An Animated Pedagogical Agent For Assisting Novice Programmers Within A Desktop Computer Environment

by

Desmond Robert Case BSc, MSc

Staffordshire University

A thesis submitted in partial fulfilment of the requirements of Staffordshire University for the award of the degree of Doctor of Philosophy in Computer Science

September 2012

Abstract

Learning to program for the first time can be a daunting process, fraught with difficulty and setback. The novice learner is faced with learning two skills at the same time each that depends on the other; they are how a program needs to be constructed to solve a problem and how the structures of a program work towards solving a problem. In addition the learner has to develop practical skills such as how to design a solution, how to use the programming development environment, how to recognise errors, how to diagnose their cause and how to successfully correct them. The nature of learning how to program a computer can cause frustration to many and some to disengage before they have a chance to progress. Numerous authorities have observed that novice programmers make the same mistakes and encounter the same problems when learning their first programming language. The learner errors are usually from a fixed set of misconceptions that are easily corrected by experience and with appropriate guidance.

This thesis demonstrates how a virtual animated pedagogical agent, called MRCHIPS, can extend the Beliefs-Desires-Intentions model of agency to provide mentoring and coaching support to novice programmers learning their first programming language, Python. The Cognitive Apprenticeship pedagogy provides the theoretical underpinning of the agent mentoring strategy. Case-Based Reasoning is also used to support MRCHIPS reasoning, coaching and interacting with the learner. The results indicate that in a small controlled study when novice learners are assisted by MRCHIPS they are more productive than those working without the assistance, and are better at problem solving exercises, there are also manifestations of higher of degree of engagement and learning of the language syntax.

Acknowledgements

I would thank my supervisors Professor Bernadette Sharp and Dr Len Noriega for their patience and support. I would also like to thank Dr Peter King for his help and support in the past.

Thanks go to various members of the faculty and students of the Northampton Business School at the University of Northampton for their technical assistance, interest and accommodation for my activities during this research.

I would like to thank and reassure the many good friends and extended family members who must now believe I only appear when the moon is blue, that I am still here, you have always been in my thoughts and that I look forward to the joy of renewing old acquaintances.

Finally I would like to offer a special thank you to the patients, faith and unwavering encouragement of three special women I have woefully neglected these many, many months: my mother, Mrs Avis Case, my sister, Ms Rose Case, and my partner Dr Alison Rudd. I love and cherish you all.

Table of Contents

Figures

Tables

Chapter 1:

Introduction

1.1 The difficulty with learning to program

When a novice programmer first begins to learn a programming language he, or she, often encounters the same problems and makes the same mistakes as others who have learned to program before. The misconceptions, mistakes and errors form a set of knowledge that can be easily corrected by simple guidance or experience and have to be leaned as part of the programming skill. Making mistakes when learning to program is a constructive part of the process, however some learners find the precision required by programming code frustrating and may become disengaged with the process. Despite rich interactive development environments, learners continue to generate errors as they experiment with the language structures and find debug messages unhelpful because of their lack of experience of the significance of error information. During practical sessions a supervisor's task is often to simply call on prior experience to provide guidance and offer reassurance that errors are all part of the development process. Away from supervision some learners can become stuck on a simple error that halts progress and prevents the chance to address other problems. The problems are often as a result of the learner failing to recognise where they have deviated from language syntax or which solution to apply to address a given problem. The problems are often easily fixed when pointed out by a tutor or even a more able peer and this kind of help can occur both within and outside the classroom situation. When the help is provided by a more able peer or the help is provided outside of the classroom situation it can be characterised as mentoring, which is support in the form of a more experienced practitioner sharing knowledge. It may include privileged access to information; it is informal in nature and the subject is driven by the concerns of the learner. A fuller discussion of mentoring is given in section 1.4.1.

When producing a software application it is necessary for the developer to organise a large body of coding and data structures, and to decide how to arrange them in such a way as to provide a solution to a given problem. A programmer is faced with a range of coding options, operators, functions, choices, knowledge representation and data structures and how to represent the above in any particular language. In addition, the programmer must devise a sequence of code for execution and possess enough insight to recognise deficiencies and make corrections. The range of possible options and the dependency between concepts makes learning to write computer programs a challenging task. The programmer must also possess enough insight into the domain of the problem to be able to encode a solution.

Although teaching a subject is primarily concerned with one party imparting knowledge to another, in reality there are other factors that affect how well a learner is able to assimilate and apply new information. Learning is a very social activity relying on relationships, in addition to the subject knowledge, as part of the process. These relationships involve things such as providing encouragement and explanation. Much of the activity of supervising novice programmers is social in that the tutor offers encouragement with often a smaller amount of time on advising corrections to code. Anecdotal observation has found that this level of technical guidance is often sought and given irrespective of the details of the design task being undertaken by the learner.

1.2 Research Aims

This research proposes the use of a pedagogical agent, called MRCHIPS, to provide mentoring support and presented as an animated character for social interaction. The aim of the research is to determine whether the use of an automated and animated pedagogical agent can provide mentoring support to novice programmers as they learn their first programming language.

Hypothesis 1) An intelligent agent with an anthropomorphic interface can provide effective mentoring support to novice programmers learning their first programming language.

The agent would appear as an interactive anthropomorphic entity that would assist the novice to determine and solve programming errors when a human mentor in the form of a tutor might not be available. The idea arose from the observation that students on a business-computing course learning a programming language for the first time would often give up at the first problem they found a challenge, limiting their exposure to later exercises (problem solving with a programming language can be a linear process) and their overall learning experience.

As the students' understanding of the subject was distributed across the range of topics they had encountered, it was difficult to predict which particular problem would impede them. In general some would understand some principles and not others in different ways. The problems would often be relatively minor and the practice of systematically reviewing the work they had produced would be enough to uncover the cause, but as reflection on ones' work is also a skill under development as people learn programming for the first time that technique is not available to novices. The information could be gathered from available documentation and literature but as the students are still impeded by the same errors these options are not taken, or not effective.

Intelligent virtual agents, sometimes called believable agents, life-like agents, synthetic agents or embedded virtual agents are part of the field of Artificial Intelligence research concerned with presenting an anthropomorphic character to represent an underlying cognitive agent in an environment. One of the uses of the virtual agent is to promote greater engagement when interacting with a human user. While this is true of a 3D game environment, is the same also true of the programming environment?

