly attribute long             screenY;
  readonly attribute long             clientX;
  readonly attribute long             clientY;
  readonly attribute boolean          ctrlKey;
  readonly attribute boolean          shiftKey;
  readonly attribute boolean          altKey;
  readonly attribute boolean          metaKey;
  readonly attribute unsigned short   button;
  readonly attribute Node             relatedNode;
  void               initMouseEvent(in DOMString typeArg,
                                    in boolean canBubbleArg,
                                    in boolean cancelableArg,
                                    in views::AbstractView viewArg,
                                    in unsigned short detailArg,
```

```
                                    in long screenXArg,
                                    in long screenYArg,
                                    in long clientXArg,
                                    in long clientYArg,
                                    in boolean ctrlKeyArg,
                                    in boolean altKeyArg,
                                    in boolean shiftKeyArg,
                                    in boolean metaKeyArg,
                                    in unsigned short buttonArg,
                                    in Node relatedNodeArg);
};
```

Figure 3.14 – MouseEvent Interface of the DOM Event Module, expressed in IDL.

Mouse event *types* may then be the usual: *click, mousedown, mouseup, mouseover, mousemove, mouseout*. There is a similar but much larger event set for the keyboard event type which is not listed here. A set of pre-existing HTML events is included for backwards compatibility with web-browsers: *load, unload, abort, error, select, change, submit, reset, focus* and *blur* – several of which apply only to HTML forms, rather than all elements in the DOM. And a set of event types under the umbrella of the Mutation Event described above in figure 3.14, is included in the specification:

- *DOMSubtreeModified*
- *DOMNodeInserted*
- *DOMNodeRemoved*
- *DOMNodeRemovedFromDocument*
- *DOMNodeInsertedIntoDocument*
- *DOMAttrModified*
- *DOMCharacterDataModified*

the names of which, amply allude to their function in mutation of a document with a DOM compliant application, from an introduction point of view.

3.6 Summary

While Chapter 2 covered relevant background technology and programming paradigms, although still background material this chapter turned attention to the client side of computing systems, to aspects of the user interface. To enable an agent based Digital Self within the operating system of a device connected to the Internet 24x7, an intersection of the backend agent technology with the HCI of the system must occur. With that in mind we looked at Norman's seven stage model, which shows that as with software agent paradigm, HCI too has been concerned with psychological models. We looked at Shneiderman's Object-Action interface model for its clarity of explanation of the work done at the interface, and at UI metaphors. We saw that Lewis used *agent* as a category of metaphor, a definition quite different yet related to the software agent paradigm view of *agent*.

The MVC model was overviewed and will be revisited later in Chapter 5 where we look at the intersection of the Shadowboard agent system, with the GUI of an operating system. Innovative aspects of the author's SlimWinX GUI system were overviewed, as a precursor to Chapter 6 where the implementation strategy for building the Shadowboard architecture built upon SlimWinX and XSpaces, is detailed. The W3C DOM client object model - which arose from web browser *wars* between Microsoft (Internet Explorer) and Netscape (Navigator) - was examined. It is drawn on later in Chapters 7 and 8 as a possible future research direction with regard to interoperability between a Shadowboard based Digital Self and third party agents.

4 ANALYTICAL PSYCHOLOGY AS MODEL

In the introduction of this thesis the concept of a Digital Self agent was initially portrayed as one holding a sophisticated model of the user, one capable of acting as a proxy or surrogate for the user, a pseudo individual in the user's absence. However, to also increase an individual user's effectiveness, efficiency and influence, in an empowering manner, the concept of a Digital Self needs to go beyond a surrogate, it needs to augment the user's skills and abilities in a synergy of human and software embedded qualities. To make such a Digital Self at all possible, one needs from the outset, an architecture based on a *model of mind*, one capable of representing a sophisticated degree of self-awareness including self-known capabilities.

In the coverage of Chapters 2 and 3, a number of the models and architectures provide an interesting lead into the subject of this section. In his approach Norman [58] outlined a model of *action*, one of the human user getting particular things done, framed in the context of the user interface. However, it lacks *deliberation*, except in the sense of *evaluating* less than successful actions in the past. It is a model about *process* and as such it lacks a model of the human mind at any level, instead assuming that a human mind is ever present throughout such situations.

Shneiderman's object-action model outlined in Chapter 3 is concerned with user interfaces and devices, and is primarily a tool for understanding or measuring the usefulness of a metaphor within the user interface. In translating a user's intentions to actions in a world that is not physical, the user has a *mental model* of the application and how his/her physical actions put plans into effect within a virtual environment, typically on a screen. When a user interface specialist talks of *mental models* it is usually these *models of the application* the user holds in his/her head, that concerns them. Interface agents too, have a mental model of an application – one which hopefully approximates the user's mental map of the application. The sophistication aimed for here with the Digital Self is the agent's *mental model of the user* and how the user deals with situations and entities.

Where user interface research has long been concerned with augmenting the user's productivity and skills and has enlisted cognitive psychology in doing so, the agent paradigm has ventured into psychological models of humans that include and go beyond cognitive psychology. Driving this trend has been the goal of making agents as *autonomous* as possible – agents being the culmination of decades of work in AI and robotics. Psychology itself has been described as *the science of autonomous agency*, so it is not surprising that agent research has enlisted psychology for insights and techniques that help achieve autonomy. In this section we examine more complex psychological models of mind, in particular drawing from analytical psychology, and we select a contemporary model upon which the Shadowboard agent architecture is based.

4.1 Beyond Folk Psychology

The BDI architecture based on Folk Psychology [60] and introduced in Section 2.6 is perhaps the most psychologically sophisticated model to date, to be employed within operational agent frameworks. The BDI Agent architecture calls upon the *mentalistic* notions of *Beliefs*, *Desires* and *Intentions* as *course* abstractions to encapsulate the hidden complexity of the inner functioning of an *individual* agent. BDI is a course-grained approach to the use of such mentalistic notions, and as such, a starting point for an intra-agent model. A finer grained *intra-agent* model based on psychological notions should result in a much stronger degree of agent self-knowledge (*self-awareness*), which in turn should improve an individual agent's performance within a social MAS environment.

Others have expressed the desire for individual agents to be more psychologically sophisticated, more dependent on human psychology, so that they may function and interact effectively within our human social systems [45, 79]. Watts' aim is for agents and humans to cooperate and otherwise socially interact, in more effective and useful ways. He would like to build agents that interact with humans and can stand in for humans - sophisticated *Interface Agents* and intelligent *Personal Assistant Agents*. Such a definition of agency draws upon human social intelligence as an ideal rather than just as a metaphor, and hence upon an individual person as a psychological *archetype* for an individual agent.

In *Society of Mind* [51] Minksy takes a reductionist view of the human mind in which every describable process (e.g. Add, Move, Grasp) is considered an agent, so that a brain would be made up of many millions of such agents. When he did discuss the higher constructs within the mind he generally referred to cognitive functionality, such as: *'that large part of the brain concerned with vision'*. Nonetheless he did discuss some higher more abstract levels of organisation of mind, such as the *Self* and the *Conservative Self*. Yet he was troubled with *'hidden'* aspects of mind, most evident in a section he titled: *Self Knowledge is Dangerous* - a point of view that is contrary to cross-discipline and cross-cultural wisdom.

Suppes et al [73] describe *representation* - in the sense of modelling within psychology - as a way of reduction of a more complex structure. They discuss two forms of *reduction* that employ representation: the first characterises a set of theoretical concepts with another, giving the example of Descarte's reduction of geometry into algebra; the second, describes (noisy) data in the context of parameters that capture the main tendencies, in a pattern recognition fashion. BDI, which takes advantage of the reduction of *behavioural concepts* into *mental concepts*, is an example of the first. The approach detailed in this thesis incorporates both the first and second forms of representation, to reduce the complexity of modelling an agent - an approach based on finer grained patterns of *behaviour of subselves* within the whole self.

Western psychology has many rich branches of psychology other than behavioural folk psychology, from which one could draw models of agency including: *cognitive psychology, analytical psychology, humanistic psychology, developmental psychology*. *Shadowboard*, the agent architecture developed within this thesis, draws upon *analytical psychology*, in particular a contemporary psychology encompassing refinements of Freudian [72], Jungian [42] and Assagiolian [4] concepts, not only for metaphor but also for structural and computational implications.

Shadowboard uses abstractions of mentalistic notions based on a well documented and clinically supported psychology revolving around *subselves* at work within the psyche of an individual. The approach is known as the *Psychology of Subselves,* an attempt to understand the whole personality of an individual - in order to model consciousness, deliberation and action [4,65,71]. Note: *subselves* are also known as

sub-personalities and the two terms are used interchangeably within this thesis and else where.

4.2 Psychology of Subselves

This section aims to give a brief description of the *Psychology of Subselves* and the background within psychoanalysis from where it came.

Main exponent:	**Sigmund Freud**	**Carl Jung**	**Roberto Assagioli**	**Hal Stone & Sidra Winkelman** (Psychoanalysis & Psychosynthesis)
Technique	Psychoanalysis		Psychosynthesis	Voice Dialogue
Model divisions of the human psyche	Ego	Persona	Centre	Aware Ego
		Self, self	Self	
	Super Ego	Higher Self		Protector/Controller Inner Critic
	Id (repression)	The Shadow		Several *Disowned Selves*
		Anima/ Animus	Many sub-selves	Pusher, Pleaser, Parental selves, many other subselves
	Mental Objects*	Archetypes	Evolved subselves	

* Concept of *Mental Objects* has been added to Freudian Psychoanalysis by contemporary Freudians Klein & Fairburn

Table 4.1 - Lineage of subpersonality exponents and some of their divisions of the psyche.

Consider the voice, gestures and language a person uses when talking to a child, then compare them to those of the same individual when he/she talks to one of their own parents. In such comparisons it is often possible to glimpse the facets of two of the subselves within the psyche of the one individual. The subselves within, often accompany different roles a person has in his/her outer life, but not always. However, one gets to see very few of the subselves in the external persona of an individual. Sliker [65:32] noted: *"Jung described in detail the persona, the personality mask developed to meet the world safely. Subpersonality knowledge reveals the extreme limitations of the persona. Usually one or two subpersonalities perform the function of persona, while perhaps the rest of the personality is rarely seen in public."*

The lineage of subselves in psychology is scantly represented in Table 1 above, not to under-appreciate it, nor to strictly categorise one against the other, but to indicate that the psychology of subselves, though based on relatively modern developments in psychology, has been evolving since the start of western analytical psychology. Sliker points out that in 1907 Freud, then aged 51, Jung, then aged 32 and Assagioli, then aged 19, all met, and she observes that: *"Although their careers overlap, Freud, Jung and Assagioli quite literally represent three generations of thought on subpersonality."*

4.3 Freud, Jung and Psychoanalysis

Both Sigmund Freud, the pioneer of psychoanalysis, and Carl Jung once a student of Freud who significantly advanced psychoanalysis, were basically dealing with pathology in the form of mental disorders in their patients, for most of their practicing lives. There they learned much about the human mind. A distant and unflattering parallel could be made to *boundary analysis* within software verification and validation: much can be learned of the inner workings of a system, at the points where they breakdown. Likewise, the human pathologies in the form of neurotic and psychotic mental states of the majority of Freud's and Jung's patients are pathological exaggerations of normal mental states that most people hold. In this thesis the primary interest in psychoanalysis is in the subdivisions of the mind that first Freud and then Jung chose to name and describe. However, something still needs to be said here of the *unconscious mind* in order to frame their respective subdivisions of the mind.

Psychoanalysis is a procedure for the investigation of mental processes that are usually inaccessible by other means. Psychoanalysis has evolved into a science built upon the systematic accumulation of knowledge about the mind for over 100 years.

4.3.1 Some Freudian Concepts

Conscious, Preconscious, Unconscious:

Consciousness can be defined as the current or real-time awareness of feelings, thoughts, memories, perceptions and fantasies, at any given waking moment in a life. Preconsciousness refers to the memories, feelings, perceptions, etc, not currently held

in the conscious mind, but which are readily available and can enter consciousness without much difficulty. For example a past memory may jump into the *conscious* mind, from *preconscious* mind. The *unconscious* mind consists of the other mental processes and mental material which is not easily accessed in the conscious mind.

Psychoanalysis deals with the unconscious mind and has built up a body of knowledge and theories (some of them competing theories) about how the mind is mentally subdivided. Freud was primarily interested in the unconscious, in particular the chaotic elements of the mind and the instinctual processes.

Freud's Id, Ego and Super Ego:

Freud first identified and named several generic patterns or distinct parts of the psyche that have become a part of his legacy to psychology, and which mark the beginning of the recognition of subselves in the formal study of western psychology, namely the: *Id*, the *Ego,* and the *Super Ego*.

Id: That part of the mind directly concerned with instinctive drives. The id has no regard for time or the consequences in the real world, of satisfying its drives. As such Freud described the id as a part of mind driven by the *pleasure-principle*. The id is unconscious.

Ego: The ego is the executive function of the personality, the choice maker. The ego also includes that part of the mind that builds an internal model of the external world, modifying it as new perceptions override older ones and filtering out inaccuracies of the model generated internally. The ego is very much aware of time. The ego is governed by the *reality-principle*, constraining the id according to constraints in the external world, while also trying to satisfy the id. The ego is conscious and rational.

Super Ego: The Super Ego is the idealist part of mind upholding norms of behaviour, derived from family, society and often religion. The super ego burdens the ego with another set of constraints on top of those coming from the external world situation, in its task of satisfying the

id. The super ego is further removed from the perceptual system than the ego, and is usually preconscious.

Note: Referring back to the BDI agent model in Section 2.6.2.2, one can see how closely BDI's *Beliefs*, *Desires* and *Intentions,* match the domain of Freud's three divisions of the psyche:

Freudian concept	**BDI mental construct**
Id	Desire
Ego	Intentions
Super Ego	Beliefs

Given that Freud is the father of Analytical Psychology and that the above mapping is so strong, we make the claim that the psychological roots of the BDI agent model is the Freudian model of mind, intentionally or otherwise.

Some other Freudian concepts:

Repression: Freud described the ego's burden as that of one serving three masters: the id, the external world and the super ego. Inevitably, in such a complex role, disturbing thoughts, conflicting urges and traumatic events arise that could overwhelm the ego, but for its defense mechanisms, the predominant one of which is to make them or the memory of them, unconscious. This ability to bury them in the unconscious is termed *repression*.

Free Association: Repressed material in the unconscious can hold such psychic energy that it imbalances the life of the individual. Freud's brand of psychoanalysis for such troubled individuals involved uncovering the repressed event or other material causing the problem. Uncovering the repressed event would usually involve the patient telling the analyst whatever thought came into his/her head, no matter how silly or senseless or unpleasant the thought might be. What he/she told him, and from what he/she also hedged around, would usually lead the

analyst to what he/she were repressing – this Freud called *Free Association*.

Bringing the repressed event or thought into the conscious mind of a patient, though often distressing the patient, will unlock the psychic energy, and bring a new healthier balance of such energy in his/her mind. The interrelated goals of Freudian therapy are to strengthen the ego as follows: to widen its view of the psyche by making conscious, aspects of mind which were previously unconscious; to include new portions of the id within its organisation and rational behaviour; to make it more independent of the super ego.

4.3.2 Some Jungian Concepts

In *Man and his Symbols,* Jung's last book and the only book he (co-)wrote with a view to reaching the layperson in psychology, with the concepts derived from his life's work, Jung put a prominent emphasis on the use of *symbols* in human communication and within the mind. In short, he saw symbols as representing the natural language of the mind, and he could clearly see the reason for it.

With mankind's limitation in knowledge about any particular thing – i.e. our knowledge of something is limited by our perceptions of it via our senses, and then by the tools we use to extend our senses (eg. microscopes and telescopes too, have their limitations) – our comprehension and understanding of *anything* is never complete. The more something is unknown to the mind, the more the mind uses symbols to represent it, so that it can deal with it in a concrete manner. "*A word or an image is used symbolically, when it implies something more than its obvious and immediate meaning. It has a wider 'unconscious' aspect that is never precisely defined or fully explained"*. [Note: A parallel in AI is the use of *Beliefs* - as in the BDI agent model - in lieu of *facts,* in a situation where all the facts are not known.]

We humans produce and use symbols spontaneously and unconsciously within our dreams. Like Freud before him, Jung relied primarily upon the interpretation of dreams to build up his knowledge on the psyche. In his lifetime he analysed more than 80,000 dreams.

Conscious and Unconscious Mind:

Jung first diverged from Freudian views with respect to the information content and usefulness of dreams. This lead to a significantly different Jungian view of the unconscious mind compared to the more simplistic view of it by Freud. Where Freud viewed the unconscious as little more than a cesspool of repressed desires and buried traumatic events, Jung saw it as a vital, helpful and complex part of an individual. To Jung, the type and structure of dreams, as well as the symbols and their meanings were infinitely variable. Where Freud saw the value of dreams as merely starting material to get to the repressed parts of an individual via his analytic tool *free-association*, Jung saw much more complexity in dreams. Jung saw dreams as holding valuable information usually in highly condensed symbolic form, meant from the unconscious mind, for the conscious mind. I.e. Jung saw *intention* in dreams, the intention to communicate helpful information from the unconscious mind to the conscious mind. To the Jungian, the *unconscious* is at least half of the individual, and is a source of significant inner advice and guidance available no where else. This immediately leads to a belief in at lease two significant selves in a single individual, the conscious and the unconscious – at least two subjects in the *subject-object* situation.

While the unconscious mind includes *repressed content* – memories one is very happy to lose - it also holds information that has never been available to the conscious mind. To partially explain the source of such 'new' information from the unconscious mind, Jung points out that the unconscious is perpetually taking in content subliminally:

"Every experience contains an indefinite number of unknown factors … There are certain events of which we have not consciously taken note; they have remained, so to speak, below the threshold of consciousness. They have happened but they have been absorbed subliminally without our conscious knowledge" [42, p.23]

Very often, this subliminal material appears in dreams, not as rational depictions but in symbolic form. However, what is of interest to this thesis is the parallel processing identified by psychoanalysts - the functioning of multiple aspects of mind in a single individual, of which he or she is not immediately conscious. To understand these

subdivisions observed in the human mind is of the utmost importance to building a Digital Self. The connection of a functional Digital Self to a 24x7 network while the user is asleep or otherwise occupied is analogous to subliminal intake of information by the unconscious mind in everyday human situations.

Jungian subdivisions of mind:

The subdivisions of mind that Jung contributed to analytical psychology are: *Persona, Self, Higher Self, Shadow, Anima* and *Animus*.

Persona: The aggregate of a person's behaviours (via roles in his or her life) used in negotiating and confronting the social world.

Shadow: The unrecognised aspect of personality, usually unwanted and pushed into some corner of the unconscious, where it may be projected onto other people. Jung's concept of *Shadow* is evolved from Freud's concept of Repression – the Shadow is a whole personality of repressed material.

Anima: Represents the feminine qualities within a man, often repressed.

Animus: Represents the masculine qualities within a woman, often repressed.

Self: Totality of the psyche and its functional centre, which (should or eventually) overrides the ego. Jung did not see the Self as directly attainable.

Higher Self: A deeply unconscious, ever present guide, the totality of being. In the realm of modern day psychology, as it moves into the territory of religious systems, the energy known as the Higher Self is the most elusive and mysterious part of the psyche. I.e. It is the most personal and has the least amount of consensus amongst psychologists.

Archetypes: Jung considered *archetypes* as the basic elements for psychic experiences. In terms of subpersonality, archetypes are simple, clear and powerful expressions of the qualities of that personality.

Some other Jungian concepts:

Individuation: A process of unfoldment (gaining of knowledge of the inner subpersonalities) in which the Ego is brought into line with the Self. Jung's idea of "individuation" depends on the differentiation and integration of psychological complexes [84].

Introverts/Extroverts: Jung's terminology, which have long since entered common usage, for two distinct personality types via four basic psychological functions: *Thinking; Feeling; Sensation; Intuition. Thinking* is the opposite of *feeling*; while *sensation* is the opposite of *intuition*. If energy is generally put into the world, these types are *extraverted*. If energy is taken from the world, these types are introverted.

4.4 Assagioli and Psychosynthesis

While Freud and Jung were basically dealing with pathology all their practicing lives, Assagioli was most interested in the human development of *'healthy'* individuals. The advocates of each, still lean in those directions: Jungians are generally occupied with building *strong foundations* by uncovering flaws and misplaced energy in the psyche, while Assagiolians are more intent on *'building splendid skyscrapers'* upon assumed solid foundations, under the heading of Psychosynthesis. Although Assagioli expressed his ideas of a synthesis of the subselves in the psyche as early as 1909, they only found a ground-swell in adoption within the humanistic psychology movement of the 1960s and 70s. Psychosynthesis always promoted self-help in its approach to psychology, perhaps a reason why it does not have the large and prosperous bodies of practitioners that Psychoanalysis does.

Psychosynthesis involves the identification and development of the subpersonalities - mostly identifiable within the conscious mind, to be made always available to conscious thinking as dependable tools for the individual. Psychosynthesis involves embracing all parts of the psyche in a team-oriented harmony, by taking them out of conflict with one another.

4.4.1 Assagiolian subdivisions of mind

Subselves: Subselves or subpersonalities are a foundational fact in the Psychosynthesis view of the mind. Subselves are pattern-oriented, and most often associated with roles that an individual performs in his/her daily life. Sliker makes the point that at least since the Renaissance, the individual has consciously been aware of and proactively developed subpersonalities: *"The Renaissance man, by definition, is a person of multiple subpersonalities. Such a person may be a scientist, a playwright, a musician, and a lover."* Subpersonalities contain personal histories, the individual's war stories. They have inclusive safety rules based on the individual's past history for use in the future. In this sense, subpersonalities take a division of labour approach to what both Freud and Jung placed upon the ego alone. The aspects of the outer world that are modeled in the mind and the reality-tests that are performed against those models are divided between the subselves.

Evolved Subselves: Having identified the subselves, the remaining goal of Psychosynthesis is to develop them into better versions of themselves – the *evolved subselves*. Every subpersonality has a useful talent that can be developed and used constructively. This is done by consciously choosing an archetypal figure or mentor that represents a clear and powerful expression of the qualities of that personality, and aiming to improve the subself in their direction.

Centre: *Centre* is the neutral open space, apart from the subpersonalities. This is equivalent to Voice Dialogue's *Aware Self*, covered in the next section. Assagioli saw the absolute importance of self-identification through disidentifying from the subpersonalities, of moving out of subpersonality constriction and into the openness of self. In most people, by the time they reach maturity, Centre is lost via a complete identification with a dominant subpersonality or two. Centre can be experienced in several natural situations, Eg. in a vast natural setting devoid of any of the trappings of society; in an emergency, when

comprehension of information and the great need to act quickly, can bypass the subpersonalities.

Given that pathology is not something one would normally build into a software agent, it might seem tempting to base an agent model simply upon the Assagiolian principle of subselves - one devoid of any aspect of Jung's concept of the *Shadow*. To emphasise the point, in her coverage of Psychosynthesis, Sliker sees the goals of Psychosynthesis as the transformation of the subselves into *'well polished tools'* that can bring about effective action – even the language seems ready made for basing software agents upon. However, what is not immediately obvious to the layman in psychology, is that Freud's repressed events, thoughts and experiences, and Jung's Shadow, are not confined to the mentally unstable and the depressed. They are represented in some form and degree in every individual that has had to live within the constraints imposed by the reality of the world, which is all of us.

4.5 Voice Dialogue – a contemporary *Psychology of Subselves*

Jung's concept of the Shadow, the parts of the personality that Assagiolians choose to gloss over, represents facets of personality repressed or projected. It is not only an interesting aspect of people with respect to explaining and predicting their overall behaviour, it is also the engine-room behind relationship-building between individuals, dysfunctional or otherwise, in the social world. Therefore, with respect to *inter*-agent activity between individual agents based on Shadowboard, an Assagiolian approach would not be enough. So, the model of subselves adapted for Shadowboard is from a psychotherapy technique called *Voice Dialogue* developed by Stone and Winkelman [71]. Voice Dialogue is founded on Jungian analytical concepts, but has also drawn refinement from Psychosynthesis, in its own effective synthesis of those earlier bodies of psychology. [Note: We are not interested here in the therapy aspect of any of the psychologies, just the *models of mind* they give us with respect to deliberation, cooperation and action.]

4.5.1 Common Dominant Subselves

As with Psychosynthesis, Stone and Winkelman take the base position that the human psyche consists of a number of subselves, usually dozens in a mature adult, and often associated with the roles a person has in his/her complex modern life. However, they identify and name several generic subselves that they can readily identify in most people: *Protector/Controller; Pusher; Inner Critic; Pleaser; Perfectionist; Inner Child; Parental Selves*. Without going into the explicit definitions here, the names themselves are sufficient to allude to their functions. There are many other less dominant subselves a given person will have developed, or is still prototyping, to fulfill the roles he/she carries out as he/she carefully negotiates his/her way in the outer world.

4.5.2 Archetypes

In addition to their own agendas, the inner subselves often work in cooperation, in small teams. For example: the *Inner Critic* might make a person feel bad about not knowing enough, then the *Pusher* will step in by compiling a reading list to help the person improve themselves. The Inner Critic is also using *archetypes* as ideal versions of a subself, against which to measure and judge the individual's worldly actions and thoughts. In this sense, archetypes are much like mentors towards which an individual's beliefs and actions are tailored.

4.5.3 Disowned Selves

Stone and Winkelman saw subselves as energy patterns, and saw illness of many sorts as blockages of such energy patterns. They realised that the various facets of Jung's *Shadow* were better understood as a group of *Disowned Selves* - subselves that have been disowned through bad experiences in the past: via behaviour that was not socially rewarding; or behaviour which exposed the individual to danger in the past. It is not always bad behaviour; it could have been something that a person was naturally good at, but which cast them into the limelight (e.g. dancing), which in turn made them vulnerable (e.g. peer-group pressure). So, *disowned selves* are the result of an individual surviving and evolving within the social system that is the society

about them. [Note: The main difference between this modern view of repression and that by Freud, is that Freud dealt with the repression of events and similar memories, while Voice Dialogue understands that a whole subpersonality can be repressed, one holding the memory of some traumatic event or experience that came about via activity which that subself was engaged in at the time.]

However, these old aspects of self are not just *disowned*, they are often also *projected* onto other people. The disowned selves with a negative (anti-social) energy are usually projected onto people that an individual strongly dislikes, someone who exhibits the disowned traits to some extreme. The positive disowned energies are most often projected onto a partner or other friend, and hence play an important part in relationships. Continuing the example of the person with a *disowned dancer*, it might be projected onto a partner or potential partner who is very comfortable at such public performances.

Stone and Winkelman make the point that we need to become aware of the disowned subselves within, for our own sakes [71:243]:

> *"The energies we continue to disown will return to us in some form to plague us, defeat us, and cause us stress. These disowned energy patterns behave like heat seeking missiles, launched by this creative intelligence; inevitably, they find their way to their disowned counterparts of us. Thus they demand our attention through discomfort they cause upon impact."*

Though it is new advice from the modern field of Psychology, there is evidence of age old advice which supports the idea. In 1945 an ancient Gnostic text called the *Gospel of Thomas* was discovered in an urn in the cliffs at Nag Hammadi, Egypt. It dated from about 200AD, a time when the orthodox church culled the number of about 20 gospels - stories about the life of Christ - to just the four gospels that make up the New Testament. People of the time were heavily encouraged to dismiss those that missed the short list, including the Gospel of Thomas. A piece of advice reported in the Gospel of Thomas, attributed to Christ, is:

> *"If you bring forth what is within you, what you will bring forth will save you. If you do not bring forth what is within you, what you do not bring forth will destroy you."*

That is a good summary of the goal of Voice Dialogue in unearthing and integrating one's disowned selves in positive and useful ways, and of the failure to do so.

We will see further down, that the concept of disowning a subself is a very useful mechanism for the way in which sub-agents are put on hold in Shadowboard. Disowned selves also give us a psychologically inspired *handle* for inter-agent relationship building.

4.5.4 The Aware Ego

One of Stone and Winkelman primary advances is on *awareness*. Historically, in psychoanalysis the Ego is seen as the executive function of the personality - *the decision maker* - but it is also seen to be in control of awareness, at least during consciousness. Stone and Winkelman see the need for individuals to actively separate the *awareness* aspect of mind from the *control* part. To them, awareness is clear space that just *witnesses*, not attached to outcomes. They see a successful result of their work on a person, as someone who has developed an *Aware Ego* - an executive decision maker that calls upon a purely awareness function of mind, one in full knowledge of all the subselves and able to call upon them individually or in teams, to optimally negotiate the challenges of life.

Of course achieving an Aware Ego involves much more than just making a commitment to attain it, but that is the starting point of the development process. Next the Aware Ego *observes* – the doing in the world continues to be done by the appropriate subselves. Later, it takes on an advisory capacity. Once the subselves recognise the value of that advice, through good outcomes in conscious life, transformation of the Ego to an Aware Ego begins. The Aware Ego takes on management duties, evolving into an Executive Director of the whole self. Nonetheless, the Aware Ego has no fear as it is the subselves that have the safety rules implicit in the life experience they hold in memory - so once delegation has placed a task with a subself, they handle the relevant local decisions (domain knowledge, appropriate sub-plans) - until pulled up by the executive function of the Aware Ego, for some overriding reason.

However, *passion for development* resides in the Aware Ego. Subself passion will result in imbalance in the psyche, and the Ego may be taken over again by a dominant subself. And further, the *decision* to pursue an Aware Ego does not in itself prevent trauma and calamity from happening in the individual's world, so the whole process is usually one of cyclic refinement.

The Aware Ego is the model upon which the central agent in the Digital Self is built. It will be the basis for the primary decision maker within the Digital Self, a dominant sub-agent fully aware of all the other sub-agents and external agents, that it may call upon to do certain tasks.

5 SHADOWBOARD ARCHITECTURE

The Shadowboard architecture is the result of applying the *Psychology of Subselves* as introduced in Chapter 4, to an analogous software architecture capable of being the foundation for an agent as sophisticated as the Digital Self outlined in the introduction of this thesis. Beyond analogy, Shadowboard also draws structural and computational implications from the Psychology of Subselves. This chapter maps the psychological terms to like-named or like-functioned sub-agents within a whole Shadowboard agent, and explains their function and how they interact with one another.

5.1 Shadowboard as Metaphor

The Shadowboard agent architecture draws its *name* from a tool-based metaphor – from the place-holder for workman tools in a workshop, a *shadowboard* upon which fasteners are attached and a silhouette of each tool painted in black against a white backboard (see Figure 5.1). Each sub-agent is metaphorically seen as a tool in a large repertoire of such tools. Each sub-agent has a specific set of known capabilities and knowledge, each knows its place, and a central agent knows where to find each of them. Groups of similar sub-agents form classes of agents, just as the drill-bits to the right side in Figure 5.1 represent a class of tools of similar function, but each with their own specialisation. For example, a class of sub-agents might be domain specific search agents.

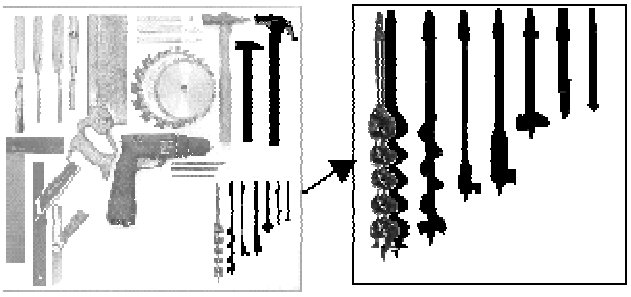

Figure 5.1 – A workshop tool-based Shadowboard as Metaphor.

However, Shadowboard doubles as a psychological metaphor in which sub-agents represent subpersonalities (also known as subselves), one that is consistent with Carl Jung's concept of *the Shadow*.

5.2 Architectural Overview of a Shadowboard *Whole Agent*

Figure 5.2 below structurally represents the Shadowboard architecture, collectively representing an individual *whole agent* made up of numerous sub-components. A brief description follows immediately, one which is expanded upon in the sections that follow.

In the centre of the agent is the *Aware Ego Agent* – the dominant sub-agent in the whole cluster of sub-agents. In the figure, the Aware Ego Agent is surrounded by eight first-level sub-agents, diagrammatically drawn as circles the same size as the Aware Ego Agent. Five of these example sub-agents are not nested any deeper (i.e. sub-agents can be clustered recursively), while the other three have clusters of circles within them, representing a second-level of sub-agents, grouped into numerous *envelopes-of-capability* – analogous to the drill-bits class of tools in Figure 5.1.

Each *envelope-of-capability*, of which there are arbitrarily eight in the figure, represents different areas of expertise that a particular whole agent embodies. As such the whole agent could perform a number of consecutive and diverse tasks, depending on what goals via what roles it has taken on in the outer world.

Each envelope-of-capability contains a number of sub-agents with similar capabilities, but each different from it neighbours in some specialised way. At the two far ends of an envelope-of-capability are two diametrically skilled agents - in figure 5.2 the two are adjacent, one is white the other is black, separated by a dotted radial line:

- One is the *basic reactive sub-agent* a purely reactive agent, as described in Section 2.6.1, with a hard-wired rule-action mechanism and no deliberative capability. The basic reactive sub-agent is called upon within a particular envelope-of-capability when time or other resources are severely constrained.

- At the other end of the scale is an *archetypal sub-agent*, one that has maximum deliberative ability used when time and other resources are plentiful.

The *aware ego agent,* as well as each of the other sub-agents (*delegation sub-agents*) that are shaded as spheres, have *knowledge about the capabilities* of sub-agents in their envelope-of-capability. They use this knowledge to select the appropriate sub-agent to achieve the particular goal that has been sent their way, from higher up the recursive hierarchy.

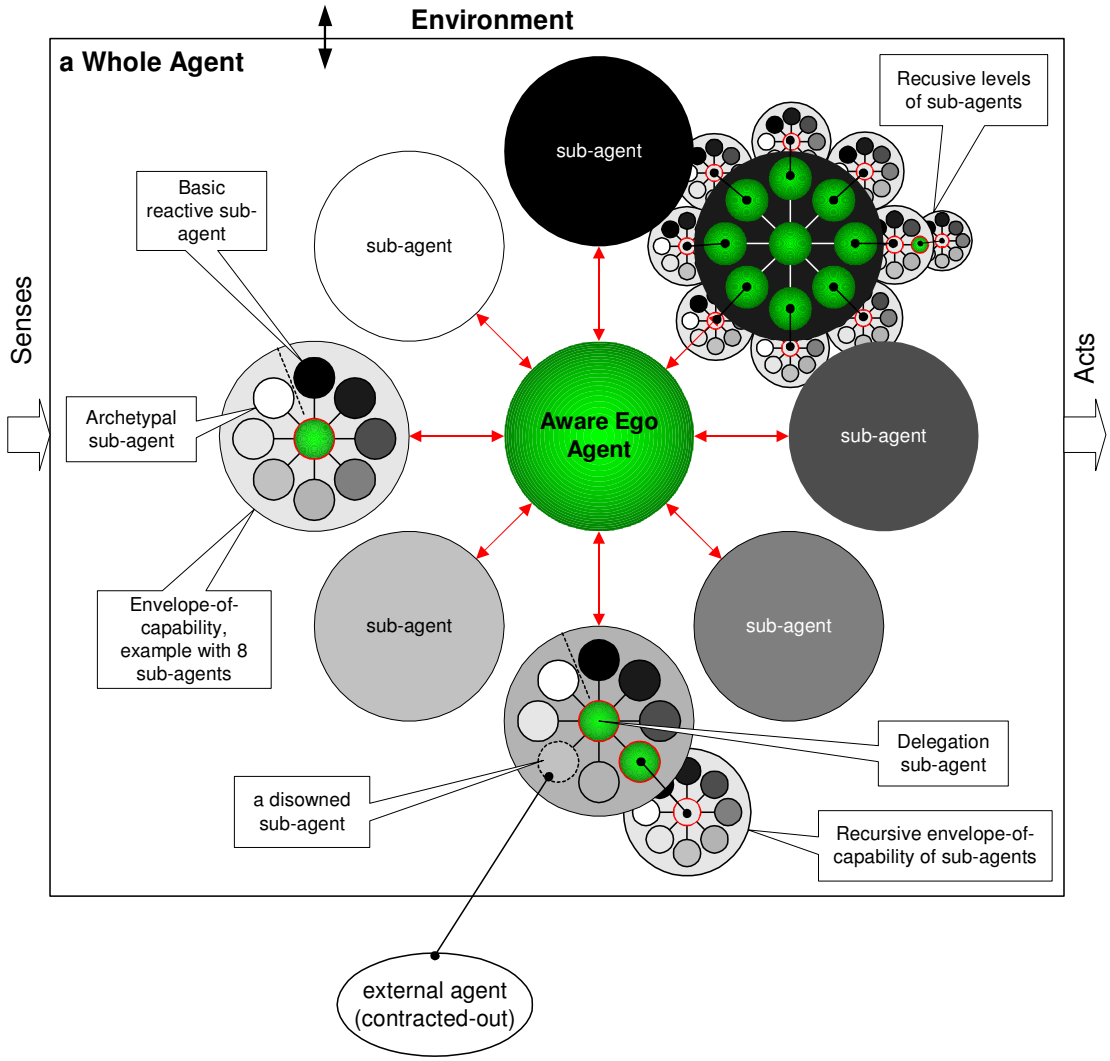

Figure 5.2 – The Shadowboard Architecture.

When a sub-agent has been found lacking in capability to achieve a specific intention, or when an external (and available) agent matches the particular specialty better than any internal sub-agents, an external agent can be called upon as if it were an internal sub-agent. This process is termed *disowning a sub-agent*.

5.3 Sub-Agents as Subselves

The mentalistic notion of a *subself* at work within the psyche of an individual, is a metaphor for the sub-agent of a Shadowboard agent. To broadly place this work in context of research upon multi-agent systems – the closest comparative field of study - most multi-agent systems (MAS) can be described as *inter*-agent systems. In contrast, the Shadowboard theory and architecture is an *intra*-agent system, one

enabling the incorporation of many components that together represent one *whole* agent, albeit a very sophisticated one. Such a *whole agent* built upon the Shadowboard architecture – e.g. the *Digital Self* – seen from without, should be seen as a fully *autonomous* individual agent, one compatible with existing definitions of agency, such as the definition adopted in this thesis as discussed in Section 2.6.

Unlike the autonomous whole agent, the inner sub-agents are semi-autonomous or even totally subservient to the *Aware Ego Agent* - the executive controller within a Shadowboard agent. Sub-agents may themselves be sophisticated agents capable of their own semi-autonomous work, or they may be Active Objects, Expert Systems, or even conventional application programs. This is a significant relaxation on the need in most MAS systems for each individual 'agent' to be a fully functioning autonomous agent. This flexibility in how agent capability is provided within a Shadowboard agent is attained by making all sub-agents ultimately subordinate to the *Aware Ego Agent*.

The inclusion of sub-agents and envelopes-of-capability of sub-agents within the Shadowboard architecture, allows it to be populated with all manner of domain specific and types of sub-agents, allowing for open-ended expansion of capability and knowledge.

Where the sub-agents are sufficiently advanced *within* Shadowboard, they will be Rule and Predicate based constraint logic solvers, in which case their beliefs include the knowledge of the environment that is most relevant to their area of specialisation, in a division-of-labour manner. In this sense, the sub-agent approach is analogous to the subself view in psychology as outlined in Chapter 4, where division of outer world knowledge and experience is divided amongst the subselves that have dealt with and continue to deal with those aspects of an individual's life experiences.

5.4 Aware Ego Agent as Aware Ego

The Aware Ego Agent is effectively the autonomous *executive director* of the whole agent, but which can hand short-term autonomy to some sub-agents. In addition to being the primary decision maker, the *Aware Ego Agent* is aware of the external environmental situation at all times, and differentiates the situations that arise. It has

full knowledge of the sub-agents available to it, and is able to call upon them individually or delegate to an envelope-of-capability of sub-agents, to negotiate challenges that the environment presents while achieving its goals. It is with respect to this knowledge of the available sub-agents and the manner it directs and delegates intentions to sub-agents, that the Aware Ego agent clearly draws upon the Aware Ego of the Psychology of Subselves.

From an implementation perspective the Aware Ego Agent detailed in Chapter 6, is an agent based on a generalised constraint logic solver, where plans are expressed in rules and predicates – as this is the most straight-forward way to modify the existing SlimWinX system into an agent system. However, the knowledge and beliefs it holds directly (within local scope) – the knowledge apart from that held by other sub-agents - are primarily about the *capabilities* it has access to via the full repertoire of sub-agents. In this respect – the capabilities representing its own self-knowledge - the Aware Ego Agent has some similarity to the Agent-0 approach covered in Section 2.6.2. But Shadowboard also has access to the superset of knowledge and beliefs held by all its sub-agents in a global scope sense.

Drawing upon the psychology further, when incorporating learning into the Shadowboard architecture, it is done at the Aware Ego Agent level, since the Psychology of Subselves tells us: that is where the *passion for improvement* should reside, in the human Aware Ego rather than in the subselves.

5.5 Envelope-of-capability, Archetype Sub-agents and Recursive Delegation

As outlined in Section 5.2 above, Shadowboard sub-agency can be clustered into an envelope-of-capability – whereby a group of like-functioned sub-agents is available to provide specialised responses within the one domain specific area. One of these sub-agents, the *delegation sub-agent* (represented by the sphere in the centre of a cluster of sub-agents in Fig. 5.2) is entirely concerned with the selection of which sub-agent in the cluster will be assigned the task, which in turn is handed to it by the Aware Ego Agent. Having made a course-grained decision on which first-level sub-agent, or which envelope-of-capability will pursue a selected intention (a specific

goal), the Aware Ego Agent delegates the finer-grained decision to the delegation sub-agent, whenever an envelope-of-capability is chosen.

The delegation sub-agent has a range of sub-agents to which it can dispense the current intention it has received from above. At one end of this range of capability, is the purely reactive *basic reactive sub-agent,* incapable of deliberation. At the other end, is the archetypal sub-agent which is the most evolved sub-agent in the envelope-of-capability, in terms of its deliberative abilities. The archetypal sub-agent will usually be the most resource intensive. (Note. The archetypal agent may be *disowned* – see next section).

A sub-agent executing an intention is committed to a plan that should achieve a specific adopted goal. This process may in turn cause other goals to be added to the Aware Ego Agent's current queue of goals. An unfulfilled intention is passed back to the *delegation sub-agent* which can either try another sub-agent in its repertoire, or pass it back up the recursive chain of delegation from where it first came. An impossible intention is dropped by the Aware Ego Agent, after being passed back to it via the *chain of delegation*.

5.6 External Agents as Disowned Selves

A delegation sub-agent, including the Aware Ego Agent itself, can have a link to an *external agent* that may be called upon to execute an intention. The reference (including the link) within the Shadowboard to the external agent may bypass an internal sub-agent which thereafter is called a *disowned sub-agent*. Effectively, the intention has been *contracted-out*. This link is analogous to the relationship that a disowned self plays within an individual human, according to the Psychology of Subselves outlined in Section 4.5.3.

Operationally, the ability for a Shadowboard agent to call upon an external agent will require the implementation of an Agent Communication Language (ACL), such as KQML or FIPA ACL, for the facility to be practicable. But more importantly to the consistency of the Shadowboard architecture, the disowned sub-agent mechanism is a psychologically founded *hook* that allows a whole-agent based on Shadowboard to collaborate within a more general external MAS infrastructure.

5.7 Believable Agents as Subpersonas

Earlier in this thesis, in Section 2.6.3.4 there is an introduction to Believable Agents and Emotional Agents. These types of agents are being developed by numerous researchers to represent human-like personas for software-based programs and agents. They typically use multimedia techniques including audio and animation, in order to help users suspend their disbelief.

Pros and cons aside regarding these forms of agents, they do have a psychological equivalent in the form of the human persona – *the personality mask developed to meet the world* – which, in the Psychology of Subselves approach to the human psyche has a multi-faceted version in the form of *multiple sub-personas*.

Adopting Believable Agents into the Shadowboard architecture as subpersona agents, allows *each* sub-agent within a Shadowboard agent to have an associated Believable Agent, in a one-to-one relationship that may be presented to the user on the screen where it is advantageous to do so. This use of persona agents is another structural consequence of basing the Shadowboard architecture on analytical psychology.

5.8 Shadowboard Architecture as an XML-DTD

The Document Type Definition (DTD) part of the eXtended Markup Language (XML) is for defining schemas. These files end with the file-extension *.dtd*. XML-DTD files are then used to direct conforming *.xml* files as to: what types of data are allowed; what they are named; in what order they must go; and how they are interrelated. The XML DTD language is useful for rendering the *structural model* of Shadowboard into *syntax* widely readable by increasing numbers of both humans and systems. Systems that can read XML files usually have an internal object model that conforms with the W3C DOM object model, as discussed in Section 3.5. The semantics of the architectural structure depicted visually in Figure 5.2 are rendered in XML-DTD in the file *Shadowboard.dtd*, shown below in Figure 5.3.

```
<!-- The Shadowboard Architecture DTD -->
<!ELEMENT whole-agent (aware-ego-agent, sub-persona?) >
<!ATTLIST whole-agent
              global-name       CDATA        #REQUIRED
              global-type       NMTOKEN      #REQUIRED
              global-id         ID           #REQUIRED  >
<!ELEMENT aware-ego-agent (sub-agent)>
<!ELEMENT sub-agent (description, sub-persona?, envelope-of-capability?,
                     external-agent?) >
<!ATTLIST sub-agent
              agent-name        CDATA        #REQUIRED
              agent-type        NMTOKEN      #REQUIRED
              agent-id          ID           #REQUIRED
              rank              CDATA        #REQUIRED
              a-delegator       (true|false) "false"
              disowned          (true|false) "false"
              external-agent_id IDREF        #REQUIRED          >
<!ELEMENT envelope-of-capability (archetype?, sub-agent+, reactive-agent?) >
<!ELEMENT archetype (sub-agent)>
<!ELEMENT reactive-agent (sub-agent)>
<!ELEMENT external-agent EMPTY>
<!ATTLIST external-agent
              agent-name        CDATA        #REQUIRED
              agent-type        NMTOKEN      #REQUIRED
              agent-id          ID           #REQUIRED
              acl               (KQML|FIPA|OTHER) "KQML"
              content-language  CDATA        #REQUIRED
              ontology          NMTOKEN      #REQUIRED  >
<!ELEMENT sub-persona EMPTY>
<!ATTLIST sub-persona
              persona-name      CDATA        #REQUIRED
              persona-type      NMTOKEN      #REQUIRED
              persona-id        ID           #REQUIRED
              icon-name         CDATA        #REQUIRED
              animation-name    CDATA        #IMPLIED
              current-status    CDATA        #IMPLIED
              options-mask      CDATA        #REQUIRED
              event-mask        CDATA        #IMPLIED
              general-expression (happy|sad|cheerfull|gloomy|fearfull|
                                  brave|determined|decisive|confident|
                                  irritated|grim|surprised|indifferent|
                                  curious|innocent|cross|nervous|guilty|
                                  arrogant|guilty|disbelieving)  "curious"  >
<!ELEMENT description (#PCDATA)>
```

Figure 5.3 – The Shadowboard XML DTD – Shadowboard.dtd.

Note: In addition to the *elements* shown in Figure 5.2, the *attributes* of those elements are also shown in Figure 5.3

The primary focus of Chapter 6 is implementing the architecture represented semantically in the Shadowboard.dtd above, and described throughout this chapter. The computational implications of the Shadowboard architecture (foreshadowed in Chapter 4) are also discussed in Chapter 6.

5.9 Declaration of a Digital Self as an XML file

It should now be clear that we can *declare* a Digital Self agent based on the Shadowboard architecture using a *.xml* file - one that conforms to the Shadowboard schema in Figure 5.3. The full declaration of a Digital Self, and how the various interests of its sub-agents are mapped to a user's existing file system, are the primary subject of Chapter 7. Nonetheless a simpler, embryonic Digital Self is present here as an .xml file in Figure 5.4 below, in order to clearly portray the distinction between the Shadowboard architecture and an implementation of an agent built upon the architecture.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE whole-agent SYSTEM "Shadowboard.dtd" >
<whole-agent global-id="IP:203.31.192.82" global-name="Deepthought"
        global-type="Shadowboard">
  <aware-ego-agent>
    <sub-agent a-delegator="true" agent-id="A1" agent-name="Brady"
        agent-type="DecisionSupport" disowned="false"
        external-agent_id="A1" rank="1">
      <description>This is the Aware Ego agent.</description>
      <envelope-of-capability>
        <archetype>
          <sub-agent a-delegator="false" agent-id="A2"
                    agent-name="InfoMaster" agent-type="PersonalAssistant"
                    disowned="true" external-agent_id="A10" rank="1">
            <description>This is the archetypal sub-agent of an
                        envelope-of-capability.
            </description>
            <sub-persona animation-name="guru.mpg" current-status="DEFAULT"
                    event-mask="128" general-expression="happy"
                    icon-name="guru.gif" options-mask="DEFAULT"
                    persona-id="A101" persona-name="Hal" persona-type="MPEG4"/>
```

```
            <external-agent acl="KQML" agent-id="A10" agent-name="TheInfoGuru"
                    agent-type="PersonalAssistant" content-language="PROLOG"
                    ontology="Informatics"/>
          </sub-agent>
        </archetype>
        <sub-agent a-delegator="false" agent-id="A3" agent-name="MailMan"
                    agent-type="PersonalAssistant" disowned="false"
                    external-agent_id="A3" rank="2">
          <description>This is an email filtering agent.</description>
        </sub-agent>
        <sub-agent a-delegator="false" agent-id="A4" agent-name="InfoMan"
                    agent-type="Information" disowned="false"
                    external-agent_id="A4" rank="3">
          <description>This is a specialized internet search agent.
          </description>
        </sub-agent>
        <reactive-agent>
          <sub-agent a-delegator="false" agent-id="A5" agent-name="Speedy"
                    agent-type="Reactive" disowned="false"
                    external-agent_id="A5" rank="99">
            <description></description>
          </sub-agent>
        </reactive-agent>
      </envelope-of-capability>
    </sub-agent>
  </aware-ego-agent>
  <sub-persona animation-name="urbane.mpg" current-status="DEFAULT"
         event-mask="128" general-expression="confident"
         icon-name="urbane.gif" options-mask="DEFAULT"
         persona-id="A100" persona-name="Clive" persona-type="MPEG4"/>
</whole-agent>
```

Figure 5.4 – Declaration of a simple Digital Self in XML – SimpleDigitalSelf.xml.