Hypothesis 2) The use of an animated virtual character user interface increases the learner's engagement with problem solving in the programming environment.

To provide the range of capabilities that would allow an agent to monitor, interact, diagnose and provide solutions to the learner it is likely to require an agent architecture that is able to coordinate multiple reasoning techniques, called a cognitive architecture. However most popular agent architectures, such as Beliefs Desires Intensions (BDI), generally support reasoning based around reactive and deliberative planning, which would be required to control the interactivity of the agent, but not enough to provide the domain reasoning of a pedagogical agent. Would it be possible to extend the capabilities of a procedural BDI agent architecture into a cognitive architecture?

Hypothesis 3) The processing capabilities of a procedural BDI agent can be extended to provide the more knowledge based reasoning capabilities of a cognitive agent architecture.

Intelligent Learning Environments (ILE) are based around specially developed software applications that the student must learn to use before transferring the skills to a real-world application environment. Intelligent Tutoring Systems (ITS) and virtual agents also usually reside in their own application environments. As learning to program is a difficult enough task without having to become familiar with multiple tools or environments, the utility of the agent is likely to be greater if the agent worked in the learner's environment and not the other way around.

Hypothesis 4) Agent based reasoning provides a framework to extend knowledge-based systems into existing computing desktop environments and avoid the need to build a specialised learning application environment.

This would test how much of the cognitive agent would have to be adapted to cope with an environment that will not be as accommodating to its requirements as an agent aware environment. This is likely to be similar to the situation faced by many network applications and robotics research, but will allow for examination of cognitive and software solutions respectively. This question also imposes an implied sub constraint that the mentoring agent should not require any special hardware or software above what could be expected on a desktop computer that would be used to learn to program in Python for example. This is a useful guideline for the development of the agent and deployment for evaluation and demonstration.

1.3 Principal contributions of the research

This research brings together a number of areas namely Intelligent Learning Environments (ILE), Intelligent Tutoring Systems (ITS), virtual agents and cognitive agent architectures, which are covered in the literature review chapters. ILE systems are usually based around specially developed software applications the student must learn to use before transferring the skills to a real-world application. As it will be shown in the literature review other research into the use of intelligent virtual agents in teaching has to date provided the environment in which the interaction with the learner takes place. Evidence from this has been able to demonstrate that novice programmers respond positively to interactive learning with animated characters when developing code (Lui and Chan, 2006). The novel approach taken by this research is to avoid the requirement for a custom agent environment and for reasons arising from the pedagogical theory explained in later chapters, the agent operates in the learners' environment of the Windows desktop. The novel approach of MRCHIPS is in its strategy, to allow the novice programmer to continue to work in the pre-existing development environment, to adhere to one of the major principles of the cognitive apprenticeship pedagogy.

MRCHIPS exists as a separate application that because of its unique architecture is able to monitor the user and provide knowledgeable assistance. The agent not only monitors the novice, but makes use of the reasoning of a cognitive architecture to provide expert level analysis of his or her work and provide a character driven interactive response to the user's errors. These capabilities allow the agent to operate where they are not usually found, in a programming environment on the desktop of a conventional computer. If the subject activity is already computer based the advantage of MRCHIPS is that the learner does not have to become familiar with a second application, such as an ILE, in order to learn the first application. In this way MRCHIPS supports a closer adherence to the requirement of the cognitive apprenticeship pedagogy for the novice to work with real world examples, as learning practice is accomplished using real world tools.

1.4 Background

In the opening chapter of the book *Inside Case-Based Reasoning* the authors, Riesbeck and Schank, describe Artificial Intelligence as a "search for the general mechanisms underlying intelligence" (Riesbeck and Schank 1989). Embodied within that view is the concept of computers as an answer-giving device. The idea of an individual being able to present a problem to a computer when facing an unfamiliar situation and to have it provide an answer not only motivates the dreams of science fiction fantasy, as a casual survey of a series such as *Star Trek* would show the purpose of the intelligent computer assistant to provide information to human characters faced with the unfamiliar, as well as exposition to the human viewers at home. But also this model of the intelligent machine advisor has been the goal of real-world research and is increasingly found in the user interface of commonly available software and hardware. The concept of the intelligent computer assistant also has merit when users are charting unfamiliar knowledge and one application area where this has been useful is in education. This research is interested in the utility of agents to assist in the learning of a first programming language.

According to Gulz (2004) educational researchers have observed that novice programmers make the same mistakes and encounter the same problems when first learning a programming language. The learner errors are usually from a fixed set of misconceptions that are easily corrected by experience and with simple guidance. Despite rich interactive development environments, learners continue to generate errors as they experiment with the language structures and find debug messages unhelpful because of their lack of experience of the significance of error information. During practical sessions a supervisor's task is often to simply call on prior experience to offer guidance and offer reassurance that errors are all part of the development process. Computer programming is a skill-based activity that involves problem solving within the constraints imposed by a computer environment. Learning to program is a fairly unique activity; there are few, everyday real world analogies to the activity, programs are constrained by a mathematical concept - logic, rather than an observable physical phenomena and the correctness of code is ultimately mediated by a machine. The difficulty of the task faced by the novice programmer is that when s/he start they have a limited idea of how to produce code to achieve a goal, or solve a problem, and little insight into how the code they produce will be interpreted by the programming language.