This simple Digital Self consists of the mandatory Aware Ego Agent (named *Brady*), with a visual accomplice persona agent (a Believable agent) named *Clive*. Brady has an envelope-of-capability which consists of two sub-agents: one called *MailMan*, an Information agent; the other *InfoMan*, a Personal Assistant agent. They have an archetypal sub-agent internally named InfoMaster, which has a Believable Agent as its visual persona called *Hal*, but which is *disowned* (sub-contracted out) to an external Information Agent named *TheInfoGuru*. The envelope-of-capability also includes a Reactive agent named *Speedy*.

5.10 Agent-Oriented MVC (AoMVC)

Having so far described the *structure* of a Shadowboard agent, this section is an explanation of how the Shadowboard agent system *intersects* with the GUI system of a Workstation operating system. (Note: Recall from Chapter 1 that an aim of the architecture is to give the agent system at least equal status with the GUI, within the workstation operating system – with which it has to interact.)

In the earlier background coverage of middleware (Section 2.4), as compared with client-side systems (Chapter 3), the MVC model (Section 3.2) was outlined as an excellent representation of the relationship between: on-screen views; behind-the-scene structures; and the controlling aspect of the software which manages the relationship between those views and those structures. Given that the Shadowboard architecture is a model of behind-the-scene structures and wherewithal, it can be presented to the reader, using the concepts and terminology of MVC.

By mapping the psychological model behind Shadowboard into an MVC model diagram, we arrive at a graphic representation of the Shadowboard architecture, which portrays hybrid UI/Agent architectural features (see Figure 5.5).

The subselves of psychology, which are represented by sub-agents in the software, represent the *Model* part of MVC – albeit a complex model. The Controller of MVC is akin to the *Aware Ego Agent,* the dominant sub-agent within a Shadowboard-based agent. The subpersonas in the psychology, which represent the *View* part of MVC, may be presented to the screen as Believable Agents, or alternatively as standard GUI windows in a conventional application sense.

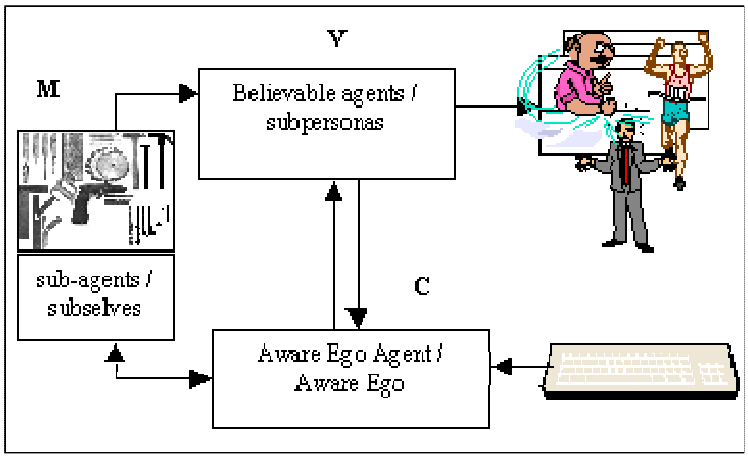

Figure 5.5 – Shadowboard Architecture portrayed as an AoMVC.

During implementation of the Digital Self, each sub-persona will initially be represented within the GUI system as having its own *window*, which is the visual foundation for a more sophisticated Believable Agent. I.e. the GUI *window*, is where the Shadowboard system *meets* the GUI system. A sub-agent that does not have an associated Believable Agent (a sub-persona agent) would typically have a conventional window interface. During configuration, the user can choose the sub-persona agent to represent a particular sub-agent, much as the user can currently select personalised wallpaper, screensaver and other visual aspects of their existing GUI systems.

5.11 Summary

In this chapter we have mapped psychological terms from the Psychology of Subselves, to like-named or like-functioned sub-agents within a whole Shadowboard agent. The architectural structure of Shadowboard, also derived from the psychology, is visually depicted in Figure 5.2, and then semantically rendered in the XML DTD file *Shadowboard.dtd* shown in Figure 5.3. To press home the difference between the Shadowboard architecture and a Digital Self agent built upon it, a simple Digital Self is declared in XML in file *SimpleDigitalSelf.xml* in Figure 5.4 – one which is heavily expanded further down in Chapter 7. Before we look at implementation issues in developing a computational system upon this agent architecture in the next Chapter (6), we mapped the Shadowboard architecture

language (terms) into the language of the MVC model – in order to help explain how the Shadowboard agent system intersects with the GUI in a conventional operating system.

6 IMPLEMENTING SHADOWBOARD UPON SLIMWINX AND XSPACES

Although SlimWinX and XSpaces were introduced in Sections 2.4.9 and 3.3, there is a need here to go into them more deeply in order to demonstrate how they will be modified to construct an implementation of the Shadowboard architecture. In particular, the next few sections address: the *windowing system* and the *event handling* system in relation to the windowing system, as it is this event-driven mechanism that will be utilised and enhanced, to *process the goals* of a Shadowboard agent.

Following on from the descriptions of existing code are details of the modifications necessary to map the *Aware Ego Agent*, the *Envelope-of-capability* and *Sub-agents*, upon existing class structures within SlimWinX. Then follows the details of how the computational system - represented by the *event-driven* system in SlimWinX - can be transformed into a *goal-driven* computational system based on Predicate Logic.

6.1 SlimWinX Class Inheritance Relationship (*Is-a*) Hierarchy

Within SlimWinX and XSpaces there are several hierarchies of organisation that interact in a complex manner, which will be exploited as a functional and tested code-base for the Shadowboard agent system. The first and most apparent, is the *Is-a* class hierarchy relationships (see Section 2.3.1.5) constructed in the underlying C++ language, which are the foundational relationships between all objects instantiated at runtime. See Figure 6.1 below: The main class, from which a new SlimWinX application is derived is *SG_Program* (top right). SG_Program multiply-inherits from three classes:

- *SG_Tree* – is the primary class of the XSpaces system;
- *SG_GPanel* – is an application's interface with the GUI window and event-driven system;

- *SG_GTOlib* – is the Graphics, Text and Object database system (GTO) which includes the Help system.

At the base of the GUI part of the main hierarchy is the *SG_Foundation1* class, which is little more than an abstraction, since all of its methods are *virtual* (abstract), apart from the constructor and the destructor. Here is the simple declaration of SG_Foundation1 in C++:

```
class SG_Foundation1 { //Base class for a hierarchy.
public:
        SG_Foundation1();
        ~SG_Foundation1();
        virtual void Show();
        virtual void Hide();
        virtual void Move(int,int,char);
        virtual void Handle_event(SG_Event * );
        virtual void Clear_event(SG_Event * );
        virtual int  Mouse_within(SG_Event *);
        virtual void Handle_dropped_upon(SG_Event * curr_event);
};
```

The next class down in the hierarchy is *SG_Pane*, which is SlimWinX's equivalent to a simple visual window upon which all other things are built including basic components such as buttons (SG_Button_Icon, SG_Button_ILabl) and hotspots (SG_Hotspot) – used to enact hyperlinks and trigger various other events.

Further down the hierarchy is *SG_Panel* (metaphorically a *panel* of simple window *panes*), which is a window that *contains* other components, particularly other components derived from SG_Panes such as buttons. Note: *SG_Panel* will be the basis for a derived Envelope-of-capability class in Shadowboard.

Each of these classes have numerous data items and methods, for example the *methods* of the *SG_Tree* class are listed further down in Figure 6.2. (Note: The classes in Figure 6.1 represent over 70,000 lines of object-oriented hand-coded C++ written by the author, prior to, but leading into the research of this thesis.)

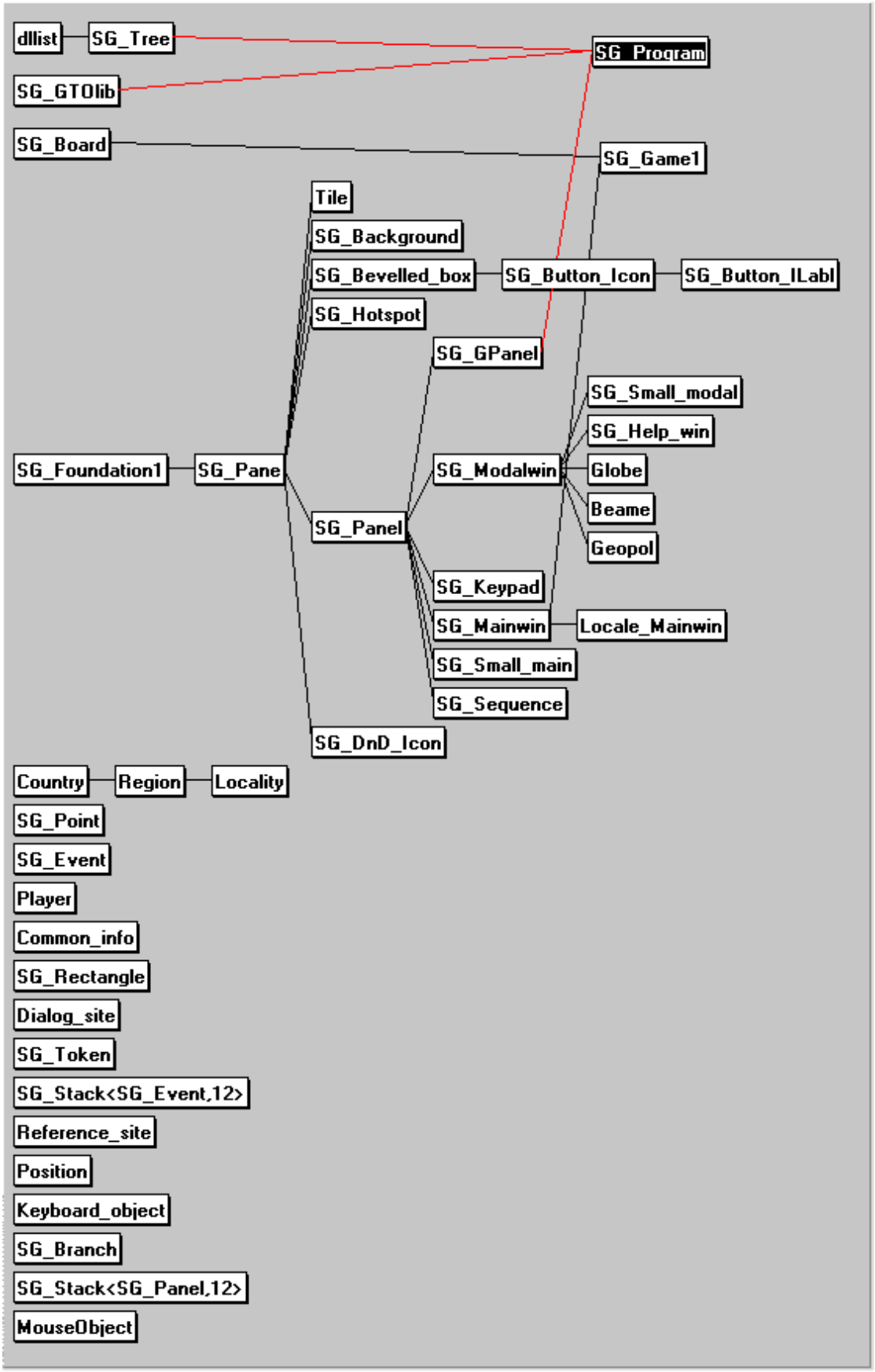

Figure 6.1 – SlimWinX Classes and Inheritance Relationships.

6.2 Containment Relationships (*Has-a*)

In addition to the class inheritance hierarchy of relationships, there are the *Has-a* relationships within classes: i.e. often the data members of an object include instantiated objects of other class types, from both within and outside of the inheritance hierarchy.

The solitary classes to the bottom of Figure 6.1 are generally involved via *Has-a* relationships. For example, classes *SG_Event*, *Keyboard_Object*, *MouseObject* are a part of the event-driven system; while *SG_Branch* is a part of the XSpaces system. Note: Other classes there encapsulate generic utilities, such as: *SG_Token* and *SG_Stack*.

The following code fragment is from the declaration file for *SG_Tree*, and holds an example of a *has-a* relationship within it. The last three lines declare three data members named, *root_token*, *parent_token* and *child_token*, which are objects of type SG_Token.

```
class SG_Tree : public dllist {

protected:
   char         hierarchy_no;   // For a series of trees. eg. C:-Z:
   char         logged;         // Tree read-in yet? TRUE or FALSE
   char         max_gen;        // Maximum generation encountered.
   char         generation;     // 'current' generation no. during recursion.
   char         graphic_built;  // TRUE/FALSE if build_branch() has run.
   long int     chars_read;     // Chars accumulated during creating SG_Branch's
                                // i.e.. a running total.
   void delete_data(void * branch);
public:
   SG_Token     root_token;     // eg. PROGRAM or MENU     or ANCESTOR
   SG_Token     parent_token;   // eg. PANEL   or POPUP    or SPOUSE
   SG_Token     child_token;    // eg. PANE    or MENUITEM or CHILD
…
```

Figure 6.2 – Code fragment from SG_Tree declaration with three *has-a* SG_Token objects.

6.3 Dynamic Object (*Create-a*) Relationships in XSpaces

In addition to traditional object-oriented *Is-a* and *Has-a* relationships, a SlimWinX application has dynamically substantiated objects, driven by the XSpaces definition file for an application. Rather than hard-coding these objects and their inter-relationships in the C++ code, they are declared and maintained within the application's XSpace .ini file. In addition to the runtime creation and destruction of these declared objects, XSpaces also enables the containment-relationships between these objects to be dynamically modified during the execution of the application program, and it allows for those modifications to be retained across different executions of the program.

For example, in Figure 6.3 below is a fragment of XSpace taken from the full *Branch-out.ini* file in Appendix A, which represents the keypad of buttons along the opening screen of the application Branch-Out, graphically depicted back in Figure 2.22. Note: That figure also depicts the dynamic nature of these objects, by showing the dragging and dropping of a 'live' (functioning) button.

```
PANEL,POffice_Keypad1,1,0,11,10,0,374,640,104,4,0
    {
        PANE,pad1_5,1,0,30,64,372,383,72,72,4,BRIEF,BRIEF1,4000,170,0
        PANE,pad1_0,1,0,30,65,7,383,72,72,4,GENIE,GENIE1,3B00,108,0
        PANE,pad1_1,1,0,30,66,80,383,72,72,4,MAILBOX,MAILBOX1,3C00,101,0
        PANE,pad1_2,1,0,30,67,153,383,72,72,4,LIBRARY,LIBRARY1,3D00,102,0
        PANE,pad1_3,1,0,30,68,226,383,72,72,4,TREE,TREE1,3E00,103,0
        PANE,pad1_4,1,0,30,69,299,383,72,72,4,TRAN,TRAN1,3F00,104,0
        PANE,pad1_6,1,0,30,70,445,383,72,72,4,GLOBE,GLOBE1,4100,106,0
        PANE,pad1_7,1,0,30,71,518,383,36,36,2,SWICH32,SWICH321,4200,100,0
        PANE,pad1_8,1,0,30,72,554,383,36,36,2,EXIT32,EXIT321,4F00,283,0
        PANE,pad1_9,1,0,30,73,518,419,36,36,2,ABOUT32,ABOUT321,4300,9100,9101
        PANE,pad1_10,1,0,30,74,554,419,36,36,2,NOTE32,NOTE321,4400,109,0
        PANE,pad1_11,1,0,30,75,591,383,40,72,4,HELP,HELP1,7800,191,0
    }
```

Figure 6.3 – Fragment of the XSpace for a keypad of buttons in application Branch-Out.

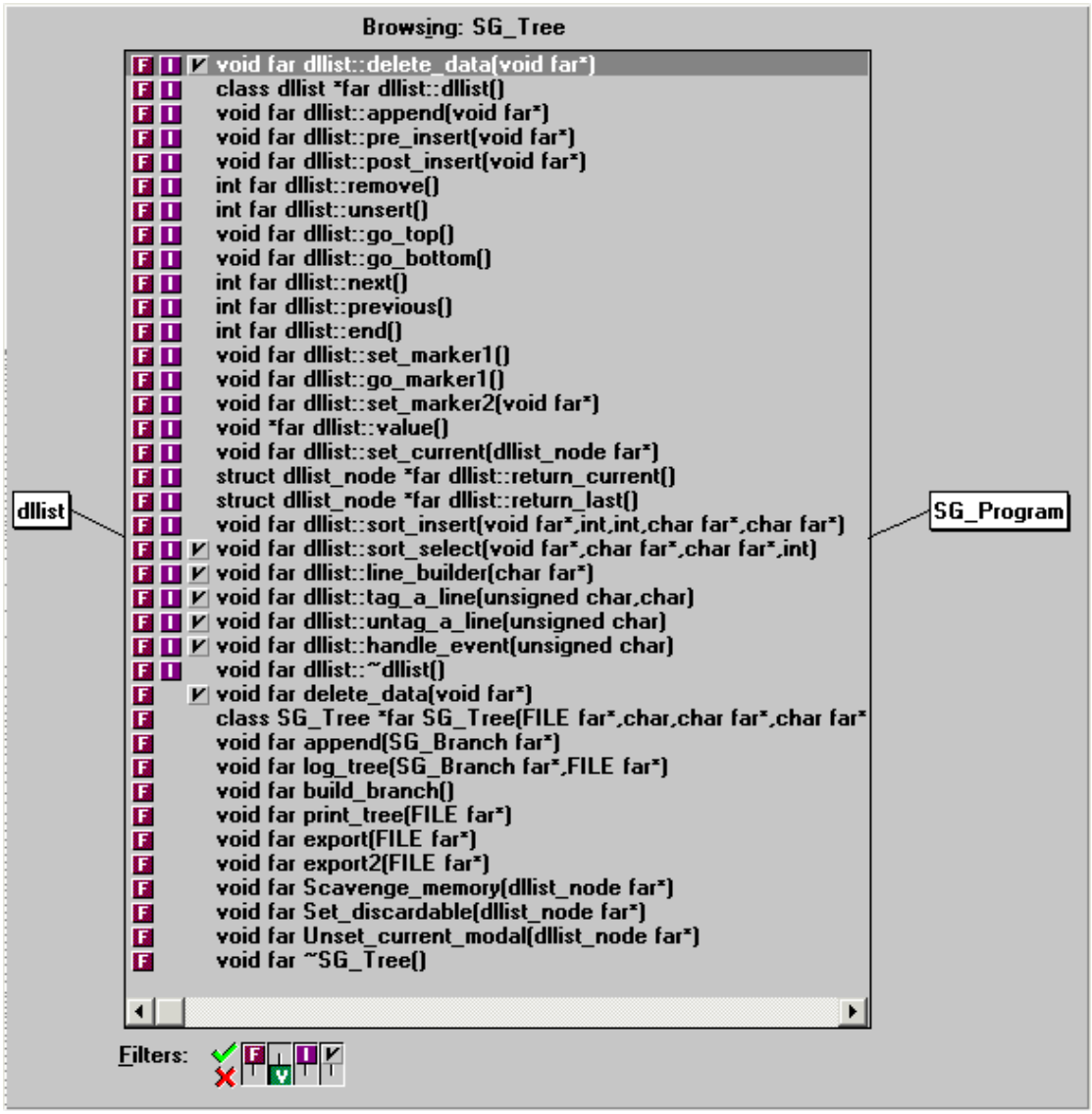

Figure 6.4 – Functions in the SG_Tree Class.

This dynamic nature of XSpaces is an essential feature for the envisaged 24x7 agent system contained in a Digital Self, whose tasks include managing client-side objects and data. Indeed, the methods within the SG_Tree and SG_Branch classes – which are the primary classes used to enact the dynamic configurability of XSpaces – have many like-named and like-functioned methods to those in the W3C DOM interfaces depicted back in Figures 2.26, 2.27 and 2.28. Note: What W3C DOM refers to as a NODE, XSpaces refers to as an SG_BRANCH.

Figure 6.4 above is a list of the methods in SG_Tree. All are marked with an 'F' indicating they are functions rather than variables (which have been filtered out of this view of the class). Those marked with an 'I' have been inherited from the underlying *dllist* (double-linked list) class. And those marked with *'V'* are instantiations of virtual/abstract classes.

6.4 SlimWinX Event Handling

Event-driven programming passes messages around the system primarily to communicate and instigate actions. In SlimWinX each receiving object has a *Handle_event* method, be it an SG_Page, SG_Panel, SG_Program or other object types. The Handle_event often calls other methods, particularly other Handle_events elsewhere in the system. If a Handle_event method is unable to service the current message, it can either pass it forward to some other object to handle, or it can pass it back to the method that called it in the first place. Another way of thinking about an event-driven system, is that it is a *polymorphic message system*, passing a singular message around until it triggers an acceptable action.

In addition to passing a message onward or backward , a Handle_event method can generate new events, which it puts into an *Event Queue* to be appropriately serviced. Eg. When using drag'n'drop within SlimWinX, to take an object from one container-object and drop it onto another, the receptive container-object generates two additional events which it puts back into the event queue, which then instigate a tidy-up process back at the container from where the object came.

As with most current-day desktop GUI systems, the main generic types of events in the SlimWinX system are *Positional Events*, *Focus Events* and *Command-message Events*:

- **Positional** events are generated by pointing devices, predominately via a Mouse.
- **Focus** events are predominantly generated via the keyboard and find their way to the window or other GUI component that currently has the so called *focus*.
- **Command-message** events are generated internally via the software, and are used to communicate various things between different elements of the system.

Programmatically it is easy to locate the intended destination of Positional events, such as a mouse button click, since the mouse cursor is literally upon the destination object when the user presses the button. The following code fragment in Figure 6.5 is the entire *Mouse_within* method implemented in SG_Pane, which determines whether the mouse was clicked within the boundaries of a given SG_Pane object.

```
int SG_Pane::Mouse_within(SG_Event * mikhail_mouse)
{
   if (pane_options & OP_SELECTABLE)
      if ((mikhail_mouse->detail.MOUSE.X > corner_coords[0] &&
           mikhail_mouse->detail.MOUSE.X < corner_coords[2]) &&
          (mikhail_mouse->detail.MOUSE.Y > corner_coords[1] &&
           mikhail_mouse->detail.MOUSE.Y < corner_coords[7]) ) return TRUE;
   return FALSE;
}
```

Figure 6.5 – Mouse_within method of the SG_Pane Class.

Then a whole panel of SG_Panes can be tested by calling all their Mouse_within methods, as is done in SG_Panel's method named *Selected*, used to return the one pane that the mouse click was upon. The complete code of *SG_Panel::Selected* is shown below in Figure 6.6.

```
//---------------------------------------------------------------------
// Finds which sub-pane [in just the next generation - not their children,
// that recursion is catered for in Panel's::Handle_event() ] the mouse
// cursor was in when a mouse event is detected but
// only if it is OP_SELECTABLE...

SG_Pane *  SG_Panel::Selected(SG_Event * mikhail_mouse)
{
    SG_Pane * current;
    SG_Branch * curr_branch;
    int  num_kids=0;
    char generation=0;

    ggparent->set_current(ego);
    curr_branch = (SG_Branch *) ggparent->value();
    num_kids = curr_branch->sibling_cnt;
```

```
    if (num_kids )
    {
        //The very next node will ALWAYS be the first child, but
        //the rest needn't necessarily follow directly (eg. grandkids)
        ggparent->next();
        curr_branch = (SG_Branch *) ggparent->value();
        generation = curr_branch->gen;
        while (!ggparent->end() && num_kids) //decrement kids as found.
        {
            curr_branch = (SG_Branch *) ggparent->value();
            if ( curr_branch->gen == generation )
            {
               num_kids--;
               if (!curr_branch->self)
                  curr_branch->Get_object(ggparent, 0);
               current = (SG_Pane *) curr_branch->self;
               if ((current->pane_options & OP_SELECTABLE))
               if (current->Mouse_within(mikhail_mouse))
               {
                  return current;
               }
            }
            ggparent->next();
        }
    }
    return NULL;
}
```

Figure 6.6 – Mouse_within method of the SG_Panel Class.

However, for non-positional events such as keystrokes at the keyboard, the concept of *focus* is an efficient method for them to find their targets to take appropriate or required action. Modal windows employ *focus* to capture all keystrokes and other non-positional events, giving the current modal window first go at handling events.

In the case of an event not being able to be handled by the window with the focus, the system is interested in the sequence of method calls that led to it, so that it can back-track effectively. The event-driven system in SlimWinX employs a *focus-chain* technique to do this as follows: as processor execution weaves a path through the sequence of methods representing the user's decision path through the application, the event system marks this focus-chain by setting and unsetting various *flags* - data values individually held in the accessed components themselves. The flags are

bitmap values, represented by constants in the C++ language, so that numerous options can be put into the one variable and masked out as needed.

There are two sets of flags in SlimWinX:

- OP or *option flags*, set once only at the start of program runtime;
- CS or *current system flags*, which can be modified during the life of the program.