1.4.1 Mentoring vs. Tutoring

The agent's operation alongside the learner allows another novel contribution to the research. The agent operates as a mentor towards the learner, as opposed to a traditional tutor of ITS. In a formal sense there is little difference between the terms mentor and tutor. The word Mentor originates from the name of the figure of Greek legend who in his old age was given charge of Telemachus, the son of Odysseus, when the latter went to fight in the Trojan wars. According to the legend Mentor performed his role so well that his name later became the proverbial phrase for a faithful and wise adviser. The term mentor describes a teaching relationship and is a synonym of teacher, as is tutor, counsellor, lecturer, coach, instructor and guru. By convention the different terms for a teacher are used to describe the nature of the participants in the learning process. For example a person might find himself or herself taught by at teacher at school, a tutor when learning the piano, a coach for learning a sport, a lecturer at university, an instructor for driving and a mentor at the start of employment. The Collins dictionary defines the role of a mentor as "a wise and trusted advisor or guide" or "an influential senior sponsor or supporter" (Collins dictionary 1987). While accurate, this definition does not encompass the scope of mentoring, which also implies a protective role. The learner in a mentoring relationship is often called the protégé, which is a French term derived from the Latin meaning "to protect" (Johnson 2007). Johnson (2007) describes the task of mentoring which "… nearly always includes an emotional/interpersonal support dimension. Components of psychosocial support may include affirmation, encouragement, counselling, and friendship", while Landsberg (1996) describes mentoring as "… a role which includes coaching, but also embraces broader counselling and support, such as career counselling, privileged access to information, etc.". So the term mentoring is mainly used where there is an emphasis on a caring aspect of the teaching in varied applications such as social care, personal friendships or employment and career development. Mentoring differs from tutoring in terms of the nature of the relationship between the participants. In a paper for the University of Michigan, Arbor (1999) specifies mentors, among other academic roles, as "… advisers, people with career experience willing to share their knowledge […] tutors, people who give specific feedback on one's performance". Therefore throughout this research the term mentor will be used to mean an advice giver who will support the learner based on experience in pursuit of providing care, while a tutor provides lesson material, assesses the learner's performance and provides specific feedback on progress.

1.5 The research framework

A research framework ensures the correct model is used to evaluate an item of research so results may be placed into an appropriate context to show their worth. As different types of research require different types of research frameworks it also defines the different activities that can be used to produce specific outputs.

1.5.1 The March and Smith framework

The precise origins of the framework as applicable to items of research is unclear from the literature as the term framework is used to cover other such diverse subjects as industrial projects, academic programs, corporate and government initiatives. However March and Smith (1995) proposed a framework for research projects relevant to the area of information technology. Their framework is based on the idea that scientific research can be divided into two categories "natural science" and "design science". Research in natural science seeks to apply scientific methods to explain some phenomena in IT with the aim of either trying to understand the nature of it, which they termed descriptive, or with the aim of improving it, which they called prescriptive. Design science based research is concerned with the development of an artefact to satisfy some particular goal. It produces tools that serve human purposes and these are assessed against criteria of value or utility. Using these categories March and Smith devised a framework that organised the research activities against the research outputs.

	Build	Evaluate	Theorise	Justify
Constructs				
Models				
Methods				
Instantiations				

Table 1.1. Example of an unpopulated March and Smith research framework

The research framework identified four research activities: build, evaluate, theorise, and justify. The build and evaluate activities are used in design science based research, whereas the theorise and justify activities are used for the natural science based ones.

1. *Building* is the process of constructing an artefact for a specific purpose;

2. *Evaluation* is the activity of determining how well the artefact performs;
3. *Theorise* is the process of constructing a theory that explains how or why something happens;
4. *Justify* refers to the activity of proving a theory. This is done by the systematic gathering of evidence that supports or refutes the theory.

The outputs of the framework were identified as constructs, models, methods and instantiations.

1. *The constructs* (or concepts) are the conceptualisations used to describe a problem in the domain. It is the specialised language and shared knowledge of a discipline.
2. *The model* is an expression of the relationships among the constructs. Models represent situations as problem and solution statements for design based activities.
3. *The method* is an algorithm – it is the sequence of steps used to perform a task.
4. *Instantiation* is the realisation of an artefact in its environment; this refers as much to the tools that address various aspects of design in addition to any eventual software artefact.

An example of the layout for the March and Smith research framework table is shown in table 1.1 above.

1.5.2 The Järvinen research framework

In a later development of the IT based research framework, Järvinen (2004) expanded the work of March and Smith to identify additional categories of research activities and research output. The Järvinen framework makes more of a distinction between the theoretical and practical activities of research, thus identifying five input activities. They also identify differences in the types of method of a research project, defining method identified by March and Smith as normative methods and specifying methods that are used in reality as positive methods. They also identified an additional output called description that allows for the documenting of interesting related phenomena that may occasionally occur. The Järvinen framework is therefore an extension on the March and Smith framework that refines the model, method and instantiations outputs.

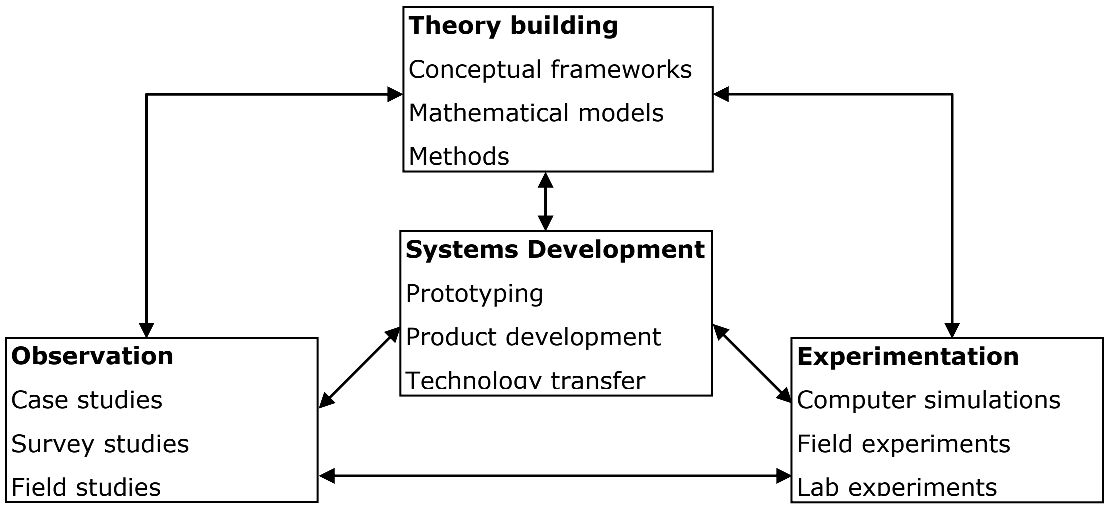


Figure 1.1. A multi-methodological approach to IS research (Järvinen, 2004)

1.5.3 Appling the framework to this research

To investigate the hypotheses this research focuses on the activities of design science. Using the Järvinen framework the following outputs and activities will be produced to address the investigation.

1. *The constructs* for this research cover concepts such as the pedagogical theory, cognitive apprenticeship, the virtual agent, the cognitive architecture, coaching, Python, the development environment, the learner and the types of coding errors.
2. *The model* for the research is used to develop the requirements for the design of the agent and is based on the analysis of errors in light of the material from the literature review.
3. *The method* is the development and implementation of the mentoring agent architecture and development of its knowledge base.
4. *Instantiation* is the evaluation of the agent with reference to the learning of novice programmers.