Table 6.1 below describes the OP flags, while Table 5.2 describes the CS flags.

C++ Constant name	**Bit mask (in octal)**	**Brief Description of Usage**
OP_SELECTABLE	0x01	Component is selectable by user.
OP_SHADOW	0x02	Component window has a shadow.
OP_MOVEABLE	0x04	Whether or not to save covered screen.
OP_RESIZEABLE	0x08	A resizable component.
OP_PASS_ON_CLICK	0x10	To pass on event that was handled.
OP_INVISIBLE	0x20	For hotspots and other tricks.
OP_DRAGABLE	0x40	Draggable beyond own window.

Table 6.1 - Option flags and their mask values.

The following CS flags are used to maintain the focus-chain and control the order in which objects get a chance to handle the current event:

CS_SELECTED: The focus-chain, which may run through numerous recursive panels (i.e. one panel may have a pane which is itself a panel, and so on), is marked by the CS_SELECTED flag, set in all those panes along the focus-chain. Only one pane per panel will have CS_SELECTED set.

CS_FOCUSED: At the end of the focus-chain is the currently active pane. This pane will receive all focus events, such as keyboard keystrokes. Only one pane can have the focus at any time, and it is always at the end of the focus-chain. I.e. It will have both CS_SELECTED and CS_FOCUSED set.

CS_PRIORITY: A pane has this flag set if it is to get a chance at handling the current event before the focused parent panel does. The usefulness of this flag is best described with an example: in the case of a Keypad of buttons, such as that represented earlier in Figure 6.3, we want the appropriate button to respond to an event (for example an accelerator key). However we do not want a single button to grab the focus, but instead want all the buttons in the keypad to have equal status in relation to incoming events. To achieve this the parent panel (the Keypad) retains the focus, while all the child buttons have their CS_PRIORITY flag set, telling the system to give them each a chance at handling the event, before the parent panel gets a chance to handle it.

C++ Constant name	Bit mask (in hex)	Brief Description of Usage
CS_VISIBLE	0x0002	Show and Hide methods affect it.
CS_CURSORVIS	0x0002	Cursor visibility over the component.
CS_DISABLED	0x0004	To ignore all events.
CS_SELECTED	0x0008	Set if it is a component in current panel
CS_FOCUSED	0x0010	Currently has the focus.
CS_MOVING	0x0020	Is being moved.
CS_RESIZING	0x0040	Is being resized.
CS_MODAL	0x0080	Is current modal pane/panel.
CS_ACTIVE	0x0100	Active is focused or child of focused.
CS_PRIORITY	0x0200	If child is to have priority at handling Events before focused parent.

Table 6.2 - Current Status flags and their mask values.

Similar to the Option mask and the Current Status mask, every component in the SlimWinX system can also have *event masks* set, to indicate which types of event the component should attempt to handle. The event masks are detailed in Table 6.3 below. The value 0xFFFF causes the component to handle all event types that come its way, while 0 causes it not to attempt any.

C++ Constant name	Bit mask (in hex)	Brief Description of Usage
EV_Idle	0x0000	Ignore all events.
EV_MouseDown	0x0001	Mouse button down event.
EV_LmouseDown	0x0001	Left mouse button down event.
EV_RmouseDown	0x0002	Right mouse button down event.
EV_MouseStillDown	0x0004	Mouse button still down event.
EV_LmouseStillDown	0x0004	Left mouse button still down event.
EV_RmouseStillDown	0x0008	Right mouse button still down event.
EV_MouseUp	0x0010	Mouse button up event.
EV_LMouseUp	0x0010	Left mouse button up event.
EV_RmouseUp	0x0020	Right mouse button up event.
EV_KeyDown	0x0100	Keyboard key pressed event.
TRANSFER	0x0200	Transfer objects via drag'n'drop event.
GENERATE_EVENT	0x0400	Create an event.
GENERATE_2EVENTS EV_Command	0x0800	Create two events.
EV_Broadcast	0x1000	Attempt command message event.
EV_Mouse	0x2000	Attempt broadcast events.
EV_Keyboard	0x00FF	Attempt all mouse events.
EV_Message	0x0100	Attempt keyboard event.
EV_All	0xF000	Attempt message b/n components.
	0xFFFF	Attempt all events.

Table 6.3 - Event Codes and their mask values.

6.4.1 The Event Object

An event in SlimWinX is itself an object (the *Event* object), which is passed around between other objects. It has a number of fields, so it can carry significant information apart from just the event type.

```
//=========================================================================
// The Event structure...
typedef  struct mikhail
{
      unsigned X;
      unsigned Y;
} fred1;
```

```
typedef  struct idle {} fred2;
typedef struct parts
{
      char char_code;
      char scan_code;
} fred3;
typedef  union kboard
{
      int keycode;
      parts part;
};
typedef union information
{
      char      char_info;
      int       int_info;
      unsigned  unsigned_info;
      long      long_info;
      void *    pointer_info;
};
typedef  struct messages
{
      unsigned    command;
      information info;
} fred4;
typedef union all_details
{
  idle     IDLE;
  mikhail  MOUSE;
  kboard   KEYBOARD;
  messages MESSAGE;
};
//=============================================================================
// The Event class...
class SG_Event
{
public:
    unsigned    event_type;
    all_details detail;
    SG_Event();
    void Clear_event();
};
//=============================================================================
```

Figure 6.7 – The SG_Event class and the typedef union *all_details*.

SG_Event has two data members: *event_type* which is simply an unsigned integer; while the second is a complex union of structures called *details*. Depending on the event_type, all_details holds appropriate supplementary details that can be gainfully used by the Handle_event methods:

Event-type	**details**
Mouse	holds the relevant mouse cursor coordinates X and Y.

Keyboard	holds keycode or char_code and the scan_code (shift key etc)
Command-message	holds either a char, integer, unsigned, long or even a pointer variable.

Back in Figure 6.5 was an example, a reference to the *details* within a mouse generated event, which looked like this:

```
mikhail_mouse->detail.MOUSE.X > corner_coords[0]
```

and elsewhere in the SlimWinX code are examples of different references to the all_details field, such as within this code fragment which tests for either the 'Q' key being pressed, or the 'ESC' key:

```
if ( (current_sfdstate & CS_MODAL) &&

            ((curr_event->detail.KEYBOARD.keycode == KB_Esc) ||

            (toupper(curr_event->detail.KEYBOARD.part.char_code) == 'Q')) )
```

or this fragment testing for a particular Command-message in the current event:

```
if (curr_event->event_type & EV_Message)
{
    if (curr_event->detail.MESSAGE.command == RETURN_ROOT)
    {
        // Now cause Execute_event for this Panel to
        // finish-up and return to previous routine...
        //
        if (curr_event->detail.MESSAGE.info.char_info==2)
             finish_modal = 2; // Don't display root::Show()
        else
             finish_modal = 1; // DO display root::Show()
        Clear_event(curr_event);
    }
    else  ggparent->SG_WindowProc(curr_event);
}
```

or this one using a pointer variable transported via an Event object, in a drag'n'drop operation:

```
curr_Pane =  (SG_Pane *) curr_event->detail.MESSAGE.info.pointer_info;
```

6.4.2 The GetEvent / PutEvent duo

The main class SG_Program maintains an event queue and has two methods which put and take events from the queue. They are the GetEvent and PutEvent methods. The following code fragment in Figure 6.8 is the whole implementation of GetEvent:

```
void SG_Program::Get_event(SG_Event * curr_event)
{
   char current_count;
   current_count = 0;
   current_count = event_queue.stack_count(); // How many is in the queue.

   if (current_count) // Then there is something on the Stack waiting...
   {
      *curr_event = *event_queue.pop(); //copy the 'value' across.
   }
   else
   {
      // Get mouse event next...
      Mouse.Event(curr_event);
      //Now clear the mouses highbyte ...
      curr_event->event_type = curr_event->event_type & 0x00FF;
      if (curr_event->event_type == EV_Idle)
      {
         // Well now try the keyboard for an event...
         Keyboard.Event(curr_event);
         // If its still idle go and do something constructive...
         if (curr_event->event_type == EV_Idle) Idle();
         // i.e.. The application overrides the Idle() method
         // with processes that can use the excess processor time.
      }
   }
}
```

Figure 6.8 – The Get_event method in Class SG_Program.

The following code fragment in Figure 6.9 is the whole implementation of PutEvent within the SG_Program class:


```
void SG_Program::Put_event(SG_Event * generated_event)
{
    char current_count;
    current_count = event_queue.stack_count(); // How many are in there.
    // Stick this new event value into a vacant address in pending_array...
    pending_array[current_count] = * generated_event;
    // Now place it's address on the Stack...
    event_queue.push(&pending_array[current_count]);
    Clear_event(generated_event);
}
```

Figure 6.9 – The Put_event method in Class SG_Program.

6.4.3 Overloading Handle_event Methods

The Handle_event methods of the myriad of objects in the system is where the actions happen, but it only happens after the event processing system locates the appropriate Handle_event for the current event, for a given situation. For objects that have several levels of inheritance, there are usually several levels of Handle_event available to the object, as each time Handle_event is specified for a newly derived class, it overloads the method. For example, SG_Pane has a Handle event (see Figure 6.10 below), which is overloaded by a method of the same name in SG_Panel.

```
//---------------------------------------------------------------------
// This can be called by Handle_events of derived classes to grab the
// focus from the parent if it needs to...
void SG_Pane::Handle_event(SG_Event * curr_event)
{
   SG_Panel * parent_win;
   SG_Branch * curr_branch;

   ggparent->set_current(ego);
   curr_branch = (SG_Branch *) ggparent->value();
   if (!curr_branch->parent->self)
      curr_branch->parent->Get_object(ggparent, 0);
   parent_win =  (SG_Panel *) curr_branch->parent->self;
   ggparent->set_current(ego);

   if (curr_event->event_type &  EV_MouseDown)
   {
      if ((pane_options & OP_SELECTABLE) &&
```

```
            !(current_state & CS_SELECTED) &&   //i.e.. not already selected
            !(current_state & CS_DISABLED))     //i.e.. and not disabled
         {
            if (!parent_win || (parent_win->current_state & CS_SELECTED) )
            {
               // parent must be cs_selected for this pane to have its
               // CS_SELECTED flag set. i.e.. maintenance of the 'focus-chain'.
               // Then "select" THIS pane!

               Set_CS_Flag(CS_SELECTED,TRUE);

               // Also grab the focus if you can...

               if (parent_win->current_state & CS_FOCUSED)
                  Set_CS_Flag(CS_FOCUSED,TRUE);

               // If the first mouse click is just supposed to
               // select the Pane and do nothing else it must NOT have
               // OP_PASS_ON_CLICK set.

               if (!(pane_options & OP_PASS_ON_CLICK)) Clear_event(curr_event);
            }
         }
         // If it is already CS_SELECTED, then the Handle_event of
         // a derived class will need to handle the event.
      }
   }
```

Figure 6.10 – The Handle_event method in Class SG_Pane.

Similarly, SG_Program, which inherits all the methods in SG_Panel, also overloads the underlying Handle_event methods. However, such a derived class can still access the overloaded version/s of a method if it is needed. For example, the SG_Program Handle_event displayed below in Figure 6.11 is a small bit of code that only processes the SG_Quit command-message, as it passes on all of the events that come its way, to the overridden SG_Panel::Handle_event, which is a much more complex piece of code, running to several pages of source-code listing. Note: It only services the SG_Quit message, after the earlier call to SG_Panel::Handle_event has failed to do so.


```
void SG_Program::Handle_event(SG_Event * curr_event)
{
   SG_Panel::Handle_event(curr_event);
   if (curr_event->event_type == EV_Command)
   {
      if (curr_event->detail.MESSAGE.command == SG_Quit)
      {
         finish_modal = SG_Quit;
         Clear_event(curr_event);
      }
   }
}
```

Figure 6.11 – The Handle_event method in Class SG_Program.

6.4.4 Panel Processing and accessing the Central Event-queue

The SG_Panel class is a particularly large and complex class, as can partially be appreciated from the long list of methods it has listed in Figure 6.13 below. It is a *panel* in the sense that it contains a number of child *window* panes and other components. Any of those individual components may itself be another panel, and so on recursively. Any Panel may be the staging-post for a sequence of new *events*. E.g. It may hold a keypad of buttons which, when individually pressed, generate all sorts of events that are never received by panels further back in the execution stack of the program.

In addition, an SG_Panel may be a *modal* window or dialog-box, which generally means that it stops the user from interacting with any other window currently open in the application, unless the current window gives up its modality. All keyboard, mouse and command-message events are directed to the modal panel first – but recall from Section 3.3, that SlimWinX panels have a *semi*-modal functionality which enables drag'n'drop operations outside of the current modal window. Only one panel can be modal at a time. Its parent (if it has one) will usually have been modal before it takes over, hence it *disables* its parent from getting events until the user quits the panel (eg. usually via an OK, ESC or CANCEL message in most applications). Consequently, SG_Panel has an *Execute_panel* method to first get and then handle the

events generated forward of it, in the execution stack. Execute_panel calls *Get_event* and *Handle_event* in an alternating manner within a loop, as clearly seen in the source code listing of its implementation in Figure 6.12 below.

```
int SG_Panel::Execute_panel()
{
   SG_Event *current_event;
   current_event = new(SG_Event);
   finish_modal = FALSE;
   Clear_event(current_event);
   do
   {
      ggparent->Get_event(current_event);
      Handle_event(current_event);
   }
   while (!finish_modal);
   delete current_event;
   return finish_modal; // Either 1 to show root::show(), or 2 not to.
}
```

Figure 6.12 – The Execute_panel method in Class SG_Panel.

However, all references to Get_event in the system get passed up to SG_Program's Get_event (listed previously in Figure 6.8) via its address in the pointer *ggparent*, which has the consequence that events are centrally gotten in the same place, in a like-manner. The Get_event calls are *passed down* via the *ggparent_win* pointer (the memory address of the application, itself derived from SG_Program). Similarly, all *Put_event* calls are also passed down to the PutEvent in SG_Program.

Individual SlimWinX components can generate *command-message* events via their HandleEvent function via a *Put_event* method, which are also passed back to the central Put_event in SG_Program. E.g. If the user clicks the mouse on the Quit button (equivalent to Alt+F4 in most Microsoft Windows applications), that button's Handle_event receives the mouse message which in turn generates an SG_Quit command-message, which a call to Put_event places on the pending command-message queue. The SG_Quit command is processed in SG_Program's Handle_event, as can clearly be seen in Figure 6.11.

What has been discussed extensively here throughout Section 6.4 is the *generic* event-handling process in SlimWinX. The application specific actions, such as 'Animate-Fred' or 'Spin-the-globe' are serviced by a large Handle_event method in the *application specific* code itself, which overloads the Handle_event in the SG_Program class. The application's Handle_event references the overloaded Handle_event methods in much the same manner as SG_Program::Handle_event does, as needed.

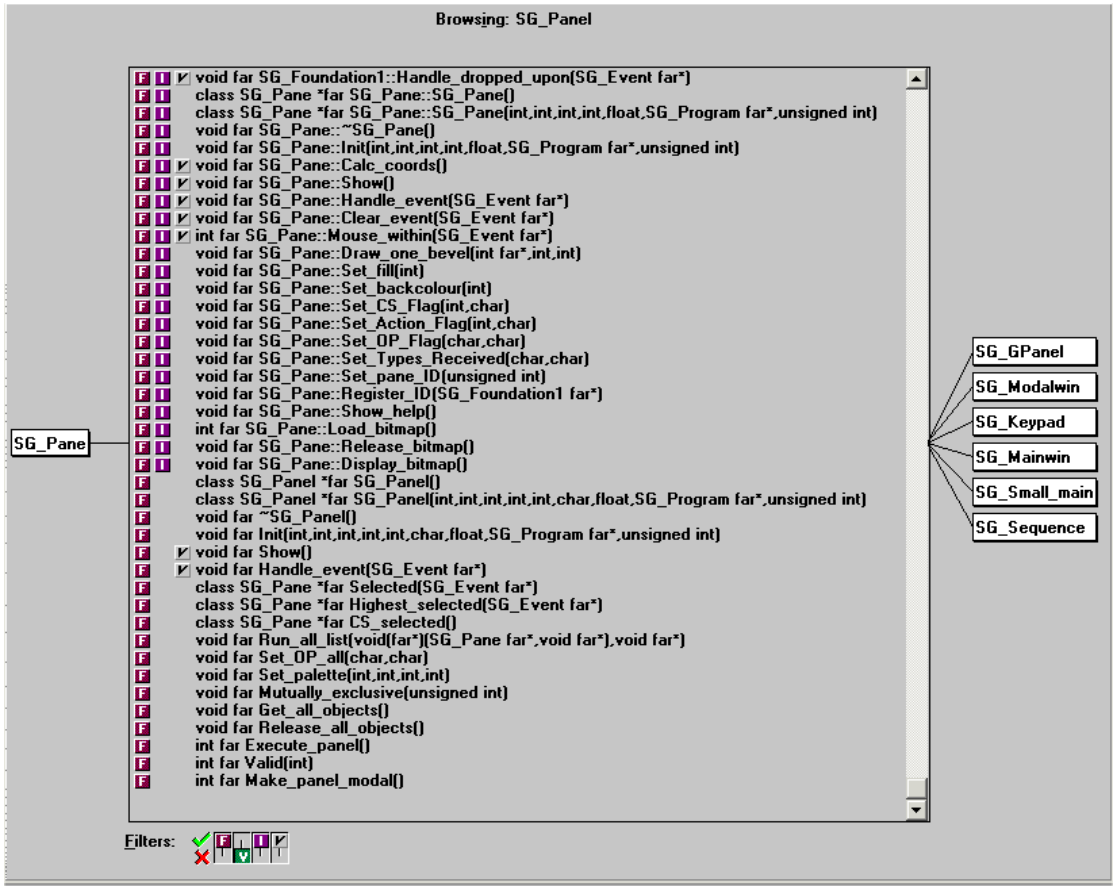

Figure 6.13 – List of the methods in Class SG_Panel.

6.5 Modifications to SlimWinX to enact the Shadowboard Architecture

We are now in a position where enough has been said about SlimWinX, to allow a meaningful explanation of how it will be modified from an event-driven, recursive panel GUI system, into a goal-driven, Shadowboard Agent system with recursively deep envelopes-of-capability. The main components within Shadowboard that are addressed here are the *Aware Ego Agent*; the *Envelope-of-capability*; and the *Sub-agent*. (Note: Most of the other entities – Elements, in XML terminology – are based on the sub-agent). In addition, the parallel between an event-driven system and a goal-driven system is exploited, as are the novel aspects of SlimWinX with a view to agent-oriented features.

6.5.1 Shadowboard Plan Selection via Generalised Logic Constraint Solver

The first aspect of *Shadowboard* that requires something more than the underlying object-oriented code of SlimWinX is that it is logic-language driven: the *Goals* and the *Facts* are expressed as *terms* in Predicate Logic, as described in Section 2.5. The Queries and Facts passed between sub-agents will be terms, either simple terms or *compound terms* such as *predicates*. E.g. Referring back to the examples in Section 2.5 **`destination(Place, Cost, Duration)`** is a predicate with three variables, as each start with a capital letter.

As foreshadowed in Section 2.5, the *plan* selection and the computation in satisfying *Intentions* (Goals selected for execution) in the Shadowboard system, are based on a Generalised Constraint Solver [50] – a constraint solver at the Aware Ego Agent level, that calls upon multiple constraint solvers down at each sub-agent class level, to best handle domain specific problems.

For example, taking the Constraint Logic Program called *Journey* in Figure 2.16, and carving that up into Shadowboard system specifics:

The problems to the left, are Goals expressed as Queries, and put to the Aware Ego Agent in the form to the right:

Problem	***Query put to the Journey program***
Which people could go to Brisbane?	`journey(brisbane, Duration, Cost, Desire, Person)?`
What places could Jill travel to?	`journey(Place, Duration, Cost, Desire, jill)?`
What does it cost to go to Canberra?	`destination(canberra, Cost, Duration)?`
How much time does John have to travel?	`available_time(john, Time)?`

The user will set the initial goals in their Digital Self, and may add to and subtract from them over time. Some goals will be set as '*perpetual*' in the system, such as: *Buy Shares Now?* The system may advise '*Buy Shares Now.*' as a Fact for the user's attention, off-and-on over a period of time as market conditions change, monitored by say, a StockAdviser sub-agent (see Section 6.5 below, for an explanation of this *data-derived reasoning*).

The *Rules* such as the one below, are held in a *delegation* agent within an envelope-of-capability, such as the Aware Ego Agent itself, or further down the hierarchy (depending on the specific rule):

```
journey(Place, Duration, Cost, Desire, Person)
                        ← destination(Place, Cost, Duration),
                          money(Person) >= Cost,
                          available_time(Person) >= Duration,
                          desire_to_travel(Person, Desire),
                          Desire = yes.
```

The individual predicates and constraints within the body of the rule, are passed by the delegation agent to appropriate sub-agents, which it knows as candidates in solving these sub-goals – by previously designating goal-types to sub-agents. I.e. The predicate `destination(Place,Cost,Duration)` may be passed to a *Travel sub-agent*, the

constraint `available_time(Person)>=Duration` may be passed to a *Diary sub-agent*, while the `money(Person)>=Cost` constraint may be passed to a *Financial-Advisor sub-agent*.

In addition, multiple rules for a single predicate gives the system non-deterministic behaviour – the choice of rule being made by the delegation agent.

6.5.2 Modeling Goal-driven processing upon Event-driven processing

While the events in SlimWinX emanate primarily from the user - namely the mouse and the keyboard – Shadowboard must deal with *terms* that are placed there either by the user via a Goal queue interface, or by other sub-agents. However, the way that events are able to be passed up and down the focus chain is an exploitable technology by Shadowboard, one already in place, transparent and debugged in SlimWinX. In particular, the way that SlimWinX handles Command-message events (so-called Logical Events in the DOM Event model), is an appropriate mechanism to handle Goals.

Where SlimWinX has the Event Object, the Shadowboard system has a Term Object, able to handle a variety of formats, as the Event Object currently does.

The Event Queue will be mirrored with a Goal Queue, but with a much larger capacity than the SlimWinX event queue (which currently only caters for situations where the user overloads SlimWinX with more clicks or keystrokes than the processor can handle - which is not often and not many).

The goal-types accepted are one of two primary types only, either: *Predicate* or *Constraint*. These compare with, say the EV_Keyboard and EV_Command of the event system within SlimWinX. Predicates such as: `journey(Place, Duration, Cost, Desire, Person); destination(Place, Cost, Duration) and available_time(Person, Time)` – are passed in the goal message. I.e. These predicates *are* goal types, and as the sub-agents are added to a system, the types of goals/queries they handle are registered with them, and within the envelope-of-capability they reside within.

6.5.3 Constraint Solving to Satisfy Goals

To answer the first Query above, *destination* will be called first as it is the only one with 'Place' as a variable, which will be substituted by the individual 'brisbane'. Down in the Travel sub-agent the anti-tuple `destination(brisbane, ?, ?)` will be put to XSpaces to return 'individuals' for both Cost and Duration. If there are multiple individuals, the other sub-agents will have to be called multiple times e.g. the `money(Person)>=Cost` constraint will be passed to a *Financial-Advisor sub-agent* as many times as necessary, and so on.

For the second problem `money(jill) >= Cost` would be called first, to get a ground term on 'Cost'.

Therefore, the goal queue processing needs to:

- Change the order of sub-goals called, depending on which parameter is 'ground'.
- Hold some sub-goals, waiting for a returned ground term. (This could be feed to the required process, via a leaf sub-agent which is sensing outer-world data).
- Call some sub-goal numerous times, depending on the number of ground terms of a kind – this will get exponential and needs to be limited by: 'How many solutions do we want? – one may be enough in many cases. This would be enabled by a 'Context condition'.
- When a lower level sub-goal has failed, it needs to step back one step at a time, giving each envelope-of-capability a chance to try another sub-goal – this is backtracking and is typical of Goal-driven processing.

Those Predicates with 'individuals' terms (terms which are 'ground' – in lowercase) need to be identified, as they will be used to select which sub-goals on the right-hand side of the rule, to be called next, as in which *order*. The order of selection of rules, is an important issue. Calls to some should be made *after* calls to others. This rule processing and the optimisation of search functionality, is the domain of *constraint solvers*.

In Constraint Programming rules are used repeatedly to replace initial constraints, until only primative constraints are left: which are passed back as the answer to the goal. This reduction process is called a *derivation*. Execution is basically a process of replacement.

In Constraint Programming there are also guidelines for controlling search, via *rule ordering* and *literal ordering* – see Chapter 7 [50]:

Rule Ordering:

- Place non-recursive rules before recursive rules.
- Place rules which are more likely to lead to answers before others (to the left).

Literal Ordering (much more subtle):

- Place a constraint at the earliest point in which it could cause failure.
- Place a constraint that cannot fail, rightmost in a conjunctive.
- Place deterministic literals before others.

A constraint solver algorithm can be adopted to accomplish a number of these guidelines. Others, such as the *'more likely to lead to answers'* rule, can be accomplished by employing performance monitoring functions on the solving of past goals, using the persistence of XSpaces to tallying results over different tasks and time periods.

6.5.4 Mapping the Aware Ego Agent upon the SG_Program Class

The Aware Ego Agent is the dominant agent in the Shadowboard system, so it is logical to base it on the SG_Program, the dominant object in the SlimWinX system. As outlined in the previous section, the most obvious addition needed to SG_Program is a Constraint Logic Solver, one based on a tree constraint solver algorithm.

Just as SG_Program inherits much capability from SG_Panel, the Aware Ego Agent will inherit much capability from the Event-of-capability, discussed further down.

6.5.5 Mapping the Envelope-of-capability upon the SG_Panel Class

Both SG_Panel and an Envelope-of-capability, are recursively deep container structures, so it is an obvious strategy to base the Envelope-of-capability upon SG_Panel. Both also have delegation responsibility with respect to their child components.

The Execute_panel method in SG_Panel is a good starting point for building a delegation method in the Envelope-of-capability, however, a SG_Panel simply tries to handle an event by passing it to all children in succession, in the equivalent of a *total-solution-space* approach in a constraint solver. Conversely, in a Shadowboard Envelope-of-capability, there is a *delegation sub-agent* which has the ability to delegate in an ordered sequence that may be varied, based on a ranking system (sub-agents within have a default 'rank' within the Envelope-of-capability, which may be modified over time, based on past performances). A delegation sub-agent needs to maintain a tally of solutions that come back from each of the sub-agents they oversee, in order to modify the ranks.

Another divergence in functionality is with regard to modality: where the SG_Panel object employs *Execute_panel* to implement a modal window, Shadowboard needs its sub-agents to run in parallel. While SlimWinX isn't currently a system that uses an inherent multi-threaded capability in the underlying C++ language, it does have a pseudo-parallel capability via a built-in time-slicing mechanism, which it uses to run numerous Active Objects (see Section 2.3.2), such as several animations on the screen at the one time. It does this via the Idle cycle in the event queue. If the event object is currently EV_Idle, then Handle_event calls a method named **Idle()** (see Figure 6.8), which is a virtual method usually overloaded by a given SlimWinX application. These periods of idleness from the event system, can be time-sliced amongst the various sub-agents that still have tasks to do, including those delegation agents, which still have Terms in their Goal Queues. The timeslicing can be regulated by a

queue of references to non-empty queues – a queue of queues. Note: This parallelism is all made possible by the fact that SlimWinX objects are instantiated (and controlled thereafter) via the XSpace system at the beginning of runtime and maintained there, rather than via the execution sequence of nested methods, such as those producing the focus-chain.

Clearly, from the above discussion, another modification required of the SG_Panel to an Envelope-of-capability, is that they need to have their own queues, rather than accessing a single queue back in the main object. A reason for this queue, is that a sub-agent may pass an unsatisfied goal back to the delegating agent, for a re-delegation. A second, more substantial reason for the queue is that sub-agents are able to put new terms upon the queue back in the Aware Ego Agent, some of which may get delegated down to the same Envelope-of-capability. I.e. The Execute_panel in the Envelope-of-opportunity object within Shadowboard, gets terms from its own queue rather than from the central queue, however they will often have originated from the central queue in the first place. These extra queues are a straight-forward modification, as the current event-queue in SlimWinX is simply an object based on the SG_Stack class, so the system can implement any number of such queue objects.

There is also a need to be able to *quit* all currently executing goals and sub-goals that are no longer useful. i.e. If a predicate further back in the Envelope-of-capability has been pulled (time-out, whatever), the system needs to signal all processing forward of it, to also quit. In multiagents systems, *context conditions* are used to discontinue running tasks. A context condition is just another *constraint* – in our case one that is broadcast as a top-down *quit*.

6.5.6 Mapping the Sub-agent upon the SG_Pane Class

Both SG_Pane and Sub-agent are the child-nodes in recursively deep, containers of components. Both are called upon to handle some event: answer a predicate-based query, or offer a solution to a simplified constraint (with perhaps a primative constraint) in the case of most Shadowboard sub-agents. Both are able to put things back into the central queue in the dominant object: via Put_event in SlimWinX; and a similarly implemented Put_terms in Shadowboard.

However, event-wise, Sub-agents are like Active Objects perpetually feeding equally upon the event cycle, sharing EV_Idle processor time as discussed above in Section 6.5.5. Just as Envelope-of-capability objects need to run in parallel, so too do the Sub-agents - not just to help solve the current Goals being directed by the Aware Ego Agent, but to also *notify* or recommend to the user, newly arising situations from their outer environment (often some part of the Internet). E.g. The email Sub-agent will check email records regularly and 'put' a predicate into the main queue which will be picked up by the Aware Ego Agent – in AI this is termed *data-derived reasoning*. These sorts of derived Goals can be put to the user as propositions. i.e. Drawing upon the earlier constraint logic example: *'You could now travel to Brisbane.'* may emanate from the Travel sub-agent. Such *propositions* will most often be driven by a perpetual user Goal, as discussed above in 6.5.1.

When a new ground tuple arrives from the outerworld down at the sub-agent level, the sub-agent has a number of things to do:

1. Store the new information as a tuple in its tuplespace store.

2. Process any of its predicates that represent perpetual goals, which use tuples of the form of the one that has just arrived.

3. Notify those goal-queue processing agents that are currently operating, which may benefit from the new tuple(s).

To accomplish this third task, it needs the list of all envelope-of-capabilities+delegation agents, which currently have goal queues, but which have already processed the sub-agent's tuplestore in its old state. I.e. The new information in the store may help achieve a goal that has not yet been achieved.

The sub-agent maintains just such a list of all delegation agents, which are still processing (the delegation agents have to help in the maintenance of this list, by removing themselves, when they are done with all sub-agents in their reach). They do so by recording the task-no (as assigned to a primary goal via the Aware Ego Agent), together with the 'self' reference of each agent that is in the chain, between the Aware Ego agent and this particular sub-agent – ie. the task-no concatenated with a list of 'self' references.

6.5.7 Putting Distributed Predicates into filename.GTO Records

One of the primary tasks of most Shadowboard sub-agents in a Digital Self, is to monitor some aspect of the external environment of interest to the user, and translate information from that environment into *ground terms* expressed in Predicate Logic, as fodder for the built-in constraint solver. Therefore, in addition to queues, these predicate-driven sub-agents will usually need to store large numbers of *ground* (individual) terms very often sourced from the Internet.

For example, calling upon the logic programming discussed in Section 6.5.1 above and expressed as ground terms in the Journey program back in Figure 2.16, a *Travel Sub-agent* would store these predicates loaded with ground terms:

```
destination(brisbane, 800, 5).
destination(sydney, 500, 3).
destination(perth, 1000, 6).
destination(canberra, 300, 2).
destination(hobart, 350, 3).
destination(adelaide, 600, 4).
```

a *Financial-Advisor sub-agent* would store these:

```
money(john, 1100).
money(jed, 700).
money(jill,400).
money(june, 800).
```

a *Diary sub-agent* would store these:

```
available_time(john, 2).
available_time(jed, 6).
available_time(jill, 4).
available_time(june, 5).
```

An indexed database system is one efficient method that can be employed to deal with large numbers of individual terms. The existing GTO (Graphics, Text and Objects) database system within SlimWinX is well suited to store, access and update

these text-oriented terms, and is already integrated into the XSpaces system such that the GTO records are selectable tuples within the XSpaces tuplespace.

6.6 Runtime Configuration and Installation of Sub-agents

Recalling from Section 3.3.4 that the XSpace of a SlimWinX application is dynamically adjustable, this revised Shadowboard version of the code employs the technique to: install new sub-agents at runtime; allow the user to move sub-agents between two Envelopes-of-capability; and reconfigure the constituents of an Envelope-of-capability entirely. Such newly installed sub-agents may be *Shadowboard* agents that originate from remote sources in the manner of Mobile Agents (see Section 2.6.3.8).

6.7 Summary

In this chapter we have been concerned with an implementation strategy for the Shadowboard architecture drawing upon two existing but interrelated systems SlimWinX and XSpaces. We have looked into the object hierarchies and some of the functionality of the existing systems and have identified base classes upon which to develop the primary entities in the Shadowboard agent system, namely:

SlimWinX base class	**Shadowboard Entities**
SG_Program	Aware Ego Agent
SG_Panel	Envelope-of-capability
SG_Pane	Sub-agent

We also looked into the event handling system in SlimWinX and extensively outlined how it could be modified to become a *goal-driven* system. This would be one in which the runtime currency changed from being *event messages* passed around in an event-driven system, to one in which predicate *logic terms* and *constraints* are passed around as *goals* and *constraints* upon goals. The Event Queue from SlimWinX is mirrored and expanded with multiple Goal Queues in Shadowboard.

We looked into the ability of XSpaces to dynamically create objects and relationships between objects, and how this can be used to maintain sub-agents working on solving different goals and sub-goals in parallel – a necessity as various sub-agents monitor and act in the outer world, 24x7.

Now that we have researched and mapped a development path for an implementation of the Shadowboard architecture, in the next chapter we return to the task of specifying and implementing a complex Digital Self

7 IMPLEMENTING A DIGITAL SELF UPON SHADOWBOARD

What we have set out to achieve in this thesis is to define a much more fine-grained, more accurate user model than previously used, and build upon it an extensive and more empowering agent system than previously attainable, one worthy of the title of *Digital Self*. However, the sub-agent examples within a Digital Self cited so far are typically either Personal Assistant Agents (Email Sub-agent, Travel Sub-agent or a time scheduler such as a Diary Sub-agent), or Information Agents – see Sections 2.6.3.2 and 2.6.3.3. These deal with the most obvious and pedestrian information, notification and scheduling aspects of a user's activities on a network of users.

If the Digital Self is to help and represent the individual person in 24x7 time, then the sub-agents have to be more numerous and diverse in their function, more attuned to the *whole* person, more symbiotic and empowering for the user, and able to spark insights in the individual, often about themselves. In constructing and enhancing the Digital Self within the Shadowboard system, the user may have as much a learning experience about themselves, as they have about their subjects of interest in the outer world.

Just as we have drawn upon an analytical psychology model of mind to get both structural and computational aspects of the Shadowboard system in place in Chapters 6 and 7, we will now draw upon it further, to specify a sophisticated Digital Self with a broad-range and variety of sub-agents, from which an individual may choose in constructing his/her representation in 24x7.

7.1 Extensive List of Generic Subselves

To get beyond the limited number and types of generic sub-selves identified by Stone and Winkelman in the *Psychology of Subselves* (cited in Table 4.1), that one may be subscribing to in his/her life, a person has to go through a considerable amount of self-analysis and reflection. While such a task is not of direct interest to this thesis,

the following hierarchy of possible generic sub-selves is a further contribution by the author, resulting from work done prior to and outside of this thesis, that is gainfully employed here, in the context of specifying an extensive set of sub-agent components for a generic Digital Self:

- Decision Maker (envelope-of-capability)
  - Executive Director (archetypal)
  - Controller
  - Judge
  - Dictator (reactive)
- Manager (envelope-of-capability)
  - Benevolent Manager (archetypal)
  - Conciliatory Manager
  - Planner
  - Scheduler
  - Coordinator
  - Recycler
  - Decisive Manager (reactive)
- Protector (envelope-of-capability)
  - Safety Officer (archetypal)
  - Defender
  - Risk Analyst
  - Environmentalist
  - Pacifier
  - Doctor
  - Exit Strategist (reactive)
- Server (envelope-of-capability)
  - Selfless Pleaser (archetypal)
  - Service Provider
  - Networker
  - Communicator
  - Marketer
  - Self Server (reactive)
- Initiator (envelope-of-capability)
  - Pusher (archetypal)
  - Success Seeker
  - Wooer
  - Resource Master
  - Trouble Shooter
  - Do Little (reactive)
- Critic (envelope-of-capability)

- Perfectionist (archetypal)
  - Editor
  - Quality Controller
  - Doubter
  - Cynic (reactive)
- Intuitive (envelope-of-capability)
  - Seer (archetypal)
  - Mood Sensor
  - Pattern Finder
  - Dreamer
  - Day Dreamer (reactive)
- Adventurer (envelope-of-capability)
  - Explorer (archetypal)
  - Risk Taker
  - Traveler
  - Vacationer
  - Lazybones (reactive)
- Knowledge Seeker (envelope-of-capability)
  - Knowledge Worker (archetypal)
  - Concept Learner
  - Information Officer
  - Data Miner
  - Random Generator (reactive)
- Logician (envelope-of-capability)
  - Rationalist (archetypal)
  - Statistician
  - Business Analyst
  - Diagnostic
  - Lawyer
  - Dogmatist (reactive)
- Children (envelope-of-capability)
  - Player (archetypal)
  - Fun-seeker
  - Inventor
  - Vital Child
  - Vulnerable Child
  - The Fool (reactive)
- Teacher (envelope-of-capability)
  - Master (archetypal)
  - Ethicist
  - Linguist
  - Mathematician

Biologist
Economist
Apprentice (reactive)
Engineer (envelope-of-capability)
Constructor (archetypal)
Material Engineer
Structural Engineer
Mechanical Engineer
Chemical Engineer
Outsourcer (reactive)
Artist (envelope-of-capability)
Wordsmith
Visual Artist
Performance Artist
Spiritual Advisor (envelope-of-capability)
Wise One (archetypal)
Mentor
Humanist
Faithful
Spatial Coordinator (envelope-of-capability)
Geographer
3D Surveyor
2D Surveyor
Intra-personal (envelope-of-capability)
Profiler
Role Keeper
Standard Bearer

Figure 7.1 – A Generic Hierarchy of Possible Sub-selves.