	Build	Evaluate	Analysing	Creating	Testing
Constructs	Intelligent Agents		The psychology of the novice programmer	The psychology of the novice programmer	
Build model	Intelligent virtual agents	Experimental overview			
Theory model			Cognitive apprenticeship	Cognitive apprenticeship	Experimental setting
Normative method	The cognitive agent architecture	The experimental exercise			
Prescriptive method			The context for learning to program	Categories of programming error	Experimental setting
Instantiations	Animated pedagogical agents	Evaluation of the hypothesis			
Description				NA	

Table 1.2. Mapping of this thesis against the Järvinen research

The mapping of this research against the Järvinen research framework is shown in table 1.2 above. The subject name for the section of the thesis that addresses the particular activity or output of the research is given in the relevant field of the table.

1.5.4 Ethical statement

As this work involves the collection of empirical data from third parties embarked on academic studies, particular care was taken to follow the ethical guidelines as set out by Staffordshire University and The University of Northampton. All the data gathered was made anonymous. Where required those participants involved in experimentation were briefed to the purpose of the exercise and priority was given to the requirements of teaching over those of experimentation in the preparation of material.

1.6 Overview of the Thesis

Chapter 2 reviews the psychology of programming, the nature of programming errors and examines the problems faced by a novice when learning to program for the first time. It also examines the techniques and tools that are available to reduce the occurrence of errors.

Chapter 3 provides a review of the methods and practices for the major pedagogical theories. The pedagogies are considered in terms of their suitability for teaching technical, practice based subjects and highlights the reasons why the cognitive apprenticeship pedagogy is suitable for a mentoring agent for teaching computer programming.

Chapter 4 describes the research on intelligent virtual agents, their properties and capabilities, followed by an analysis of other intelligent tutoring systems that have adopted a cognitive apprenticeship focus.

Chapter 5 is an introduction to agent architecture, types of agent reasoning and the aspects of knowledge-based reasoning that are applicable to cognitive agent systems.

Chapter 6 provides an analysis of the problem domain and the errors produced by novice Python programmers. It includes a brief introduction to the features of Python before giving an account of the programming errors gathered from observation of programming students. The errors are then classified into categories depending on their cause and this analysis is used to inform the design of the agent knowledge base.

Chapter 7 brings together the theories from the literature review and the evidence of the previous chapters making the case for the capabilities of an agent based mentoring assistant for novice learners and a mapping is made from the cognitive apprenticeship pedagogy to the agent architecture.

Chapter 8 describes the design and implementation of MRCHIPS, the mentoring agent system. A description is given of its various subsystems, demonstrating how the agent's behaviour and knowledge of programming errors is used to fulfil the requirements of mentoring.

Chapter 9 describes the evaluation of MRCHIPS in mentoring novice Python programmers; the options for testing and evaluation of the agent are briefly discussed. A description is given of the experimental arrangement used for the evaluation. A discussion is given that examines the strengths and limitations of MRCHIPS.

Chapter 10 presents the findings and analysis of the evaluation. A brief description is given of the reasoning behind the statistical methods of the t-test analysis. An account is given of the analysis of the findings and the results presented. A discussion is then given for the significance of the results.

Chapter 11 brings together the questions of the hypotheses and the empirical findings to summarise the outcomes of the research. A discussion is given reflecting points arising and choices made during the research. Suggestions are then made for future directions where the research and the agent development may be taken.

Chapter 2:

The psychology of the novice programmer

2.1 Introduction

In this chapter an analysis is made of the difficulties faced by students when learning to program for the first time. It gives a number of examples of the nature of the errors made, the different programming tools and makes the observation that although the development environments aid the identification of errors, learners still continue to make the same kinds of errors based on similar misconceptions. The literature, as will be reviewed in the following sections, supports the assertion that learning to program for the first time is a particularly difficult activity. The reason for the difficulty is that there are few analogies in the real world to describe many of the concepts in software. As a result learners have to master two skills when learning to program: they are (i) how to analyse problems to model them within the computer and (ii) how a programming language may be used to express the solutions to problems.

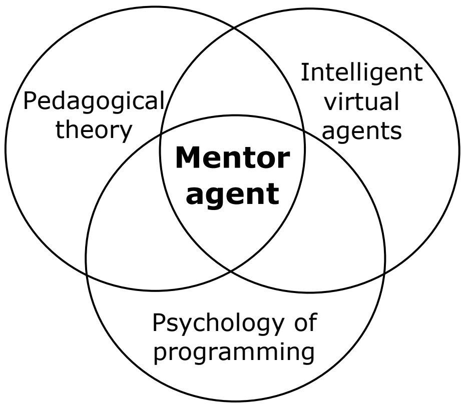


Figure 2.1 Theories informing the mentor agent

2.1.1 Two typical examples of novice programming errors

In 1990 Gilmore made observations of novice programmers as they tackled the problem of constructing a correctly looping program to visit each item on a list (Gilmore 1990). The students had been taught how to code for both the iterative and recursive loop and were allowed to use any method to produce a solution in the POP-11 programming language. He noted one student's particularly tortuous route to a solution as he wrote code incorporating the single error of omitting the initialisation of a variable for the loop with the result that the code did not behave as expected. Rather than attempting to determine the source of the error the student chose to write the code for the recursive solution but again made the single error of not returning a value for the terminating condition. Although the errors required different lines of code to correct both were conceptually analogous, but rather than trying to directly determine the source of the error the student chose the less useful strategy of switching between the different versions of the code a dozen times before he finally noticed his mistake. The observer noticed that when subsequently trying to produce an iterative loop the student again failed to initialise the loop properly but this time only required five attempts to correct his mistakes.

A similar observation was carried out, as part of this research, in 2008 where a novice student programmer was given an exercise that required the implementation of a loop as part of the solution. A group of students had been taught how to code for the two types of iterative loop supported in the Python scripting language, (a recursive solution was also possible but not part of the curriculum). The task was to visit each item in a string and count the total number of vowels present. The observation of one student noted that he had produced a workable iterative loop but was confused by looking for the vowels. The student was asked to simplify the problem to look for occurrences of the letter "e". The student completed the program but placed the initialisation of the counter variable on the line immediately above the one to increment it all within the loop. The student was guided to verify the answer given before he noticed the possibility of an error, but attempts to find a correction involved rewriting the implementation of the loop. Further guidance asking the student to trace the state of the variable led to the student determining the source of the error and finally, after proving that removing the initialisation line was not the solution, the student was able to move the line to occur before the loop to produce a working solution.

2.2 The context for learning to program

The area of study used for this research is the teaching of a first programming language to university students. Learning to program for the first time is a challenging task. Programming a computer is a skill based activity that involves problem solving using the opportunities and within the constraints imposed by a computer environment. In order to characterise the difficulties encountered by programming novices an examination of the psychology of programming is required to provide a context for the errors the novices make.

2.2.1 The novice and the program

A definition of the programming novice is provided by Mayer as a user who has had little or no previous experience with computers, who does not intend to become a professional programmer and who thus lacks specific knowledge of computer programming (Mayer 1980). Programming is the craft of devising a set of instructions for a computer to perform a task, or to solve a problem. The nature of the instructions may be diverse and different authorities have taken different views as to the nature of a program at different times (Pane & Myers 1996). Early programming languages such as Fortran considered the program as a sequence of calculations. Little or no consideration was given to the programming structure and unstructured programming code was shown as an easy way to obfuscate understanding and to introduce errors. Programming structures were devised to control the sequence of instructions and increase the safety of programs, for example the Pascal language. Other authorities viewed programming structures as a means to control access to data thereby reducing the chance of errors during a program's execution (Booch 1993). The functional view defines a program as a series of functional elements that process data and act as input or output to other functions, no static data elements are encouraged and a program becomes an enlarging library of functions. Another perspective is the object-oriented view where a program is considered to be a collection of data elements effectively bound to the instructions capable of processing the data (forming the objects). Objects then process tasks in response to requests from other objects and send messages to other objects to request they process their data (Booch 1993).

Despite the different views on the construction of programs, programming languages are generally represented as a script that describes a series of tasks to be performed by a computer system. All programming languages present two major forms to its user (Pane & Myers 1996): the syntax, the syntactic rules that define how data and code are expressed in the language, and the semantics, the meaning of the statements expressed in the language. An understanding of both the syntax and semantics of a language are important for effective use of the language for solving problems. The programmer must understand the sequence of execution (program flow), the transformational effects of operations on data (data flow) and the purposes of statement grouping (functional design) (Pennington & Grabowski 1990). A programmer's ability to understand computer code is characterised by the ability to comprehend meaning at the different levels of abstraction (Hoc *et. al.* 1990). Skilled programmers are assumed to be able to successively regroup statements into different levels or patterns to determine meaning. Traditionally, programming courses begin by teaching the syntax of a programming language before consideration of the semantics (in reality the processes overlap but semantics lag behind syntax). In education the usual emphasis when teaching a first language is to minimise the number of new abstract ideas to be acquired and to provide immediate feedback to program activity. Languages like Logo are often used for teaching in elementary school, however while Logo is designed as a language for children with no computer experience it is not designed for teaching programming. Other experimental programming languages are being developed to teach programming, such as GRAIL (McIver 2000), but are not widely known or used.

The primary activity of writing a computer program is a design-centred task in that the users construct their own knowledge of the language and how to use it to solve problems. It is similar in essence to other design activities such as architecture, music composition, electrical circuit design or writing an instruction manual (Pennington & Grabowski 1990). However the difficulty with developing a computer program is not only the challenge of using the programming tool to solve a design problem but to also have sufficient insight into the problem solving methods for the domain being modelled. The simplest computer problems might involve computing and arithmetic, for other domains might require computing and accounting, computing and physics, economics, statistics, etc. which means understanding of an additional subject. However the use of a second domain is an aid to understanding. Experiments in teaching mathematical procedures demonstrated that children who were taught by modelling grounded in real-world examples were better able to transfer their skills to more complicated problems than those who were taught the techniques as a set of abstract rules (Mayer 1980). Research also suggests that novice programmers respond positively to interactive learning when developing code (Lui & Chan 2006). In a study into agile software development (also called extreme programming) the performance of an individual was compared against the performance of pairs of programmers when solving example problems. Although no discernable increase in performance could be measured between pairs of expert programmers compared to a single expert programmer, for novice programmers working in pairs there was a notable improvement in productivity over novices working alone. Experienced programmers are able to call on past experience for programming tasks. Results from studies indicate that even programmers with intermediate skills solve programming problems by the application of prior strategies when faced with new situations (Kummerfeld 2006).

Compared to more discrete fields such as physics or mathematics, results from the psychology of programming identify the difficulties for novice programmers in modelling program plans is two-fold: firstly, there are no everyday intellectual activities that are analogous to programming that may encourage spontaneous creativity in the field, and secondly, programs operate on a notional machine (albeit in a physical machine) whose function and operation remain opaque to the learner (Rogalski & Samurcay 1990). This opacity does not allow for the spontaneous construction of programming concepts. Rogalski and Samurcay identify four areas that novice programmers must acquire during the learning process:

1) *A coherent conceptual model of the underlying programming or processing environment of a computer*: The conceptual model was called a 'notional machine' (Rogalski & Samurcay 1990), and difficulty forming a notional machine leads to the learner misunderstanding the activity and behaviour of a running program. Another level of complexity is the similarities and differences between the notional machines of different programming languages. A strictly typed procedural Pascal notional machine is different from an object-oriented Smalltalk machine and a declarative Prolog notional machine, even though in different contexts they may share similar syntactic constructs (e.g. arithmetic). Novice programmers appear to face a great deal of difficulty with constructing their notional models due to the complexity of any useful model that needs to incorporate two major concepts: the use of command systems and the virtual memory structures such as variables, file handlers, etc. to simulate entities with no physical identity.
2) *Control structures*: the primary characteristic of any control structure is that it can interrupt the linear flow of a program's execution. Earlier research was able to demonstrate that structured programs were easier to understand and maintain than non-structured programs (Green 1980). However this has little effect on the difficulty of the use of test conditions for selection and controlling iteration. Control structures provide two areas of difficulty for the novice programmer: the conditional expression and block of executed code as a result. The difficulty a beginner faces with recursive loops is where an iterative loop describes the actions modifying the state during each iteration, while the

recursion describes the relationship between each state of the loop (Rogalski & Samurzay 1990). In general, iteration is taught before recursion and their studies show that students have great difficulty learning recursion. Learners with a greater grounding in logic and mathematics were found to learn the new structures more rapidly.