7.2 Expressing the Generic Digital Self in XML

Taking the extensive list of possible subselves in Figure 7.1 and expressing them in XML using the Shadowboard.dtd gives us the structural blueprint for a complex generic Digital Self, from which most people could readily select some useful subset to represent their interests, 24x7. Figure 7.2 below represents a fragment of this XML, taken from the file DigitalSelf.xml, shown in Appendix B in its entirity.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE whole-agent SYSTEM "Shadowboard.dtd" >
<whole-agent global-id="IP:203.31.192.82" global-name="Deepthought"
                                                    global-type="Shadowboard">
  <aware-ego-agent>
    <sub-agent a-delegator="true" agent-id="AE1" agent-name="Brady"
          agent-type="AwareEgo" disowned="false" external-agent_id="A1" rank="1">
      <description>This is the Aware Ego agent.</description>
      <envelope-of-capability>
        <sub-agent a-delegator="true" agent-id="DM1" agent-name="DecisionMakers"
              agent-type="DecisionMaker" disowned="false" rank="1">
          <description/>
          <envelope-of-capability>
            <archetype>
              <sub-agent agent-id="DM2" agent-name="ExecutiveDirector"
                    agent-type="DecisionMaker" disowned="false" rank="1"/>
            </archetype>
            <sub-agent agent-id="DM3" agent-name="Controller"
                 agent-type="DecisionMaker" disowned="false" rank="2"/>
           ...
          </envelope-of-capability>
        </sub-agent>
      </envelope-of-capability>
    </sub-agent>
  </aware-ego-agent>
  <sub-persona animation-name="urbane.mpg" current-status="DEFAULT" event-mask="128"
               general-expression="confident" icon-name="urbane.gif"
               options-mask="DEFAULT" persona-id="A100" persona-name="Clive"
               persona-type="MPEG4"/>
</whole-agent>
```

Figure 7.2 – A fragment of a Complex Digital Self in XML.

Note: This file is *well-formed* XML, but it is not fully *validated*. I.e. The XML tags are syntactically correct, but numerous attributes of most elements have been left out of this figure, for human readability sake.

The primary *agent types* in Shadowboard are derived from the general envelopes-of-capability in Figure 7.1, namely:

Shadowboard Agent Type	**ID Prefix Code**
AwareEgo	AE

DecisionMaker	DM
Manager	MR
Protector	PR
Server	SR
Initiator	IR
Critic	CT
Intuitive	IV
Adventurer	AV
KnowledgeSeeker	KS
Logician	LN
OpenMinded	OM
Teacher	TR
Mentor	MT
Engineer	ER
Artist	AT
SpiritualAdvisor	SA
SpacialCoordinator	SC
Intrapersonal	IP
External Types referenced:	
PersonalAssistant	PA
Information	IF
Reactive	RV
DecisionSupport	DS
(and Believable via Sub-persona)	

Table 7.1 – Shadowboard Agent Types.

Note that the example sub-agents cited in the earlier SimpleDigitalSelf.xml of Figure 5.4 also reappear in Appendix B, but they are placed further down the hierarchy in this more complete DigitalSelf: *"MailMan"* has been moved into an envelope-of-capability within Communicator itself within Server; while *"InfoMaster"*, *"TheInfoGuru"* and *"Speedy"* have all been moved into an envelope-of-capability within *InformationOfficer* itself within *KnowledgeSeeker*. This is done as an example of

how several useful agents that would usually be added to a user's system in an adhoc manner can be integrated easily into the Shadowboard system.

7.3 XML and W3C DOM Compliant Internal Objects and Events

What is immediately obvious from this complex model of a Digital Self is the requirement for a large number of very specifialised yet diverse sub-agents. While the implementation of Shadowboard outlined in Chapter 6 does provide a mechanism for communicating and processing the logical terms and predicates that are the currency of the architecture, it is a system-oriented framework to be populated by agents from various sources, necessarily including many third party (non-Shadowboard) agents. One method of addressing this is the *disowned sub-agent* mechanism which has already been described. However, a future-oriented strategy that will be employed is to make the system as interoperable as possible with those W3C standards that most involve related software engineering endeavours, in particular: external declaration via XML; and internal object compatibility via the W3C DOM.

Though not necessary for the initial implementation of Shadowboard, the SG_Tree class will be made W3C DOM compliant, by renaming internal methods and adding a few extra methods for which it currently has no equivalents. The parser within SG_Tree (and SG_Token), which currently reads and writes the ASCII-based tree structure in the application.ini file (Eg. Appendix A), will be modified to directly read and write XML, so that declarations of Shadowboard agents in .xml - such as in the SimpleDigitalSelf.xml file back in Figure 5.4 - are directly accessible by the system.

W3C compliance can go further: given that the Shadowboard implementation upon SlimWinX has an intrinsic event queue and an internal object structure, it would also be advantageous to make the internal objects W3C DOM compliant. The advantages would stem from the amount of interaction expected with agents and systems outside of itself. As XML gains usage for declarative syntax across different agent systems, the automation of translations between the schemas of independently

developed systems will be possible. This will improve a Shadowboard agent's ability to inter-communicate and use growing numbers of external agents, as either contracted-out *disowned sub-agents*, or as *archetype sub-agents* to learn from or to calibrate against.

The SlimWinX event model has equivalents for all event types and terms expressed in the DOM Event Model as introduced in Section 3.5:

DOM	**SlimWinX**
UI events –	*EV_mouse and EV_keyboard events;*
UI Logical events –	*device independent command events, such as SG_Event;*
Mutation events –	*events that modify the structure of the tree at runtime (as used in drag'n'drop);*
Capturing events –	*the reverse of CS_PRIORITY, but which function in a like-manner;*
Bubbling –	*bubbling upwards through the ancestors after being handled, as effected by OP_PASS_ON_CLICK.*

By making the internal event system comply with the open and generic DOM Event Model, it becomes future-ready for participation in distributed processing at a deeper level than sub-agent declaration and instantiation. For example, a number of DOM compliant agent systems may, in the future, standardise on the meaning and use of a number of *DOM UI Logical* events, negating the need to manually compile descriptions of external agents and their methods of communication, such as is done for *external-agents* in the Shadowboard.dtd in Figure 5.3.

7.4 Training the Aware Ego Agent

An advantage of having the agent system intersect at the window level of the GUI system, such that every sub-agent has its own window (Section 5.10), is that an interactive training interface, inclusive of all sub-agents, is straightforward to build. During configuration, the user may initially train the Aware Ego Agent to select appropriate combinations of sub-agents and also rank competitive sub-agents, for solving a given user-goal. This is most effective for reoccurring goal types, or

perpetual goals, such as: receiving appropriate email; or advise buying shares whenever conditions are right.

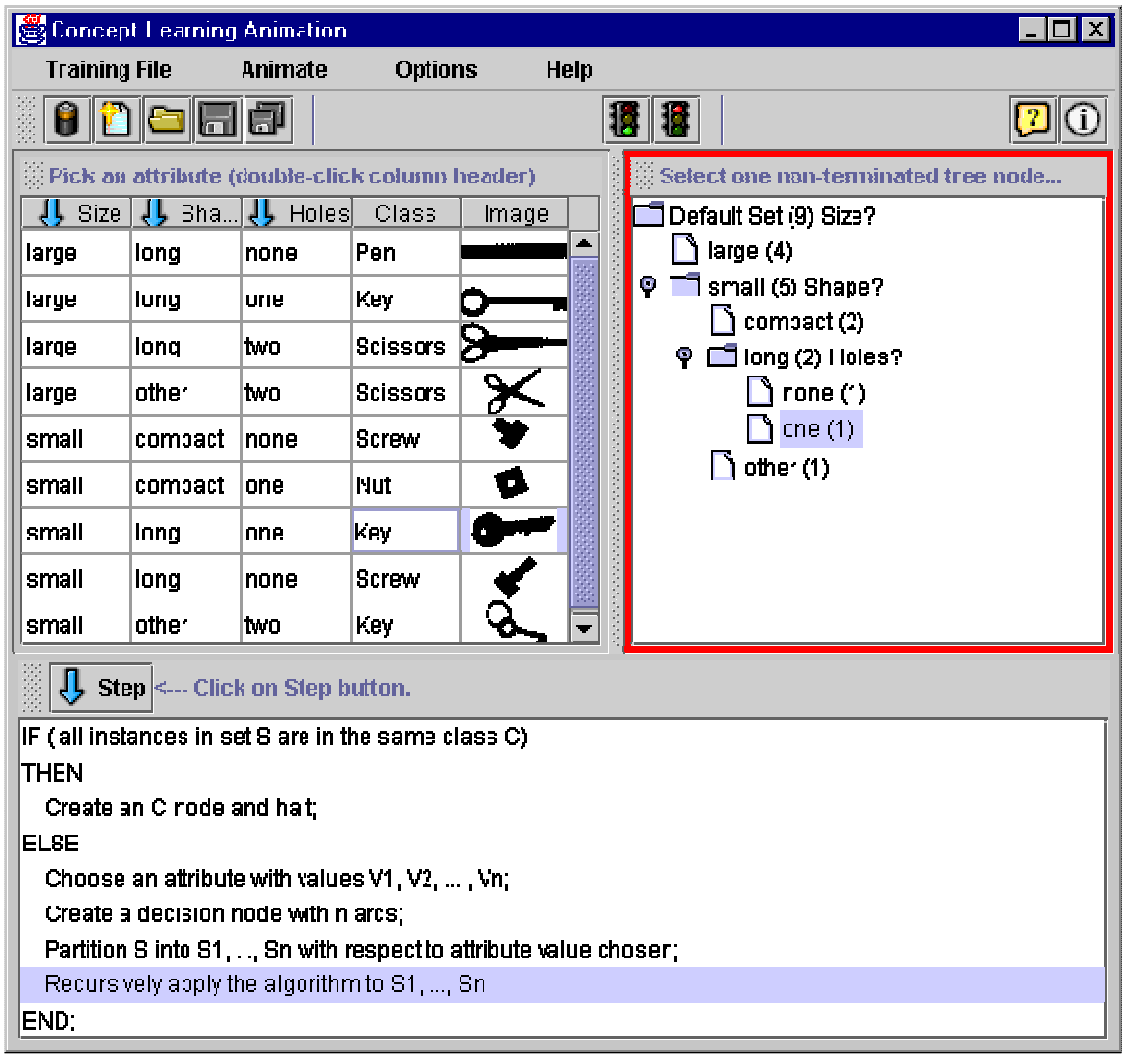

Figure 7.3 – Animation of the Concept Learning Scheme building a Decision Tree.

User training exercises driven by code enacting the Concept Learning Scheme (CLS) algorithm can *induce decision trees* – i.e. build decision trees. See Figure 6.3 above, which is a screenshot of a program by the author, which is an animation of CLS that builds and records decision trees (right-side window), based on a training data set (left-side window).

The resultant decision trees are expressible in logical form as conjunctions of rules, which the Aware Ego Agent is capable of storing and using directly. The same CLS system can be enhanced to induce new rules, where the constraint logic solvers have

returned solutions that the user had not choreographed in the training session(s). The incorporation and use of these induced rules would effectively make the system a self-optimizing agent-oriented generalised constraint solver.

7.5 Summary

In this chapter we have outlined the need for an extensive range of sub-agents which are in total, inclusive of some of the activities that a typical user will involve themselves in life, and hence within a 24x7 networked environment such as the Internet. We have gone back to analytical psychology for direction and come up with an extensive hierarchy (Figure 7.1) of generic subselves upon which to model the sub-agents and those Envelopes-of-capability in which the sub-agents are clustered, for a realistic and useful Digital Self.

We used the Shadowboard.dtd from Chapter 4 to express this hierarchy as a well-formed XML file named *DigitalSelf.xml*, to give us a reliable blueprint for a realistic and hence complex Digital Self.

In Table 7.1 we declared the Shadowboard Agent types which will cover the sub-agents for which we can foresee a use.

We looked to the W3C DOM activity as a strong candidate in the near future for enabling third party agents as *external agents* in a Shadowboard agent system. And we looked to a technique and a tool incorporating the Concept Learning System, as a likely path for the training of agents (and hence constraint solvers) within a Shadowboard system.

8 DISCUSSION AND CONCLUSION

We developed the concept of a sophisticated model of the user, one well grounded upon contemporary analytical psychology – the *Psychology of Subselves*. We built the Shadowboard Agent architecture upon that sophisticated model and depicted it graphically, verbally and as an XML DTD in Shadowboard.dtd in Chapter 5. The result is a flexible architecture capable of cohesively integrating all manner of Personal Assistant Agents, Information Agents, Interactive Agent and other agents, which were previously added to the user's system in a relatively adhoc manner.

In Chapter 6, as we researched and detailed an implementation strategy, and after deciding upon *predicate logic terms* and *constraints* as the messages which are passed around the system, we came to realise that both the Envelopes-of-capability (with its delegation sub-agent) and numerous sub-agents needed to be *constraint solvers*. Given the considerable number of sub-agents and Envelope-of-capability that would be in a typical implemented system, Shadowboard quickly took on the guise of a *sophisticated Generalised Constraint Solver* – at least with respect to its *decision making* system. This choice was foreshadowed in papers by Goschnick [31,32].

It is worth noting here, that the two main components in a Constraint Programming system are: the *constraint solver*; and the *search engine* - which implements some strategy for exploring the *search space,* e.g. backtracking. The Envelope-of-capability, together with its delegation agent holds rules and makes choices in the manner of a *logic constraint solver*. The event system, which has been modified to parse and queue *goals* and *constraints* rather than *events*, works in the manner of such a *search space* search engine. The sub-agents at the leaf nodes of the recursive structure of Shadowboard sense information from the outer world, transform it into new ground terms and hold it in store in their GTO files. Sub-agents are also responsible for putting actions out into the world. This *sensing* and *acting* in a 24x7 environment is the primary feature of the Shadowboard system, that makes it an Agent system first, and a Generalised Constraint Solver second. Nonetheless, the appropriate configuration of sub-agents would make for a powerful, flexible and useful Generalised Constraint Solver.

The following sections discuss how this agent architecture compares with MAS, and also reflects on contemporary agent issues such as *autonomy* and *trust*.

8.1 Shadowboard, Blackboard, MAS and Autonomy

The sub-agents of the Digital Self are analogous to the group of *specialists*, the *knowledge sources* of the Blackboard model (Section 2.4.5). The knowledge sources in the Blackboard system are often people, while the solution-space - the blackboard - is in the computer system. What we are doing with the Shadowboard architecture is putting sub-agents in as knowledge sources, each interacting with some subset of the solution space, which represents the superset of the user's interests and activities.

Penny Nii attributes the first reference to the term *blackboard* in AI literature to Newell [53], and quotes him thus:

> *"Metaphorically we can think of a set of workers, all looking at the same blackboard: each is able to read everything that is on it, and to judge when he has something worthwhile to add to it. This concept is just that of Selfridges's Pandemonium: a set of demons, each independently looking at the total situation and shrieking in proportion to what they see that fits their natures …"*

If one puts 'agents' into that paragraph as the 'workers', then it fits something of what a MAS system of autonomous agents is. It would fit the situation with Shadowboard except for the Aware Ego agent and its central role. It is the Aware Ego Agent's control over all the sub-agents, that makes it quite different from a MAS of autonomous agents. Even though some sub-agents may be working independently, or in a parallel sense, pushing facts up to the Aware Ego Agent in the data derived reasoning sense; this still does not make them autonomous agents. I.e. They may be interrupted at any time by the Aware Ego Agent with a more pressing problem; and those facts that they may be deriving, in a seemingly independent sense, will be in answer to perpetually held goals that the central agent has and holds for the user, such as: *Always receive email from Fred*.

Nonetheless, it is evident from the literature on multi-agent systems, particularly amongst those papers based on the practicalities of building MAS systems, that inner *sub-agents* are often constructed and intra-agent communication is then supported, in some of those systems [6,55]. As with the sub-agents in Shadowboard, those sub-agents are either semi-autonomous or totally subservient to a primary agent. The sub-agents in these other systems have been introduced for practical implementation reasons, or else for representing specific distinct Personal Assistant Agents. Where as the sub-agents conceptually developed here in Shadowboard are *first-class entities* based directly on *subselves* - mentalistic entities in the underlying analytical psychology. They are not entities watered-down from the underlying theory in order to produce a practical working model.

MAS systems usually make use of agent communication languages (ACLs) to enable co-operation between heterogeneous agents. However, the more information one has, the less the need for protocols - as seen with Blackboard Systems in Section 2.4.5, co-operation can be achieved without a specific communication language, even across heterogeneous hardware platforms. The sub-agent system within Shadowboard, implemented upon SlimWinX and XSpaces as detailed in Chapter 6, gains substantially in efficiency and effort, due to the implicit communication mechanism used.

8.2 Trust and Transparency

Lewis [46] cites the Concept Learning Scheme of human categorisation schemes as too black-boxed for a user to gain much trust in the agents acting on his/her behalf. Trust is also a heavily discussed issue in the Believable agent area. Trust is needed for people to yield certain tasks to agents. The user needs explicit feedback and a good degree of transparency. The Shadowboard agent has a natural visual profile via its integration into the GUI system. However, features such as the induction of decision-trees need to be interactively examinable by the user, in order to gain a degree of transparency. The animation of the CLS algorithm in Figure 7.3 is a technique that can be used to display the process transparently in a graphical manner – and hence gain trust.

In addition, the sheer number, choice and diversity of sub-agents that can be incorporated into a Digital Self makes for a far more trustable entity than a single Believable Agent. Agents and humans need accurate models of each other for trust to grow, and the complexity of the user model in Shadowboard is more accurate than any agent model before it, such as BDI - which we have directly compared to the much early Freudian model of the psyche.

8.3 Self Awareness and Sub-agency

The Digital Self represented in Chapter 7 was just one of an unlimited number that could be declared using the Shadowboard.dtd because of the recursive structure of Shadowboard. The one we chose, together with the Shadowboard agent-types we selected, demonstrates that a Digital Self with a broad range of sub-agents is capable of helping and representing the user on a 24x7 network, in most facets of a life, and in a manner way beyond the niches so far addressed by information, notification and scheduling personal assistant agents. The degree that a user's life will be genuinely empowered by this agent system is significantly dependent on two things:

1. The availability of sub-agents, which genuinely service the broad range of needs and interests of an individual.

2. The extent of the self-knowledge that the user has.

We believe the disowned-agent concept, together with an embrace of certain open standards, is capable of addressing the first issue. The second issue has wide variance across individuals, but the very process of specifying and then using an extensive Digital Self will probably lead the user on a path of growth. The contemplation of the possible subselves and the circumspection of the appropriate sub-agents one will employ, then observing them in action - will probably lead the user to some of those things that agent researchers have been catering for in our agents for a good long while now: *goal revision*, *belief revision*, *intention revision* and *plan selection*. There is a complementary dichotomy between agents learning from humans and humans learning from agent simulations. With Shadowboard, the user - in receipt of feedback from the various sub-agents in the Digital Self, and able to look

into the mechanism of rule induction and other sub-agent internals - is gaining far more than *trust* from the process: he/she is learning about Self from a model of Self.

8.4 Future Research

Several areas for future research have already been flagged as possible areas for us to actively pursue, and a few are added here:

- Having taken the decision making process of the system into the realm of constraint solvers, incorporating sophisticated techniques from the Constraint Programming paradigm would be fertile ground for research.

- As outlined in Section 7.3, as the W3C DOM specification evolves, it is a likely candidate area for research into ways and means of easily incorporating third party *external agents* into a Shadowboard system.

- The extensive Digital Self offered up in Chapter 7 as a structural blueprint for a complex generic Digital Self needs building and then testing against whole cohorts of Internet users. The testing, measuring and adjusting of the *Shadowboard Agent types* declared in Table 7.1, with regard to general usefulness, is fertile ground for future research.

- Software systems that help users identify subselves and roles that hold sway in their lives are fertile and novel territory for future research.

8.5 Conclusion

Regardless of the degree to which the system empowers a given individual, or how many useful sub-agents the user can amass, the Shadowboard architecture is generic enough to accommodate what he/she comes up with. It enables the building of software that is more attuned to the multi-functional roles that an individual has within his or her modern life. By incorporating logic and constraint solvers into the implementation, the Digital Self becomes a powerful real-time aide, one in which the user could add simply specified constraints, unlike any tool currently available. And by taking into consideration the multiple facets of a person's character and life, the

software built upon Shadowboard will be better tailored to serve the individual on a 24 hour, 7 day per week basis.

The devices that could best be partnered with such software include: workstations with permanent Internet connections; set-top boxes tailored for both Internet and Interactive TV; and other personal digital assistants (PDAs) such as Internet enabled 24x7 web-phones. The markets for these types of devices are emerging as strong IT growth areas in the first decade of the 21st century as the market for the traditional desktop PC wanes, and so the economic benefits derivable from Shadowboard ought to be considerable.

The social benefits will be derived in the form of a better meshing between people and the technology they use. This will improve the symbiotic relationship between man and machine, thereby making humans more efficient, effective and influential, amongst immediate peers and within *greater society* – the international society operating in 24x7 time.

In the near future, we want to build the Shadowboard implementation as detailed in Chapter 6.

References:

[1] *Age of Empires*, real-time strategy game, 2001. Details available at <http://www.ensemblestudios.com/aoe/index.shtml>

[2] Allport G.W. *The Nature of Prejudice*. 25th Anniversary Edition. Adison-Wesley, 1979.

[3] *Ananova*, the virtual newscaster, 2001. Available at <http://www.ananova.com/>

[4] Assagioli, Roberto. *Psychosynthesis*. New York. Viking, 1965.

[5] Billinghurst M. and Starner T. Wearable Devices: New Ways to Manage Information, *Computer*, pp57-64, January 1999.

[6] Botti V., Carrascosa C., Julian V. and Soler J. Modelling Agents in Hard Real-Time Environments, pp 63-76 in *Multi-Agent Systems Engineering*, Garijo F. J. and Boman, M.(eds), *9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, MAAMAW'99. Springer, LNAI 1647. 1999.

[7] Bratman M.E., Israel D.J. and Pollack M.E. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4), 1988.

[8] Brooks C., Durfee E.H. and Armstrong, A. An Introduction to Congregating in Multiagent Systems. In, *Proceedings of Fourth International Conference on MultiAgent Systems* (ICMAS-2000), pp 79-86, 2000.

[9] Castelfranchi C. Commitments: from individual intentions to groups and organisations. In *Proceedings of International Conference on Multi-Agent Systems* (ICMAS'95), 1995.

[10] Castelfranchi C., Dignum F., Jonker C.M., Treur J. Deliberative Normative Agents: Principles and Architecture. *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages* (ATAL-99), Orlando, July 15-17, 1999.

[11] Cavedon L. and Sonenberg E.A. On Social Commitment, Roles and Preferred Goals. In *Proceedings of the 1998 International Conference on Multi-Agent Systems* (ICMAS'98), Paris, Demazeau Y., (ed), pp 80-87, July 1998.

[12] Cohen P.R. and Levesque H.J. Intention is choice with commitment. *Artificial Intelligence*, 42(3), 1990.

[13] Compound Presentation and Interchange. Revised submission by Apple Inc to the Common Facilities Task Force RFP1, 5-part document 1995/95-12-30,31,32,33,34, 1995. Available at <http://www.omg.org/technology/documents/vault>

[14] CORBA 2.5 Specification. 2001. Documentation available at <http://www.omg.org/technology/documents/specifications.htm>

[15] CORBA Mobile Agent Facility. Documentation available at <http://www.omg.org/technology/documents/formal/mobile_agent_facility.htm>

[16] DCOM: the Distributed Component Object Model, 2001. Documention and software available at <http://www.microsoft.com/com/tech/dcom.asp>

[17] Dignum F., Sonenberg L. and Kinny D. *Formalizing motivational attitudes of agents: On desires, obligations and norms.* Submitted to workshop on Agent Theories, Architectures, and Languages (ATAL-2001), 2001.

[18] Dignum F., Morley D., Sonenberg E.A. and Cavedon L. Towards socially sophisticated BDI agents. In, *Proceedings of Fourth International Conference on MultiAgent Systems(ICMAS-2000)*, pp 111-8, 2000.

[19] Elliott C. and Brzezinski J. Autonomous Agents as Synthetic Characters, *AI Magazine*, American Association for Artificial Intelligence, pp13-30, Summer, 1998.

[20] Engelmore R. and Morgan T. (eds.). *Blackboard Systems*. Addison-Wesley, 1988.

[21] Enterprise JavaBeans. 2001. Documentation and system available at <http://java.sun.com/products/ejb/>

[22] Etzioni O. Moving Up the Information Food Chain: Deploying Softbots on the World Wide Web. Abstract of invited talk, in *Proceedings of AAAI-96*, 1996.

[23] Extensible Markup Language (XML) 1.0. Second edition, 2000. Specification available at <http://www.w3.org/XML/>

[24] Extensible Stylesheet Language (XSL), W3C XSL Working Group, 2001. Available at <http://www.w3.org/Style/XSL/>

[25] Finin T., Labrou Y. and Mayfield J. KQML as an agent communication language. *Proceedings of the 3rd International Conference on Information and Knowledge Management* (CIKM'94), 1994.

[26] *FIPA ACL Message Structure Specification*. Foundation for Intelligent Physical Agents, 2001. Available at < http://www.fipa.org/specs/fipa00061/>

[27] Firby R.J. An Architecture for A Synthetic Vacuum Cleaner. *Proceedings of AAAI Symposium on Instantiating Real-World Agents*, 1993.

[28] Gelernter D. Generative Communication in Linda, *TOPAS*, 7 (1), 1985.

[29] Gelernter D. and Bernstein A.J. Distributed Communications via Global Buffer, *Proc. of the ACM*, 32(4), 1989.

[30] Goldberg A. *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Publishers, Menlo Park, 1983.

[31] Goschnick S.B. Shadowboard: A Whole-Agent Architecture that draws Abstractions from Analytical Psychology, *Proceedings of PRIMA 2000*, Melbourne, August 2000.

[32] Goschnick S.B. Shadowboard: Revisiting the individual in MAS. *Technical Report TR2000/6*, Dept. of Computer Science & Software Engineering, University of Melbourne, May 2000.

[33] Goschnick S.B. *SlimWinX Technology*, Executive Overview, Internal document, Solid Software Pty Ltd, 1996.

[34] Goschnick S.B. *BranchOut – Multimedia Game Concept*, Executive Overview, Internal document, Solid Software Pty Ltd, 1996.

[35] Goschnick, Steve and Sterling, Leon. Shadowboard an Agent-oriented MVC Architecture for a Digital Self. Paper accepted in: *Internation Workshop on Agent Technologies over Internet Applications (ATIA)*, in conjunction with Seventh Distributed Multimedia Systems (DMS'2001), Taiwan, September 2001.

[36] Green T.R. and Benyon D.R. *The skull beneath the skin: Entity-relationship modelling of Information Artefacts.* International Journal of Human-Computer Studies 44(6) pp. 801-28, 1996.

[37] Griffiths N. and Luck M. Cooperative Plan Selection Through Trust. In *Multi-Agent System Engineering*, Garijo F. J. and Boman, M.(eds), 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99. Springer, LNAI 1647, pp 162-74, 1999.

[38] Grosz B.J. and Kraus S. Collaborative Plans for Complex Group Action. *Artificial Intelligence*, 86(2), 1996.

[39] HTML 4.01 Specification, W3C Recommendation, December 1999. Available at <http://www.w3.org/TR/html401/>

[40] Jennings N.R. and Wooldridge, M.J. *Agent Technology: Foundations, Applications and Markets*, 1995.

[41] Jini Network Technology. 2001. Documentation and system available at <http://www.sun.com/jini/>

[42] Jung C.G. *Man and his Symbols*. Aldus Books, 1964.

[43] Kitani H. RoboCup Rescue: A Grand Challenge for Multi-Agent Systems. In, *Proceedings of Fourth International Conference on MultiAgent Systems(ICMAS-2000)*, pp 5-12, 2000.

[44] Krasner G.E. and Pope S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object Oriented Programming*, pp.26-49, Aug/Sep, 1988.

[45] Lashkari Y., Metral M. and Maes P. Collaborative Interface Agents. *In Proceedings of the 12th National Conference of Artificial Intelligence*, 444-450, 1994.

[46] Lewis M. Designing for Human-Agent Interaction*. AI Magazine*, pp. 67-78, Summer, 1998.

[47] Luenberger D.G. *Introduction to Dynamic Systems - Theory, Models and Applications.* John Wiley & Sons, 1979.

[48] Maes P. and Kozierok R. Learning Interface Agents. In *Proceedings of AAAI Conference*, 1993.

[49] Malone, Tom. Keynote talk. ICMAS-2000 – International Conference on Multi-Agent Systems, Boston, July 7-12, 2000.

[50] Marriott K. and Stuckey P.J. *Programming with Constraints: an Introduction.* MIT Press. 1998.

[51] Minsky, Marvin. *The Society of Mind.* Simon and Schuster Inc. 1986.

[52] Negroponte, Nicholas. *Being Digital*, Random House, 1995.

[53] Newell, A. Some problems of basic organization in problem-solving programs. In Yovits, M.C., Jacobi, G.T. and Goldstein G.D. (eds*) Conference on Self-Organizing Systems*, Washington, C.C, Spartan Books, 1962.

[54] Newman W.M. and Lamming M.G. *Interactive System Design*. Addison-Wesley, p.6, 1995.

[55] Nikolaos, Sharmeas. *Agents as Objects with Knowledge Base State*. Imperial College Press, 1999.

[56] Noda I., Suzuki S., Matsubara H. and Kitano H. RoboCup-97 the first robot world cup soccer games and conferences. *AI Magazine*, pp49-59, Fall 1998

[57] Norling E., Sonenberg E.A. and Ronnquist R. *Enhancing Multi-Agent Based Simulation with Human Decision-Making Strategies.* Paper presented at the Fifth Australasian Cognitive Science Conference, Melbourne, Australia, Jan. 2000.

[58] Norman D. *The Psychology of Everyday Things,* Harper Collins Publishers, 1988.

[59] Penny Nii, H. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. In: *Computation and Intelligence - Collected Readings*, Luger, G.F. (ed), The MIT Press, 1995.

[60] Rao A.S. and Georgeff M.P. An Abstract Architecture for Rational Agents. *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[61] Rao A.S. and Georgeff M.P. Modelling rational agents within a BDI-architecture. In, Allen J., Fikes E. and Sandewall (eds) *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[62] Rao A.S. and Georgeff M.P. An Abstract Architecture for Rational Agents. *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 439-449, 1992.

[63] RoboCup 2000, the 4th World Robot Soccer Championships and Workshop, Melbourne, 2000. Details available at
<http://www.robocup2000.org/htm/content/frameset.html>

[64] Shoham Y. Agent-oriented programming. *Artificial Intelligence* 60:51-92, 1993.

[65] Sliker, Gretchen. *Multiple Mind - Healing the Split in Psyche and World.* Shambhala Publications Inc. 1992.

[66] Shneiderman, B. *Designing the User Interface, Strategies for Effective Human-Computer Interaction*, third edition, Addison-Wesley, 1997.

[67] Shneiderman B. and Maes P. Direct Manipulation versus Interface Agents. *Interactions*, (4)6, 1997.

[68] Star Craft, real-time strategy game, 2001. Details available at
<http://www.blizzard.com/starcraft/>

[69] Sterling, Leon. On Finding Needles in WWW Haystacks. In *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence*, Springer Lecture Notes in Computer Science, Vol. 1342, pp 15-36, 1998.

[70] Sterling, Leon and Shapiro, Ehud. *The Art of Prolog*, Second Edition, MIT Press, 1994.

[71] Stone, Hal and Winkelman, Sidra. *Embracing Ourselves - the Voice Dialogue Manual.* New World Library. 1989.

[72] St. Clair M. *Object Relations and Self Psychology*. Brooks/Cole Publishing Co., 1996.

[73] Suppes P., Pavel M., and Falmagne J. Presentations and Models in Psychology. In Porter, L.W. and Rosenzweig, M.R., editors, *Annual Review of Psychology*, 45:517-44, 1994.

[74] *Telescript Language Reference*, Version 1.0. General Magic, Inc. October, 1995. Available at: <http://science.gmu.edu/~mchacko/Telescript/docs/telescript.html>

[75] Tambe M. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, (7), pp 83-124, 1997.

[76] Tambe M., Pynadath D.V., Chauvat N., Das A. and Kaminka G.A. Adaptive Agent Integration Architectures for Heterogeneous Team Members. In, *Proceedings of Fourth International Conference on MultiAgent Systems(ICMAS-2000)*, pp 301-8, 2000.

[77] Tidbar G. Team-oriented programming: Social structures. *Technical Report 47*, Australian Artificial Intelligence Institute, 1993.

[78] Tidbar G., Sonenberg E. and Rao A.S. A Framework for BDI Teams. *Technical Report TR 1999/12*, Dept. of Computer Science and Software Engineering, University of Melbourne, 1999.

[79] Watt S. Artificial Societies and Psychological Agents. In *British Telecom Journal, Special Issue on Intelligent Agents*, Autumn 1996.

[80] Wyckoff P., McLaughry S.W., Lehman T.J. and Ford D.A. T Spaces. *IBM Systems Journal*, 37(3), 1998.

[81] Wooldridge, Michael. Agent-Based Software Engineering. In *IEEE Proceedings on Software Engineering*, 144(1), pp 26-37, 1997.

[82] Wooldridge, Michael and Jennings, Nicholas. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115-152, 1995.

[83] World Wide Web Consortium Document Object Model (DOM). 2001. Documentation available at <http://www.w3.org/DOM/>

[84] Young-Eisendrath, Polly and Hall, James. *Jung's Self Psychology: a Constructivist Perpsective.* The Guilford Press, 1991.

[85] Yuen C.K. and Feng M.D. Active objects as atomic control structures in BaLinda K, Computer Languages 24, 229-244, Pergamon, 1998.

[86] Zini, Floriano, and Sterling, Leon. *Designing Ontologies for Agents*. Dept of Comp. Science, University of Melbourne, Technical Report 1999/15.

Appendix A - Full Hierarchical XSpace for the Branch-Out Application

```
PROGRAM,POffice,0,1,3,0
{
    PANEL,POffice_Keypad1,1,0,11,10,0,374,640,104,4,0
    {
        PANE,pad1_5,1,0,30,64,372,383,72,72,4,BRIEF,BRIEF1,4000,170,0
        PANE,pad1_0,1,0,30,65,7,383,72,72,4,GENIE,GENIE1,3B00,108,0
        PANE,pad1_1,1,0,30,66,80,383,72,72,4,MAILBOX,MAILBOX1,3C00,101,0
        PANE,pad1_2,1,0,30,67,153,383,72,72,4,LIBRARY,LIBRARY1,3D00,102,0
        PANE,pad1_3,1,0,30,68,226,383,72,72,4,TREE,TREE1,3E00,103,0
        PANE,pad1_4,1,0,30,69,299,383,72,72,4,TRAN,TRAN1,3F00,104,0
        PANE,pad1_6,1,0,30,70,445,383,72,72,4,GLOBE,GLOBE1,4100,106,0
        PANE,pad1_7,1,0,30,71,518,383,36,36,2,SWICH32,SWICH321,4200,100,0
        PANE,pad1_8,1,0,30,72,554,383,36,36,2,EXIT32,EXIT321,4F00,283,0
        PANE,pad1_9,1,0,30,73,518,419,36,36,2,ABOUT32,ABOUT321,4300,9100,9101
        PANE,pad1_10,1,0,30,74,554,419,36,36,2,NOTE32,NOTE321,4400,109,0
        PANE,pad1_11,1,0,30,75,591,383,40,72,4,HELP,HELP1,7800,191,0
    }
    PANEL,POffice_Mainwin,1,0,12,0,0,0,640,378,0,GENEON.PCX,0,0
    {
        PANEL,PHelp_Modal,1,1,15,0,595,387,32,64
        {
            PANE,Help_wand1,1,0,41,53,HELP2,595,387,30,32,64,64,HELP3,0
        }
        PANEL,PBrief,1,1,13,0,400,215,195,150,1,1
        {
            PANEL,PBrief_Small_main,1,0,14,0,402,217,191,98
            {
                PANE,PHeirloom1,1,0,41,54,ICOH10,410,225,30,72,72,1,ICOS10,0
            }
            PANEL,PBrief_Keypad1,1,0,11,0,410,315,178,36,0,0
            {
                PANE,pad71_0,1,0,30,0,410,315,36,36,2,RIGHT32,RIGHT321,4D00,173,0
                PANE,pad71_1,1,0,30,1,447,315,36,36,2,LEFT32,LEFT321,4B00,174,0
                PANE,pad71_2,1,0,30,2,514,315,36,36,2,ESC32,ESC321,11B,171,0
                PANE,pad71_3,1,0,30,3,551,315,36,36,2,HELP32,HELP321,3B00,172,0
            }
        }
        PANEL,PMail,1,1,13,1,20,215,195,150,1,1
        {
            PANEL,PMail_Small_main,1,0,14,1,22,217,191,98
            {
                PANE,PHeirloom2,1,0,41,55,ICOH24,30,225,30,56,67,1,ICOS24,0
            }
            PANEL,PMail_Keypad1,1,0,11,1,30,315,178,36,0,0
            {
                PANE,pad72_0,1,0,30,4,134,315,36,36,2,ESC32,ESC321,3B00,181,0
                PANE,pad72_1,1,0,30,5,171,315,36,36,2,HELP32,HELP321,3C00,182,0
            }
        }
```

```
PANEL,PGame1,1,1,10,6
{
    PANE,esc_game1,1,0,30,92,598,438,36,36,2,ESC32,ESC321,011B,9999,0
}
PANEL,About_Mainwin,1,0,12,10,0,0,640,374,0,ABOUT.PCX,0,0
{
}
PANEL,PBboard,1,1,10,0
{
    PANEL,PBboard_Mainwin,1,0,12,1,0,0,640,378,0,BBS.PCX,0,0
    {
    }
    PANEL,PBboard_Keypad1,1,0,11,2,1,379,737,72,4,0
    {
        PANE,pad21_0,1,0,30,6,151,383,72,72,4,NEXT,NEXT1,3B00,120,0
        PANE,pad21_1,1,0,30,7,224,383,72,72,4,PREVIOS,PREVIOS1,3C00,121,0
        PANE,pad21_2,1,0,30,8,297,383,72,72,4,TAKECAS,TAKECAS1,3D00,122,0
        PANE,pad21_3,1,0,30,9,370,383,72,72,4,UNSOLVE,UNSOLVE1,3E00,123,0
        PANE,pad21_4,1,0,30,10,443,383,72,72,4,GLOBE,GLOBE1,3F00,124,0
        PANE,pad21_5,1,0,30,11,515,383,36,36,2,DESK32,DESK321,4700,9999,0
        PANE,pad21_6,1,0,30,12,551,383,36,36,2,TREE32,TREE321,4800,126,0
        PANE,pad21_7,1,0,30,13,515,419,36,36,2,BRIEF32,BRIEF321,4B00,170,0
        PANE,pad21_8,1,0,30,14,551,419,36,36,2,NOTE32,NOTE321,4D00,128,0
        PANE,pad21_9,1,0,30,15,588,383,40,72,4,HELP,HELP1,7800,191,0
    }
}
PANEL,PFtree,1,1,10,4
{
    PANEL,PFtree_Mainwin,1,0,12,2,0,0,640,378,0,TREE.PCX,0,0
    {
    }
    PANEL,PFtree_Keypad1,1,0,11,3,28,379,598,72,4,0
    {
        PANE,pad31_0,1,0,30,16,32,383,38,72,3,ADDCHI,ADDCHI1,3B00,9110,0
        PANE,pad31_1,1,0,30,17,71,383,38,72,3,ADDPAR,ADDPAR1,3C00,9111,0
        PANE,pad31_2,1,0,30,18,110,383,38,72,3,ADDSIB,ADDSIB1,3D00,110,0
        PANE,pad31_3,1,0,30,19,149,383,38,72,3,DEL,DEL1,5300,9112,0
        PANE,pad31_4,1,0,30,20,188,383,36,36,2,UP32,UP321,4800,9113,0
        PANE,pad31_5,1,0,30,21,188,419,36,36,2,DOWN32,DOWN321,5000,9114,0
        PANE,pad31_6,1,0,30,22,225,383,72,72,4,LISTGIF,LISTGIF1,3E00,111,0
        PANE,pad31_7,1,0,30,23,298,383,72,72,4,LISTA_Z,LISTA_Z1,3F00,112,0
        PANE,pad31_8,1,0,30,24,371,383,72,72,4,TRAN,TRAN1,4000,193,0
        PANE,pad31_9,1,0,30,25,444,383,72,72,4,GLOBE,GLOBE1,270F,115,0
        PANE,pad31_10,1,0,30,26,517,383,36,36,2,DESK32,DESK321,5000,9999,0
        PANE,pad31_11,1,0,30,27,553,383,36,36,2,BOOMERA,BOOMERA1,4F00,116,0
        PANE,pad31_12,1,0,30,28,517,419,36,36,2,BRIEF32,BRIEF321,5000,170,0
        PANE,pad31_13,1,0,30,29,553,419,36,36,2,NOTE32,NOTE321,4F00,118,0
        PANE,pad31_14,1,0,30,30,590,383,40,72,4,HELP,HELP1,7800,191,0
    }
}
PANEL,PGlobe,1,1,50,0
{
    PANE,pad42_0,1,0,30,31,0,18,152,61,0,(null),IMP1,3B00,1135,0
    PANE,pad42_1,1,0,30,32,526,28,104,101,0,(null),CHERUB1,3C00,1136,0
```

```
    PANE,pad41_0,1,0,31,0,0,0,128,160,4800,1120
    PANE,pad41_1,1,0,31,1,128,0,128,160,4800,1121
    PANE,pad41_2,1,0,31,2,256,0,128,160,4800,1122
    PANE,pad41_3,1,0,31,3,384,0,128,160,4800,1123
    PANE,pad41_4,1,0,31,4,512,0,128,160,4800,1124
    PANE,pad41_5,1,0,31,5,0,160,128,160,4800,1125
    PANE,pad41_6,1,0,31,6,128,160,128,160,4800,1126
    PANE,pad41_7,1,0,31,7,256,160,128,160,4800,1127
    PANE,pad41_8,1,0,31,8,384,160,128,160,4800,1128
    PANE,pad41_9,1,0,31,9,512,160,128,160,4800,1129
    PANE,pad41_10,1,0,31,10,0,320,320,160,4800,1130
    PANE,pad41_11,1,0,31,11,128,320,128,160,4800,1131
    PANE,pad41_12,1,0,31,12,256,320,128,160,4800,1132
    PANE,pad41_13,1,0,31,13,384,320,128,160,4800,1133
    PANE,pad41_14,1,0,31,14,512,320,128,160,4800,1134
}
PANEL,PBeame,1,1,51,0
{
    PANEL,PBeame_Mainwin,1,0,12,3,0,0,640,378,0,TRANSPTR.PCX,0,0
    {
        PANE,pad52_0,1,0,31,15,270,161,96,193,5000,1
    }
    PANEL,PBeame_Keypad1,1,0,11,4,99,418,524,36,2,0
    {
        PANE,pad51_0,1,0,30,33,99,418,36,36,2,DOWN,DOWN,4700,130,0
        PANE,pad51_1,1,0,30,34,505,419,36,36,2,DESK32,DESK321,4800,9999,0
        PANE,pad51_2,1,0,30,35,541,419,36,36,2,BRIEF32,BRIEF321,4900,170,0
        PANE,pad51_3,1,0,30,36,585,419,36,36,2,HELP32,HELP321,4900,132,0
    }
}
PANEL,PLocale,1,1,10,1
{
    PANEL,PLocale_Keypad2,1,0,11,5,11,30,64,290,0,0
    {
        PANE,pad62_0,1,0,30,37,11,30,64,32,0,THEIRSU,THEIRS,4700,160,0
        PANE,pad62_1,1,0,30,38,11,64,64,32,0,MOODSU,MOODS,4800,161,0
        PANE,pad62_2,1,0,30,39,11,98,64,32,0,GREETINU,GREETING,4900,162,0
        PANE,pad62_3,1,0,30,40,11,132,64,32,0,GESTSU,GESTS,5100,163,0
        PANE,pad62_4,1,0,30,41,11,200,64,32,0,TALLIESU,TALLIES,4D00,164,0
        PANE,pad62_5,1,0,30,42,11,234,64,32,0,VISITSU,VISITS,4900,165,0
        PANE,pad62_6,1,0,30,43,11,268,64,32,0,QUERYU,QUERY,270F,166,0
    }
    PANEL,PLocale_Keypad3,1,0,11,6,565,30,64,290,0,0
    {
        PANE,pad63_0,1,0,30,44,565,30,64,32,0,YOURSU,YOURS,4700,200,0
        PANE,pad63_1,1,0,30,45,565,64,64,32,0,YMOODSU,YMOODS,4800,201,0
        PANE,pad63_2,1,0,30,46,565,98,64,32,0,YGREETIU,YGREETIN,4900,202,0
        PANE,pad63_3,1,0,30,47,565,132,64,32,0,YGESTSU,YGESTS,5100,203,0
        PANE,pad63_4,1,0,30,48,565,166,64,32,0,DRESSESU,DRESSES,4D00,204,0
        PANE,pad63_5,1,0,30,49,565,200,64,32,0,QUESTIOU,QUESTION,4900,205,0
        PANE,pad63_6,1,0,30,50,565,234,64,32,0,GOU,GO,270F,206,0
        PANE,pad63_7,1,0,30,51,565,268,64,32,0,QBUTTONU,QBUTTON,270F,207,0
    }
    PANEL,PLocale_Keypad1,1,0,11,7,4,379,625,72,4,0
```



```
{
    PANE,pad61_0,1,0,30,52,8,383,72,72,4,LIBRARY,LIBRARY1,4700,140,152
    PANE,pad61_1,1,0,30,53,81,383,72,72,4,EMBASSY,EMBASSY1,4800,141,152
    PANE,pad61_2,1,0,30,54,154,383,72,72,4,ARCHIVE,ARCHIVE1,4900,142,152
    PANE,pad61_3,1,0,30,55,227,383,72,72,4,SCHOOL,SCHOOL1,5100,144,152
    PANE,pad61_4,1,0,30,56,300,383,72,72,4,POST,POST1,4D00,143,152
    PANE,pad61_5,1,0,30,57,373,383,72,72,4,CEMETRY,CEMETRY1,4900,145,152
    PANE,pad61_6,1,0,30,58,446,383,72,72,4,GLOBE,GLOBE1,270F,146,152
    PANE,pad61_7,1,0,30,59,518,383,36,36,2,DESK32,DESK321,5000,9999,0
    PANE,pad61_8,1,0,30,60,554,383,36,36,2,TREE32,TREE321,4F00,148,0
    PANE,pad61_9,1,0,30,61,518,419,36,36,2,BRIEF32,BRIEF321,5000,170,0
    PANE,pad61_10,1,0,30,62,554,419,36,36,2,NOTE32,NOTE321,4F00,150,0
    PANE,pad61_11,1,0,30,63,591,383,40,72,4,HELP,HELP1,7800,191,0
}
PANEL,PLocale_Mainwin,1,0,52,0,0,0,640,378,0,LOCN01.PCX,0,0
{
    PANEL,PLibrary,1,0,12,9,0,0,640,378,0,LIBRARY.PCC,90,30
    {
        PANE,h1_0,1,0,31,16,172,65,126,19,3B00,1200
        PANE,h1_1,1,0,31,17,152,98,185,24,3C00,1201
        PANE,h1_2,1,0,31,18,113,143,245,26,3D00,1202
        PANE,h1_3,1,0,31,19,143,186,240,25,3E00,1203
        PANE,h1_4,1,0,31,20,112,223,237,33,3F00,1204
        PANE,h1_5,1,0,31,21,132,262,231,42,4000,1205
    }
    PANEL,PEmbassy,1,0,12,8,0,0,640,378,0,EMBASSY.PCC,90,30
    {
    }
    PANEL,PPost,1,0,12,4,90,30,460,281,0,POST.PCC,90,30
    {
    }
    PANEL,PArchive,1,0,12,5,90,30,460,281,0,ARCHIVE.PCC,90,30
    {
    }
    PANEL,PSchool,1,0,12,6,90,30,460,281,0,SCHOOL.PCC,90,30
    {
    }
    PANEL,PCemetry,1,0,12,7,90,30,460,281,0,CEMETRY2.PCC,90,30
    {
        PANE,h2_0,1,0,31,22,125,116,66,83,3B00,1210
        PANE,h2_1,1,0,31,23,216,114,39,98,3C00,1211
        PANE,h2_2,1,0,31,24,276,120,51,100,3D00,1212
        PANE,h2_3,1,0,31,25,335,138,53,92,3E00,1213
        PANE,h2_4,1,0,31,26,392,171,31,59,3F00,1214
        PANE,h2_5,1,0,31,27,432,143,36,96,4000,1215
        PANE,h2_6,1,0,31,28,467,123,54,102,4100,1216
    }
}
PANEL,their_mood,1,1,13,2,77,36,72,94,0,0
{
    PANE,their_mbutn,1,0,30,76,77,36,72,94,0,ARRO,ARRO1,4700,1,0
}
PANEL,their_gesture,1,1,13,3,77,126,72,114,0,0
{
```

```
    PANE,their_gbutn,1,0,30,77,77,126,72,114,0,GTAP,GTAP1,4700,1,0
}
PANEL,theirs_both,1,1,13,4,77,30,81,219,0,0
{
    PANE,their_mbtn,1,0,30,78,82,35,72,94,0,ARRO,ARRO1,4700,1,0
    PANE,their_gbtn,1,0,30,79,82,131,72,114,0,GTAP,GTAP1,4800,1,0
}
PANEL,yours_all3,1,1,13,5,453,30,110,372,0,1
{
    PANE,yours_mbtn,1,0,30,80,472,35,72,94,0,ARRO,ARRO1,4700,1,0
    PANE,yours_gbtn,1,0,30,81,472,131,72,114,0,GTAP,GTAP1,4800,1,0
    PANE,your_dress,1,0,30,82,458,247,104,150,0,FRENCHF,(null),4700,276,0
}
PANEL,PMoods,1,1,10,2
{
    PANEL,PMoods_Keypad1,1,0,11,8,0,0,640,480,4,0
    {
        PANE,pad81_0,1,0,32,0,56,45,ARRO,ARRO1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_1,1,0,32,1,132,45,ANXC,ANXC1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_2,1,0,32,2,208,45,NERV,NERV1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_3,1,0,32,3,284,45,SLY,SLY1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_4,1,0,32,4,360,45,EXAS,EXAS1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_5,1,0,32,5,436,45,WRY,WRY1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_6,1,0,32,6,512,45,HAPP,HAPP1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_7,1,0,32,7,56,144,GUIL,GUIL1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_8,1,0,32,8,132,144,CONF,CONF1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_9,1,0,32,9,208,144,RUDE,RUDE1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_10,1,0,32,10,284,144,INNO,INNO1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_11,1,0,32,11,360,144,CROS,CROS1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_12,1,0,32,12,436,144,DETE,DETE1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_13,1,0,32,13,512,144,CURI,CURI1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_14,1,0,32,14,56,243,IRRI,IRRI1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_15,1,0,32,15,132,243,MISC,MISC1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_16,1,0,32,16,208,243,DISB,DISB1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_17,1,0,32,17,284,243,SAD,SAD1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_18,1,0,32,18,360,243,DISG,DISG1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_19,1,0,32,19,436,243,GRIM,GRIM1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_20,1,0,32,20,512,243,SURP,SURP1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_21,1,0,32,21,56,342,DISM,DISM1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_22,1,0,32,22,132,342,SHOC,SHOC1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_23,1,0,32,23,208,342,SHY,SHY1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_24,1,0,32,24,284,342,SURL,SURL1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_25,1,0,32,25,360,342,CHEE,CHEE1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_26,1,0,32,26,436,342,INDI,INDI1,4700,210,0,0,74,72,20,1,5
        PANE,pad81_27,1,0,30,83,512,364,36,36,2,ESC32,ESC321,5000,208,0
        PANE,pad81_28,1,0,30,84,512,400,36,36,2,NOTE32,NOTE321,4F00,150,0
        PANE,pad81_29,1,0,30,85,549,364,40,72,4,HELP,HELP1,7800,40302,0
    }
}
PANEL,PGestures,1,1,10,3
{
    PANEL,PGestures_Keypad1,1,0,11,9,0,0,640,480,4,0
    {
        PANE,pad82_0,1,0,32,27,56,45,GKISS,GKISS1,4700,240,0,0,84,72,30,5,1
```

```
            PANE,pad82_1,1,0,32,28,132,45,GCROS,GCROS1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_2,1,0,32,29,208,45,GNOSE,GNOSE1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_3,1,0,32,30,284,45,GPURS,GPURS1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_4,1,0,32,31,360,45,GCHEK,GCHEK1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_5,1,0,32,32,436,45,GEYEL,GEYEL1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_6,1,0,32,33,512,45,GJERK,GJERK1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_7,1,0,32,34,56,164,GFLIK,GFLIK1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_8,1,0,32,35,132,164,GRING,GRING1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_9,1,0,32,36,208,164,GHORN,GHORN1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_10,1,0,32,37,284,164,GHORZ,GHORZ1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_11,1,0,32,38,360,164,GFIG,GFIG1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_12,1,0,32,39,436,164,GTOSS,GTOSS1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_13,1,0,32,40,512,164,GPALM,GPALM1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_14,1,0,32,41,56,283,GCHIN,GCHIN1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_15,1,0,32,42,132,283,GSTRK,GSTRK1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_16,1,0,32,43,208,283,GTHUM,GTHUM1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_17,1,0,32,44,284,283,GTETH,GTETH1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_18,1,0,32,45,360,283,GEAR,GEAR1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_19,1,0,32,46,436,283,GTAP,GTAP1,4700,240,0,0,84,72,30,5,1
            PANE,pad82_20,1,0,30,86,512,325,36,36,2,ESC32,ESC321,5000,239,0
            PANE,pad82_21,1,0,30,87,512,361,36,36,2,NOTE32,NOTE321,4F00,150,0
            PANE,pad82_22,1,0,30,88,549,325,40,72,4,HELP,HELP1,7800,40302,0
        }
    }
    PANEL,PDress,1,1,10,5
    {
        PANEL,PDress_Keypad1,1,0,11,11,0,0,640,480,4,0
        {
            PANE,pad83_0,1,0,32,47,10,9,KOORIM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_1,1,0,32,48,114,9,JEWISHM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_2,1,0,32,49,218,9,DUTCHF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_3,1,0,32,50,322,9,BRITM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_4,1,0,32,51,426,9,SCOTF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_5,1,0,32,52,530,9,FRENCHF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_6,1,0,32,53,10,162,GERMANM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_7,1,0,32,54,114,162,VIETNAMF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_8,1,0,32,55,218,162,SCOTM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_9,1,0,32,56,322,162,GREEK1,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_10,1,0,32,57,426,162,SIHKM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_11,1,0,32,58,530,162,CHINESEM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_12,1,0,32,59,10,315,GREEK2,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_13,1,0,32,60,114,315,MUSLEMF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_14,1,0,32,61,218,315,ITALIANM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_15,1,0,32,62,322,315,INDIANF,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_16,1,0,32,63,426,315,WESTERNM,(null),4700,271,0,0,116,100,34,4,6
            PANE,pad83_17,1,0,30,89,554,394,36,36,2,ESC32,ESC321,5000,270,0
            PANE,pad83_18,1,0,30,90,554,430,36,36,2,NOTE32,NOTE321,4F00,150,0
            PANE,pad83_19,1,0,30,91,591,394,40,72,4,HELP,HELP1,7800,40302,0
        }
    }
    PANEL,Geopol,1,1,53,0
    {
        PANE,h3_0,1,0,31,29,0,0,640,480,3B00,9999
    }
```

```
}
PANEL,globe,1,0,42,0,1,0,0
{
    PANE,s2_0,1,0,41,0,GLOBE,447,387,180,64,64,0,(null),1
    PANE,s2_1,1,0,41,1,GLOBE1,447,387,180,64,64,0,(null),1
    PANE,s2_2,1,0,41,2,GLOBE2,447,387,180,64,64,0,(null),1
}
PANEL,fan,1,0,42,1,1,1,0
{
    PANE,s1_0,0,0,41,3,FAN13,59,80,240,40,43,0,(null),1
    PANE,s1_1,0,0,41,4,FAN14,59,80,240,40,43,0,(null),1
    PANE,s1_2,0,0,41,5,FAN15,59,80,240,40,43,0,(null),1
    PANE,s1_3,0,0,41,6,FAN16,59,80,240,40,43,0,(null),1
    PANE,s1_4,0,0,41,7,PLANT02,159,73,10,56,69,0,(null),1
    PANE,s1_5,0,0,41,8,FAN17,59,80,230,40,43,0,(null),1
    PANE,s1_6,0,0,41,9,PLANT03,159,73,10,56,69,0,(null),1
    PANE,s1_7,0,0,41,10,FAN18,59,80,230,40,43,0,(null),1
    PANE,s1_8,0,0,41,11,PLANT02,159,73,10,56,69,0,(null),1
    PANE,s1_9,0,0,41,12,FAN19,59,80,230,40,43,0,(null),1
    PANE,s1_10,0,0,41,13,PLANT03,159,73,10,56,69,0,(null),1
    PANE,s1_11,0,0,41,14,FAN18,59,80,230,40,43,0,(null),1
    PANE,s1_12,0,0,41,15,PLANT02,159,73,10,56,69,0,(null),1
    PANE,s1_13,0,0,41,16,FAN17,59,80,230,40,43,0,(null),1
    PANE,s1_14,0,0,41,17,PLANT01,159,73,10,56,69,0,(null),1
    PANE,s1_15,0,0,41,18,FAN16,59,80,230,40,43,0,(null),1
    PANE,s1_16,0,0,41,19,FAN15,59,80,240,40,43,0,(null),1
    PANE,s1_17,0,0,41,20,FAN14,59,80,240,40,43,0,(null),1
    PANE,s1_18,0,0,41,21,FAN13,59,80,240,40,43,0,(null),1
    PANE,s1_19,0,0,41,22,FAN12,59,80,240,40,43,0,(null),1
    PANE,s1_20,0,0,41,23,FAN11,59,80,240,40,43,0,(null),1
    PANE,s1_21,0,0,41,24,FAN10,59,80,240,40,43,0,(null),1
    PANE,s1_22,0,0,41,25,FAN09,59,80,240,40,43,0,(null),1
    PANE,s1_23,0,0,41,26,FAN08,59,80,240,40,43,0,(null),1
    PANE,s1_24,0,0,41,27,FAN07,59,80,240,40,43,0,(null),1
    PANE,s1_25,0,0,41,28,FAN06,59,80,120,40,43,0,(null),1
    PANE,s1_26,0,0,41,29,SCARF01,6,110,0,16,42,0,(null),1
    PANE,s1_27,0,0,41,30,FAN05,59,80,120,40,43,0,(null),1
    PANE,s1_28,0,0,41,31,FAN04,59,80,120,40,43,0,(null),1
    PANE,s1_29,0,0,41,32,SCARF02,6,110,0,16,42,0,(null),1
    PANE,s1_30,0,0,41,33,FAN03,59,80,120,40,43,0,(null),1
    PANE,s1_31,0,0,41,34,FAN02,59,80,120,40,43,0,(null),1
    PANE,s1_32,0,0,41,35,SCARF03,6,110,0,16,42,0,(null),1
    PANE,s1_33,0,0,41,36,FAN01,59,80,160,40,43,0,(null),1
    PANE,s1_34,0,0,41,37,FAN00,59,80,240,40,43,0,(null),1
    PANE,s1_35,0,0,41,38,SCARF04,6,110,0,16,42,0,(null),1
    PANE,s1_36,0,0,41,39,FAN01,59,80,160,40,43,0,(null),1
    PANE,s1_37,0,0,41,40,FAN02,59,80,120,40,43,0,(null),1
    PANE,s1_38,0,0,41,41,SCARF05,6,110,0,16,42,0,(null),1
    PANE,s1_39,0,0,41,42,FAN03,59,80,120,40,43,0,(null),1
    PANE,s1_40,0,0,41,43,FAN04,59,80,120,40,43,0,(null),1
    PANE,s1_41,0,0,41,44,SCARF06,6,110,0,16,42,0,(null),1
    PANE,s1_42,0,0,41,45,FAN05,59,80,120,40,43,0,(null),1
    PANE,s1_43,0,0,41,46,FAN06,59,80,120,40,43,0,(null),1
    PANE,s1_44,0,0,41,47,SCARF07,6,110,0,16,42,0,(null),1
```