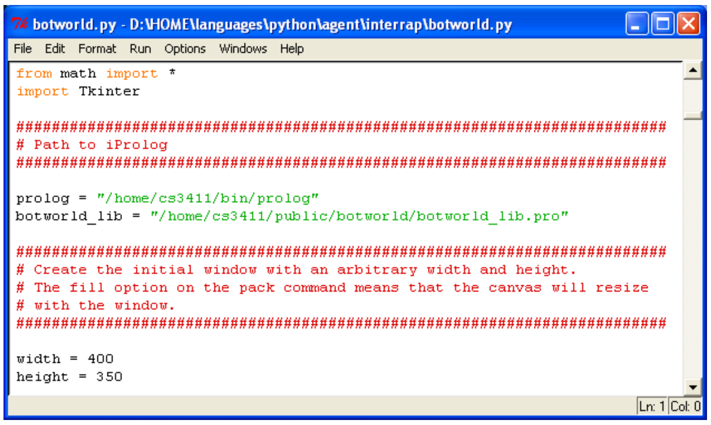
3) *Variables, data structures and data representation*: all programming languages allow for the manipulation of entities used to represent knowledge within the domain. Novice programmers often produce errors due to misconceptions concerning the content of variables, the name of variables and their relation to other elements within a program, the manipulation of a variable's content and the scope of variables (Rogalski & Samurcay 1990). They note that a higher level of conceptual understanding is required for novice programmers to follow the behaviour of variables within an iterative or recursive loop.
4) *Programming methods*: these are the supporting strategies and techniques that aid the programmer in solving problems, such as top-down design, the waterfall model, object-oriented design, etc. Even when familiar with the syntax and semantics of a programming language, inexperienced programmers tend to lack sufficient knowledge to know how to design solutions for specific problems. Studies have shown that beginner programmers find structured design processes more difficult to use because their models are based on the input data and are oriented to processes rather than the more object-based view that expert programmers take (Rogalski & Samurcay 1990).

For the novice programmer an important skill is not only to recognise certain problem situations but they also require knowledge of how to apply appropriate tools and techniques in developing a solution. A study by Perkins and Martin in 1986, which used a series of interviews, allowed them to formulate the nature of the major difficulties faced by novice Basic programmers. They characterised the difficulties as "fragile knowledge" and "neglected strategies". With fragile knowledge the *learner* is aware of the required information but fails to see the opportunity to use it. The researchers identified 4 types of fragile knowledge: *missing knowledge* is knowledge that has simply not been acquired; *inert knowledge* refers to knowledge that the student has but fails to retrieve when needed; *misplaced knowledge* refers to knowledge that is used in the wrong context; *conglomerated knowledge* is a misuse of knowledge in which a programmer combines two or more known structures incorrectly. They were able to confirm that the learner was sometimes in possession of the knowledge by providing hints and clues that would not contain the actual knowledge, but recorded that on nearly 50% of occasions the student then went on to solve the problem. Neglected strategies refer to the way students do not use techniques to gain further understanding of the problem they are solving. They determined that the main strategy that learners neglected was to properly read the code to determine what it actually does (Perkins & Martin 1986).

2.3 Enhanced development environments

2.3.1 Code sensitive editors

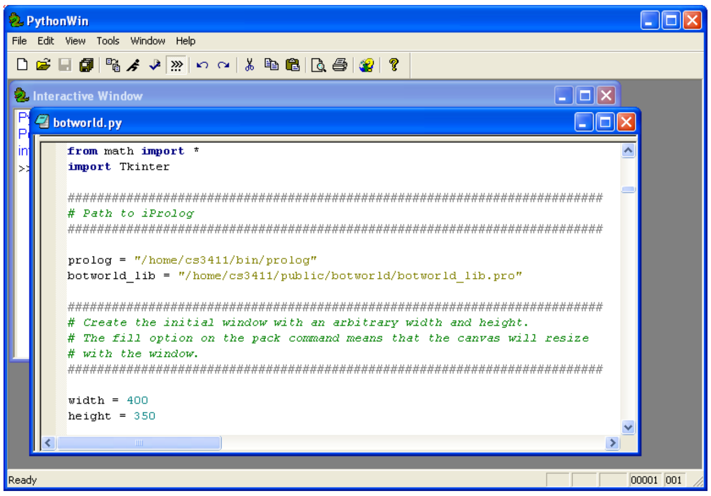
One innovation to aid software development has been the adoption of colour syntax highlighting for program code in text editors (Figures 2.2 and 2.3). This allows the different components of a program script to be displayed in a different colour depending on what category the component belongs to for instance all mathematical operators may be displayed in red, constant numerical and string values in green and language keywords in blue. The purpose of syntax highlighting is to aid the readability of code so that simple errors, such as a misspelling may be noted by the non-appearance of the expected colour and corrected before the code is compiled or run. Colour syntax highlighting is now a common feature of most program text editors, it is unclear whether colour syntax highlighting has any effect on novice programmers; anecdotal observations indicate programmers appear to make little use of the feature. Research on experienced programmers show a preference for syntax colouring with swifter identification of cognitive structures within code, although no corresponding increase in productivity was found with novice programmers (Green 1989). Work by Davies (1991) indicates that syntax highlighting has an influence on the development and problem solving strategies employed by the programmer.



botworld.py - D:\HOME\languages\python\agent\interrap\botworld.py
File Edit Format Run Options Windows Help
Ln: 1 Col: 0

```
from math import *
import Tkinter

##########################################################################
# Path to iProlog
##########################################################################

prolog = "/home/cs3411/bin/prolog"
botworld_lib = "/home/cs3411/public/botworld/botworld_lib.pro"

##########################################################################
# Create the initial window with an arbitrary width and height.
# The fill option on the pack command means that the canvas will resize
# with the window.
##########################################################################

width = 400
height = 350
```

Figure 2.2. The Tkinter colour syntax highlighting editor for Python code

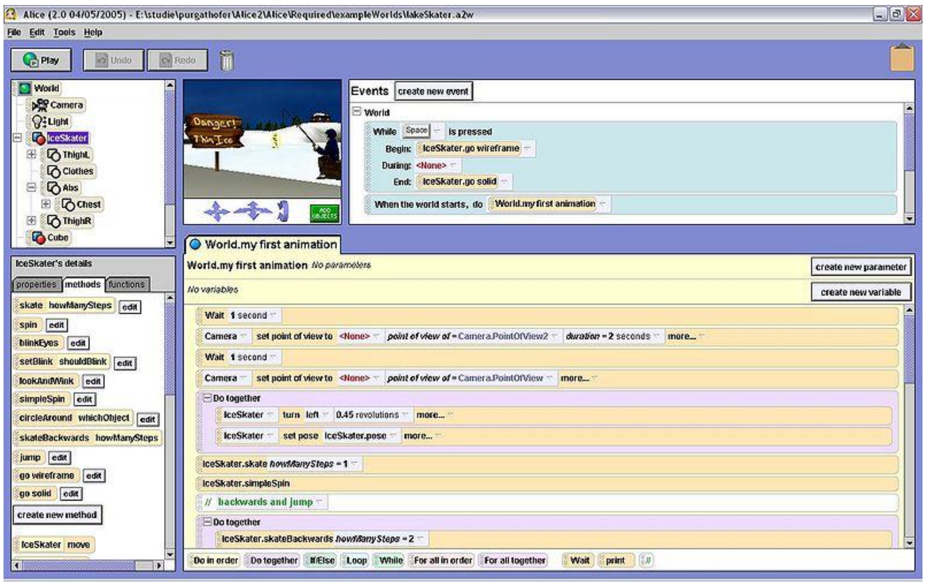


```
from math import *
import Tkinter

##########################################################################
# Path to iProlog
##########################################################################

prolog = "/home/cs3411/bin/prolog"
botworld_lib = "/home/cs3411/public/botworld/botworld_lib.pro"

##########################################################################
# Create the initial window with an arbitrary width and height.
# The fill option on the pack command means that the canvas will resize
# with the window.
##########################################################################

width = 400
height = 350
```

Figure 2.3. The Win32 colour syntax highlighting editor for Python code

2.3.2 Visual programming languages

Software presents the additional challenge to learners in that code can be used to represent not only physical objects but also insubstantial concepts. There are therefore times when visual examples that may be acquired from the real world are not available and designing a suitable analogue for use on a computer can be inflexible and error prone. A number of strategies have been investigated to attempt to remedy the difficulties. One approach is the development of languages especially for teaching, such as LOGO or GRAIL, a more recent example of this type of system is the Alice programming environment (Cooper *et al.* 2003) developed by the Stage 3 Research Group at Carnegie Mellon University. Alice is a 3D interactive programming environment where students are taught the principles of programming code in terms of manipulating characters and objects in a 3D environment. The Alice system is presented as a series of windows that present different resources to the programming environment. One window depicts the 3D scene under development where objects from a library in another window may be drag-and-dropped into the scene (Figure 2.4).



```
World
  While Space is pressed
    Begin:  IceSkater.go wireframe
    During: <None>
    End:    IceSkater.go solid
  When the world starts, do World.my first animation
```

```
World.my first animation No parameters
No variables
Wait 1 second
Camera set point of view to <None> point of view of = Camera.PointOfView2 duration = 2 seconds more...
Wait 1 second
Camera set point of view to <None> point of view of = Camera.PointOfView more...
Do together
    IceSkater turn left 0.45 revolutions more...
    IceSkater set pose IceSkater.pose more...
IceSkater.skate howManySteps = 1
IceSkater.simpleSpin
// backwards and jump
Do together
    IceSkater.skateBackwards howManySteps = 2
```

Figure 2.4. The Alice user interface (courtesy of www.alice.org)

The properties of any object in the scene may be viewed by selecting it and behaviours added by adding code to events that the object responds to. Instead of the learner having to write code in a script, they are shielded from the syntax details by building code from pull-down menus, edit boxes and list boxes. According to the literature Alice was designed to encourage students (typically female students who may not have been exposed to computer programming) to engage with computing by emphasising the use of programming as a method of story telling. In controlled studies involving novice programming students on their first programming course the use of Alice was credited for an average grade increase from C to B and an increase in retention from 47% to 88% (Moskal *et al.* 2004).

However the limitation of this approach is that the learners avoid learning the features of syntax for languages likely to be encountered beyond education. Teachers have found that students who can program in Alice have trouble making the transition to traditional programming languages that use a text editor. So another approach is to design new application tools that guide the learner and couple them with new pedagogical models that specifically address issues, such as the logic of programming structures, the manipulation of different data types that arise in the programming domain.

2.3.3 Intelligent assistance

One application area related to the development of software is that of the intelligent assistant. An intelligent assistant is a software application designed to support a design activity by taking over some of a user's more menial tasks or providing checks and verification of their activity. The nature of the assistance can be passive, only responding to the user's requests or activity; monitoring the user's work and carrying out operations according to set goals.

An example of a passive intelligent assistant is the Genie application (Kaiser 1990). Genie is a question and answer system that is similar to the application help facility available on desktop programs and is designed to provide expert information on the use of a development environment to new users. The information in Genie exists in a single knowledge base but the application acts intelligently in the way that the user is able to interact with it. Genie was designed to address the need to search large knowledge bases to find the appropriate information for the immediate need of the user and to present the answer at the appropriate level for the user's ability. New users to a system may possess different levels of expertise. The system assumes novice users require precise shorter answers while expert users may require more detailed and comprehensive information. So Genie models three levels of user expertise “novice”, “intermediate” or “expert” and tailors its help to be either:

a) *An introduction* where a command is taught that a user may not have encountered before;
b) *A reminder* where a brief description is given of a command that may have been forgotten;
c) *A clarification* to explain the details or options about commands;
d) *Elucidation* to correct user misunderstandings that have arisen or;
e) A direct *execution* of a command on behalf of the user.

Input questions to Genie are constructed in a natural language form from a selection of templates where the user inserts domain specific keywords into appropriate fields to form queries that are then analysed. Typical questions to Genie may be in the form of “What does command C do?” or “How do I accomplish goal G?”.

Marvel is an example of an active intelligent assistant system designed to monitor and automate many of the tasks for organising software development projects (Kaiser 1990). Marvel is similar to a Make facility but is useful for large or complex developments that may be spread across many teams or platforms and not limited to any one programming language, method of development or type of project. It uses a production system that is able to reason about and manage many of the resources in a software development project in accordance with a set of rules and information is processed based on a knowledge base, which contains a description of the project in terms of:

a) *Resources*: software libraries, classes and objects, the development tools and the source code and target platform;
b) *Relations*: among the objects, inputs and outputs, products and variations;
c) *Rules*: which are similar to those in expert systems with a conventional condition part but the action is expressed as a single activity with a set of post-conditions and used to model the requirements of each project.