```
    PANE,s1_45,0,0,41,48,FAN07,59,80,120,40,43,0,(null),1
    PANE,s1_46,0,0,41,49,FAN08,59,80,120,40,43,0,(null),1
    PANE,s1_47,0,0,41,50,SCARF08,6,110,0,16,42,0,(null),1
    PANE,s1_48,0,0,41,51,FAN09,59,80,240,40,43,0,(null),1
    PANE,s1_49,0,0,41,52,FAN10,59,80,240,40,43,0,(null),1
}
PANEL,title,1,0,42,2,0,1,0
{
    PANE,s3_0,1,0,41,56,KIDS01,320,200,500,64,64,0,(null),1
    PANE,s3_1,1,0,41,57,KIDS02,320,200,400,64,64,0,(null),1
    PANE,s3_2,1,0,41,58,KIDS03,320,200,300,64,64,0,(null),1
    PANE,s3_3,1,0,41,59,KIDS04,320,200,500,64,64,0,(null),1
    PANE,s3_4,1,0,41,60,KIDS05,320,200,300,64,64,0,(null),1
    PANE,s3_5,1,0,41,61,KIDS06,320,200,400,64,64,0,(null),1
    PANE,s3_6,1,0,41,62,KIDS07,320,200,500,64,64,0,(null),1
    PANE,s3_7,1,0,41,63,KIDS08,320,200,500,64,64,0,(null),1
    PANE,s3_8,1,0,41,64,KIDS09,320,200,500,64,64,0,(null),1
    PANE,s3_9,1,0,41,65,KIDS99,344,217,500,64,64,0,(null),1
    PANE,s3_10,1,0,41,66,SUB0,193,256,500,64,64,0,(null),1
    PANE,s3_11,1,0,41,67,FAM0,290,273,500,64,64,0,(null),1
    PANE,s3_12,1,0,41,68,COPYR,225,338,500,64,64,0,(null),1
}
PANEL,quill,1,0,42,3,1,1,0
{
    PANE,s4_0,0,0,41,69,SKULL01,215,20,220,30,30,0,(null),1
    PANE,s4_1,0,0,41,70,SKULL02,215,20,420,30,30,0,(null),1
    PANE,s4_2,0,0,41,71,SKULL03,215,20,220,30,30,0,(null),1
    PANE,s4_3,0,0,41,72,SKULL04,215,20,220,30,30,0,(null),1
    PANE,s4_4,0,0,41,73,SKULL05,215,20,220,30,30,0,(null),1
    PANE,s4_5,0,0,41,74,SKULL06,215,20,220,30,30,0,(null),1
    PANE,s4_6,0,0,41,75,SKULL07,215,20,220,30,30,0,(null),1
    PANE,s4_7,0,0,41,76,SKULL08,215,20,220,30,30,0,(null),1
    PANE,s4_8,0,0,41,77,HEART01,295,51,0,30,30,0,(null),1
    PANE,s4_9,0,0,41,78,SKULL09,215,20,320,30,30,0,(null),1
    PANE,s4_10,0,0,41,79,HEART02,295,51,0,30,30,0,(null),1
    PANE,s4_11,0,0,41,80,SKULL08,215,20,120,30,30,0,(null),1
    PANE,s4_12,0,0,41,81,SKULL07,215,20,120,30,30,0,(null),1
    PANE,s4_13,0,0,41,82,HEART03,295,51,0,30,30,0,(null),1
    PANE,s4_14,0,0,41,83,SKULL06,215,20,120,30,30,0,(null),1
    PANE,s4_15,0,0,41,84,SKULL05,215,20,120,30,30,0,(null),1
    PANE,s4_16,0,0,41,85,HEART04,295,51,0,30,30,0,(null),1
    PANE,s4_17,0,0,41,86,SKULL04,215,20,120,30,30,0,(null),1
    PANE,s4_18,0,0,41,87,SKULL03,215,20,120,30,30,0,(null),1
    PANE,s4_19,0,0,41,88,HEART05,295,51,0,30,30,0,(null),1
    PANE,s4_20,0,0,41,89,HEAD01,469,153,0,30,30,0,(null),1
    PANE,s4_21,0,0,41,90,SKULL01,215,20,120,30,30,0,(null),1
    PANE,s4_22,0,0,41,91,SKULL00,215,20,120,30,30,0,(null),1
    PANE,s4_23,0,0,41,92,HEART00,295,51,0,30,30,0,(null),1
    PANE,s4_24,0,0,41,93,QUILL01,446,170,400,30,30,0,(null),1
    PANE,s4_25,0,0,41,94,HEART01,295,51,0,30,30,0,(null),1
    PANE,s4_26,0,0,41,95,ANGEL01,146,27,0,30,30,0,(null),1
    PANE,s4_27,0,0,41,96,QUILL02,446,170,200,30,30,0,(null),1
    PANE,s4_28,0,0,41,97,HEART00,295,51,0,30,30,0,(null),1
    PANE,s4_29,0,0,41,98,ANGEL02,146,27,0,30,30,0,(null),1
```

```
    PANE,s4_30,0,0,41,99,QUILL03,446,170,400,30,30,0,(null),1
    PANE,s4_31,0,0,41,100,ANGEL03,146,27,0,30,30,0,(null),1
    PANE,s4_32,0,0,41,101,QUILL02,446,170,200,30,30,0,(null),1
    PANE,s4_33,0,0,41,102,ANGEL01,146,27,0,30,30,0,(null),1
    PANE,s4_34,0,0,41,103,QUILL01,446,170,400,30,30,0,(null),1
    PANE,s4_35,0,0,41,104,ANGEL02,146,27,0,30,30,0,(null),1
    PANE,s4_36,0,0,41,105,QUILL00,446,170,200,30,30,0,(null),1
    PANE,s4_37,0,0,41,106,ANGEL03,146,27,0,30,30,0,(null),1
    PANE,s4_38,0,0,41,107,QUILL01,446,170,400,30,30,0,(null),1
    PANE,s4_39,0,0,41,108,ANGEL00,146,27,0,30,30,0,(null),1
    PANE,s4_40,0,0,41,109,HEAD02,469,153,0,30,30,0,(null),1
    PANE,s4_41,0,0,41,110,QUILL02,446,170,200,30,30,0,(null),1
    PANE,s4_42,0,0,41,111,ANGEL01,146,27,0,30,30,0,(null),1
    PANE,s4_43,0,0,41,112,QUILL03,446,170,400,30,30,0,(null),1
    PANE,s4_44,0,0,41,113,ANGEL02,146,27,0,30,30,0,(null),1
    PANE,s4_45,0,0,41,114,QUILL02,446,170,200,30,30,0,(null),1
    PANE,s4_46,0,0,41,115,ANGEL00,146,27,0,30,30,0,(null),1
    PANE,s4_47,0,0,41,116,QUILL01,446,170,400,30,30,0,(null),1
    PANE,s4_48,0,0,41,117,QUILL00,446,170,200,30,30,0,(null),1
    PANE,s4_49,0,0,41,118,HEAD00,469,153,0,30,30,0,(null),1
}
PANEL,kooka,1,0,42,4,1,1,0
{
    PANE,s5_0,0,0,41,119,GLASS01,454,190,100,30,30,0,(null),1
    PANE,s5_1,0,0,41,120,KOOKA02,94,30,203,30,30,0,(null),1
    PANE,s5_2,0,0,41,121,KOOKA03,94,30,305,30,30,0,(null),1
    PANE,s5_3,0,0,41,122,GLASS02,454,190,0,30,30,0,(null),1
    PANE,s5_4,0,0,41,123,KOOKA04,94,30,305,30,30,0,(null),1
    PANE,s5_5,0,0,41,124,KOOKA05,94,30,305,30,30,0,(null),1
    PANE,s5_6,0,0,41,125,GLASS03,454,190,0,30,30,0,(null),1
    PANE,s5_7,0,0,41,126,KOOKA04,94,30,305,30,30,0,(null),1
    PANE,s5_8,0,0,41,127,KOOKA05,94,30,305,30,30,0,(null),1
    PANE,s5_9,0,0,41,128,KOOKA06,94,30,204,30,30,0,(null),1
    PANE,s5_10,0,0,41,129,KOOKA01,94,30,204,30,30,0,(null),1
    PANE,s5_11,0,0,41,130,KOOKA06,94,30,204,30,30,0,(null),1
    PANE,s5_12,0,0,41,131,KOOKA01,94,30,204,30,30,0,(null),1
    PANE,s5_13,0,0,41,132,GLASS04,454,190,0,30,30,0,(null),1
    PANE,s5_14,0,0,41,133,KOOKA04,94,30,305,30,30,0,(null),1
    PANE,s5_15,0,0,41,134,KOOKA05,94,30,305,30,30,0,(null),1
    PANE,s5_16,0,0,41,135,KOOKA04,94,30,305,30,30,0,(null),1
    PANE,s5_17,0,0,41,136,GLASS05,454,190,0,30,30,0,(null),1
    PANE,s5_18,0,0,41,137,KOOKA03,94,30,305,30,30,0,(null),1
    PANE,s5_19,0,0,41,138,KOOKA01,94,30,203,30,30,0,(null),1
}
PANEL,mobile,1,0,42,5,1,1,0
{
    PANE,s6_0,0,0,41,139,MOBIL01,154,90,600,30,30,0,(null),1
    PANE,s6_1,0,0,41,140,MOBIL02,154,90,400,30,30,0,(null),1
    PANE,s6_2,0,0,41,141,MOBILB00,453,43,0,30,30,0,(null),1
    PANE,s6_3,0,0,41,142,MOBIL03,154,90,400,30,30,0,(null),1
    PANE,s6_4,0,0,41,143,MOBILB01,453,43,0,30,30,0,(null),1
    PANE,s6_5,0,0,41,144,GIRL00,406,219,0,30,30,0,(null),1
    PANE,s6_6,0,0,41,145,BIRD01,126,138,0,30,30,0,(null),1
    PANE,s6_7,0,0,41,146,MOBIL04,154,90,400,30,30,0,(null),1
```

```
    PANE,s6_8,0,0,41,147,BOY00,258,216,0,30,30,0,(null),1
    PANE,s6_9,0,0,41,148,BIRD00,126,138,0,30,30,0,(null),1
    PANE,s6_10,0,0,41,149,MOBILB02,453,43,0,30,30,0,(null),1
    PANE,s6_11,0,0,41,150,MOBIL05,154,90,400,30,30,0,(null),1
    PANE,s6_12,0,0,41,151,BIRD00,126,138,0,30,30,0,(null),1
    PANE,s6_13,0,0,41,152,GIRL01,406,219,0,30,30,0,(null),1
    PANE,s6_14,0,0,41,153,BOY01,258,216,0,30,30,0,(null),1
    PANE,s6_15,0,0,41,154,MOBILB03,453,43,0,30,30,0,(null),1
    PANE,s6_16,0,0,41,155,HAND01,278,126,0,30,30,0,(null),1
    PANE,s6_17,0,0,41,156,MOBIL06,154,90,400,30,30,0,(null),1
    PANE,s6_18,0,0,41,157,HAND00,278,126,0,30,30,0,(null),1
    PANE,s6_19,0,0,41,158,MOBILB04,453,43,0,30,30,0,(null),1
    PANE,s6_20,0,0,41,159,BOY00,258,216,0,30,30,0,(null),1
    PANE,s6_21,0,0,41,160,GIRL00,406,219,0,30,30,0,(null),1
    PANE,s6_22,0,0,41,161,MOBIL07,154,90,400,30,30,0,(null),1
    PANE,s6_23,0,0,41,162,HAND01,278,126,0,30,30,0,(null),1
    PANE,s6_24,0,0,41,163,MOBILB03,453,43,0,30,30,0,(null),1
    PANE,s6_25,0,0,41,164,GIRL01,406,219,0,30,30,0,(null),1
    PANE,s6_26,0,0,41,165,MOBIL08,154,90,600,30,30,0,(null),1
    PANE,s6_27,0,0,41,166,BOY01,258,216,0,30,30,0,(null),1
    PANE,s6_28,0,0,41,167,MOBILB02,453,43,0,30,30,0,(null),1
    PANE,s6_29,0,0,41,168,GIRL00,406,219,0,30,30,0,(null),1
    PANE,s6_30,0,0,41,169,MOBIL09,154,90,600,30,30,0,(null),1
    PANE,s6_31,0,0,41,170,MOBILB01,453,43,0,30,30,0,(null),1
    PANE,s6_32,0,0,41,171,MOBIL08,154,90,600,30,30,0,(null),1
    PANE,s6_33,0,0,41,172,BOY00,258,216,0,30,30,0,(null),1
    PANE,s6_34,0,0,41,173,MOBILB00,453,43,0,30,30,0,(null),1
    PANE,s6_35,0,0,41,174,MOBIL07,154,90,400,30,30,0,(null),1
    PANE,s6_36,0,0,41,175,MOBILB01,453,43,0,30,30,0,(null),1
    PANE,s6_37,0,0,41,176,MOBIL06,154,90,400,30,30,0,(null),1
    PANE,s6_38,0,0,41,177,BOY01,258,216,0,30,30,0,(null),1
    PANE,s6_39,0,0,41,178,MOBILB02,453,43,0,30,30,0,(null),1
    PANE,s6_40,0,0,41,179,MOBIL05,154,90,400,30,30,0,(null),1
    PANE,s6_41,0,0,41,180,MOBILB03,453,43,0,30,30,0,(null),1
    PANE,s6_42,0,0,41,181,MOBIL04,154,90,400,30,30,0,(null),1
    PANE,s6_43,0,0,41,182,BOY00,258,216,0,30,30,0,(null),1
    PANE,s6_44,0,0,41,183,MOBILB04,453,43,0,30,30,0,(null),1
    PANE,s6_45,0,0,41,184,MOBIL03,154,90,400,30,30,0,(null),1
    PANE,s6_46,0,0,41,185,MOBIL02,154,90,400,30,30,0,(null),1
    PANE,s6_47,0,0,41,186,GIRL01,406,219,0,30,30,0,(null),1
    PANE,s6_48,0,0,41,187,MOBIL01,154,90,600,30,30,0,(null),1
    PANE,s6_49,0,0,41,188,MOBIL00,154,90,600,30,30,0,(null),1
}
PANEL,post_kids,1,0,42,6,1,1,0
{
    PANE,s7_0,0,0,41,189,PAT01,433,152,250,30,30,0,(null),1
    PANE,s7_1,0,0,41,190,KID01,322,214,250,30,30,0,(null),1
    PANE,s7_2,0,0,41,191,PAT02,433,152,500,30,30,0,(null),1
    PANE,s7_3,0,0,41,192,PAT01,433,152,500,30,30,0,(null),1
    PANE,s7_4,0,0,41,193,PAT02,433,152,500,30,30,0,(null),1
    PANE,s7_5,0,0,41,194,PAT01,433,152,500,30,30,0,(null),1
    PANE,s7_6,0,0,41,195,PAT03,433,152,500,30,30,0,(null),1
    PANE,s7_7,0,0,41,196,PAT01,433,152,500,30,30,0,(null),1
    PANE,s7_8,0,0,41,197,PAT04,433,152,500,30,30,0,(null),1
```

```
            PANE,s7_9,0,0,41,198,PAT01,433,152,500,30,30,0,(null),1
            PANE,s7_10,0,0,41,199,PAT02,433,152,500,30,30,0,(null),1
            PANE,s7_11,0,0,41,200,PAT01,433,152,500,30,30,0,(null),1
            PANE,s7_12,0,0,41,201,PAT02,433,152,500,30,30,0,(null),1
            PANE,s7_13,0,0,41,202,PAT01,433,152,500,30,30,0,(null),1
            PANE,s7_14,0,0,41,203,PAT03,433,152,500,30,30,0,(null),1
            PANE,s7_15,0,0,41,204,PAT01,433,152,500,30,30,0,(null),1
            PANE,s7_16,0,0,41,205,PAT04,433,152,250,30,30,0,(null),1
            PANE,s7_17,0,0,41,206,KID00,322,214,250,30,30,0,(null),1
            PANE,s7_18,0,0,41,207,PAT00,433,152,500,30,30,0,(null),1
            PANE,s7_19,0,0,41,208,PAT00,433,152,500,30,30,0,(null),1
        }
    }
}
```


Appendix B - The Complex Digital Self depicted in Figure 6.1, rendered here in XML file: DigitalSelf.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE whole-agent SYSTEM "Shadowboard.dtd" >
<whole-agent global-id="IP:203.31.192.82" global-name="Deepthought"
                                                    global-type="Shadowboard">
  <aware-ego-agent>
    <sub-agent a-delegator="true" agent-id="AE1" agent-name="Brady"
          agent-type="AwareEgo" disowned="false" external-agent_id="A1" rank="1">
      <description>This is the Aware Ego agent.</description>
      <envelope-of-capability>
        <sub-agent a-delegator="true" agent-id="DM1" agent-name="DecisionMakers"
              agent-type="DecisionMaker" disowned="false" rank="1">
          <description/>
          <envelope-of-capability>
            <archetype>
              <sub-agent agent-id="DM2" agent-name="ExecutiveDirector"
                   agent-type="DecisionMaker" disowned="false" rank="1"/>
            </archetype>
            <sub-agent agent-id="DM3" agent-name="Controller"
                 agent-type="DecisionMaker" disowned="false" rank="2"/>
            <sub-agent agent-id="DM4" agent-name="Judge" agent-type="DecisionMaker"
                 rank="3"/>
            <reactive-agent>
              <sub-agent agent-id="RV1" agent-name="Dictator" agent-type="Reactive"
                 rank="4"/>
            </reactive-agent>
          </envelope-of-capability>
        </sub-agent>
        <sub-agent a-delegator="true" agent-name="Managers" disowned="false">
          <description/>
          <envelope-of-capability>
            <archetype>
              <sub-agent agent-id="MR1" agent-name="BenevolentManager"
                   agent-type="Manager" disowned="false"/>
            </archetype>
            <sub-agent agent-id="MR2" agent-name="ConciliatoryManager"
                                                        agent-type="Manager"/>
            <sub-agent agent-id="MR3" agent-name="Planner" agent-type="Manager"/>
            <sub-agent agent-id="MR4" agent-name="Scheduler" agent-type="Manager"/>
            <sub-agent agent-id="MR5" agent-name="Coordinator" agent-type="Manager"/>
            <sub-agent agent-id="MR6" agent-name="Recycler" agent-type="Manager"/>
            <reactive-agent>
```



```
        <sub-agent agent-id="RV2" agent-name="DecisiveManager"
             agent-type="Reactive" disowned="false"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Protectors" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="SafetyOfficer"/>
      </archetype>
      <sub-agent agent-name="Defender" disowned="false"/>
      <sub-agent agent-name="RiskAnalyst"/>
      <sub-agent agent-name="Environmentalist"/>
      <sub-agent agent-name="Pacifier"/>
      <sub-agent agent-name="Doctor"/>
      <reactive-agent>
        <sub-agent agent-name="ExitStrategist"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Servers" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="SelflessPleaser"/>
      </archetype>
      <sub-agent agent-name="ServiceProvider"/>
      <sub-agent agent-name="Networker"/>
      <sub-agent agent-name="Communicator">
        <description/>
        <envelope-of-capability>
          <sub-agent a-delegator="false" agent-id="A3" agent-name="MailMan"
               agent-type="PersonalAssistant" disowned="false"
               external-agent_id="A3" rank="2">
            <description>This is an email filtering agent.</description>
          </sub-agent>
        </envelope-of-capability>
      </sub-agent>
      <sub-agent agent-name="Marketer"/>
      <reactive-agent>
        <sub-agent agent-name="SelfServer"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Initiators" disowned="false">
```

```
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="Pusher" disowned="false"/>
      </archetype>
      <sub-agent agent-name="SuccessSeeker"/>
      <sub-agent agent-name="Wooer" disowned="false"/>
      <sub-agent agent-name="ResourceMaster"/>
      <sub-agent agent-name="TroubleShooter"/>
      <reactive-agent>
        <sub-agent agent-name="DoLittle"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Critic" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="Perfectionist"/>
      </archetype>
      <sub-agent agent-name="Editor"/>
      <sub-agent agent-name="QualityControl"/>
      <sub-agent agent-name="Doubter"/>
      <reactive-agent>
        <sub-agent agent-name="Cynic"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Intuitives" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="Seer"/>
      </archetype>
      <sub-agent agent-name="MoodSensor"/>
      <sub-agent agent-name="PatternFinder"/>
      <sub-agent agent-name="Dreamer"/>
      <reactive-agent>
        <sub-agent agent-name="DayDreamer"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Adventurer" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
```



```
        <sub-agent agent-name="Explorer"/>
      </archetype>
      <sub-agent agent-name="RiskTaker"/>
      <sub-agent agent-name="Traveler"/>
      <sub-agent agent-name="Vacationer"/>
      <reactive-agent>
        <sub-agent agent-name="Lazybones"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="KnowledgeSeeker" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="KnowledgeWorker" disowned="false"/>
      </archetype>
      <sub-agent agent-name="ConceptLearner"/>
      <sub-agent agent-name="InformationOfficer">
        <description/>
        <envelope-of-capability>
          <archetype>
            <sub-agent a-delegator="false" agent-id="A2" agent-name="InfoMaster"
                agent-type="Information" disowned="true"
                external-agent_id="A10" rank="1">
              <description>This is the archetypal sub-agent of an
                          envelope-of-capability.
              </description>
              <sub-persona animation-name="guru.mpg" current-status="DEFAULT"
                          event-mask="128" general-expression="happy"
                          icon-name="guru.gif" options-mask="DEFAULT"
                          persona-id="A101" persona-name="Hal"
                          persona-type="MPEG4"/>
              <external-agent acl="KQML" agent-id="A10" agent-name="TheInfoGuru"
                             agent-type="PersonalAssistant"
                             content-language="PROLOG" ontology="Informatics"/>
            </sub-agent>
          </archetype>
          <sub-agent a-delegator="false" agent-id="A4" agent-name="InfoMan"
                     agent-type="Information" disowned="false"
                     external-agent_id="A4" rank="3">
            <description>This is a specialized internet search agent.
            </description>
          </sub-agent>
          <reactive-agent>
            <sub-agent a-delegator="false" agent-id="A5" agent-name="Speedy"
                       agent-type="Reactive" disowned="false"
```

```
                    external-agent_id="A5" rank="99">
            <description/>
          </sub-agent>
        </reactive-agent>
      </envelope-of-capability>
    </sub-agent>
    <sub-agent agent-name="DataMiner"/>
    <reactive-agent>
      <sub-agent agent-name="RandomGenerator"/>
    </reactive-agent>
  </envelope-of-capability>
</sub-agent>
<sub-agent a-delegator="true" agent-name="Logician" disowned="false">
  <description/>
  <envelope-of-capability>
    <archetype>
      <sub-agent agent-name="Rationist" disowned="false"/>
    </archetype>
    <sub-agent agent-name="Statistician"/>
    <sub-agent agent-name="BusinessAnalyst"/>
    <sub-agent agent-name="Diagnostic"/>
    <sub-agent agent-name="Lawyer"/>
    <reactive-agent>
      <sub-agent agent-name="Dogmatist"/>
    </reactive-agent>
  </envelope-of-capability>
</sub-agent>
<sub-agent a-delegator="true" agent-name="Children" disowned="false">
  <description/>
  <envelope-of-capability>
    <archetype>
      <sub-agent agent-name="Player" disowned="false"/>
    </archetype>
    <sub-agent agent-name="FunSeeker" disowned="false"/>
    <sub-agent agent-name="Inventor"/>
    <sub-agent agent-name="VitalChild"/>
    <sub-agent agent-name="VulnerableChild"/>
    <reactive-agent>
      <sub-agent agent-type="TheFool"/>
    </reactive-agent>
  </envelope-of-capability>
</sub-agent>
<sub-agent a-delegator="true" agent-name="Teachers" disowned="false">
  <description/>
  <envelope-of-capability>
    <archetype>
```



```
        <sub-agent agent-name="Master" disowned="false"/>
      </archetype>
      <sub-agent agent-name="Ethicist"/>
      <sub-agent agent-name="Linguist"/>
      <sub-agent agent-name="Mathematician"/>
      <sub-agent agent-name="Biologist"/>
      <sub-agent agent-name="Economist"/>
      <reactive-agent>
        <sub-agent agent-name="Apprentice"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-type="Engineers" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent a-delegator="true" agent-name="Constructor"
                   disowned="false"/>
      </archetype>
      <sub-agent agent-name="MaterialEngineer"/>
      <sub-agent agent-name="MechanicalEngineer"/>
      <sub-agent agent-name="StructuralEngineer"/>
      <sub-agent agent-name="ChemicalEngineer"/>
      <reactive-agent>
        <sub-agent agent-name="Outsourcer"/>
      </reactive-agent>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="Artists" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent/>
      </archetype>
      <sub-agent agent-name="Wordsmith"/>
      <sub-agent agEnt-name="VisualArtist"/>
      <sub-agent agent-name="PerformanceArtist"/>
    </envelope-of-capability>
  </sub-agent>
  <sub-agent a-delegator="true" agent-name="SpiritualAdvisor" disowned="false">
    <description/>
    <envelope-of-capability>
      <archetype>
        <sub-agent agent-name="WiseOne" disowned="false"/>
      </archetype>
      <sub-agent agent-name="Mentor"/>
```

```
            <sub-agent agent-name="Humanist"/>
            <sub-agent agent-name="Faithful"/>
          </envelope-of-capability>
        </sub-agent>
        <sub-agent a-delegator="true" agent-name="Spacial" disowned="false">
          <description/>
          <envelope-of-capability>
            <sub-agent agent-name="Geographer"/>
            <sub-agent agent-name="3D-Surveyor"/>
            <sub-agent agent-name="2D-Surveyor"/>
          </envelope-of-capability>
        </sub-agent>
        <sub-agent a-delegator="true" agent-name="Intrapersonal" disowned="false">
          <description/>
          <envelope-of-capability>
            <sub-agent agent-name="Profiler"/>
            <sub-agent agent-name="RoleKeeper"/>
            <sub-agent agent-name="StandardBearer"/>
          </envelope-of-capability>
        </sub-agent>
      </envelope-of-capability>
    </sub-agent>
  </aware-ego-agent>
  <sub-persona animation-name="urbane.mpg" current-status="DEFAULT" event-mask="128"
               general-expression="confident" icon-name="urbane.gif"
               options-mask="DEFAULT" persona-id="A100" persona-name="Clive"
               persona-type="MPEG4"/>
</whole-agent>
```