Marvel can be made to model the stages of the software life cycle and the activities required to transform from one stage to the next.

Processing is carried out opportunistically using both forward and backward chaining and automatically switching between the two when necessary. When, for example, a new procedure needs to be added to a project Marvel knows which dependences need to be updated and performs the necessary operation.

2.4 Summary

For the novice, learning to program for the very first time is fraught with difficulties. To build anything more than the most trivial program skilled practitioners have acquired the skills of how to understand a problem, how it can be represented in a computer, and how to encode it in a given programming language. In addition the practitioner needs the experience to know how to analyse the resultant output of a program, how to trace faults and how to devise solutions to correct errors. In order to make any progress as a programmer, the novice has to acquire these same skills and apply them. The nature of developing software means these skills have to be developed roughly in parallel and to avoid either one undermining the capacity to make progress with the other skills. The major obstacles to understanding for the novice programmer can be summarised as "fragile knowledge", where the learner is aware of the required information but fails to see the opportunity to use it, and "neglected strategies", where the learner does not use techniques to gain further understanding of problem solving in the domain. It was also shown that producing errors while learning to program cannot be avoided, software applications that simply attempt to remove the chance of errors often only delay learning about parts of the language.

Chapter 3:

Cognitive apprenticeship

3.1 Introduction

The cognitive apprenticeship approach grew out of and is a part of the constructivist family of pedagogical techniques; it shares common attributes with methods such as Scaffolding where both require learning materials to be based on real world examples, i.e. materials that are similar to those used by expert practitioners on a subject. The cognitive apprenticeship model differs from others in providing a greater flexibility in the nature of the interaction between teacher and learner and therefore is better able to accommodate computer-supported learning environments. The cognitive apprenticeship model also accounts for a relationship between factual knowledge about the domain that may be gained from traditional textbook based sources and the requirement for heuristic knowledge that experts develop through problem solving practice. The model depends on a learner centred approach; it expects the learner to be motivated to learn the subject, to be attentive, to have access to the learning materials and to be skilled enough to be able to reproduce the desired outcomes. It specifies the need for the learner to develop monitoring, diagnostic and remedial strategies to regulate problem solving so as to be able reflect on their reasoning in a process called meta-cognition. The cognitive apprenticeship model attempts to develop the skills of the learner by allowing them to observe, enact and practice them under the guidance of the teacher with the participants taking on roles that pre-date formal traditional education. For example the way that knowledge was imparted from master craftsman to an apprentice is embedded in the social, deliberative and physical context where the learning activities were guided by interactions between teacher and learner.

The Cognitive Apprenticeship pedagogy was first proposed by Collins *et al.* in 1989 to address what they saw as some of the shortcomings of curricular practices. They proposed addressing these issues by revisiting the traditional apprenticeship model and adapting some of its characteristics to teaching cognitive skills (Collins *et al.* 1989). They observed that apprenticeships involved the social context in which the learning takes place and that important cognitive characteristics are not only derived from didactic instruction but also as a result of a culture of self-motivated exploration from the learners. The work was primarily concerned with the teaching of reading, writing and mathematical skills so the researchers proposed the adaptation of traditional apprenticeships to cognitive apprenticeships for two reasons. Firstly, the pedagogy is primarily aimed at teaching the processes that experts use when handling complex tasks. For this reason conceptual and factual knowledge is made subordinate to the problem-solving context of the task. They argued that an expert in a field is one who is able to solve problems, monitor their performance, make self-corrections, reflect on features and possibly make creative developments in their field. Secondly, this allows the learner to demonstrate a deep understanding of a field. The proponents believed that using real-world knowledge in the relevant context, as opposed to much simplified training exercises should be the basis for developing similar skills.

The researchers then chose to retain the apprenticeship aspects of the model to emphasise that the learning was to be acquired through guided experience, as it was for traditional skills. They acknowledged that models for the learning of physical and cognitive skills were necessarily different but that both shared characteristics on observation, refinement, and correction towards the production of a measurable outcome. They proposed that applying cognitive skills to apprenticeships required the externalisation of processes that were normally internalised. Effective coaching of the learner is impeded because there is no natural access to the cognitive process. The process is also true the other way around in that the masters of a skill may not necessarily have insight into how to explain all of the processing involved in using that skill when teaching and the learner may have limited access to the teacher's reasoning. The cognitive apprenticeship model therefore, was designed to bring these processes into the open through an encouraging of the various stages of the pedagogy.

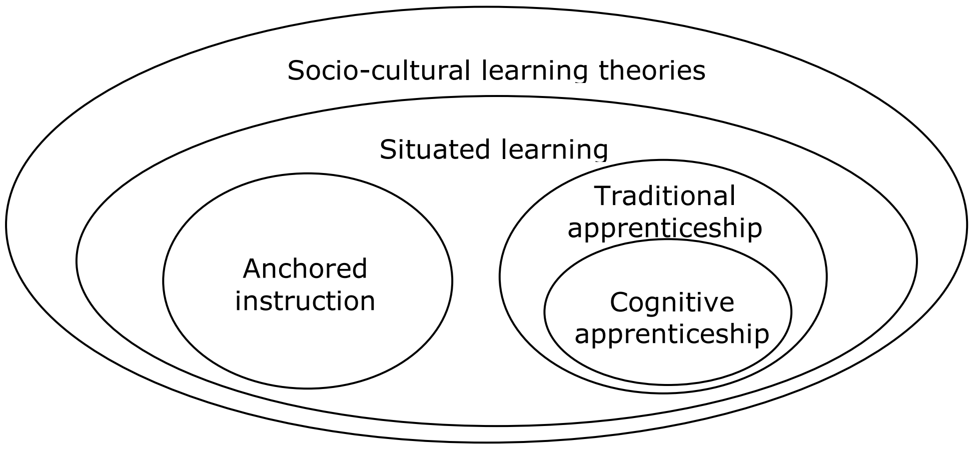


Figure 3.1. The place of cognitive apprenticeship in educational literature – (*Courtesy of Ghefaili 2003)*

Just as with traditional apprenticeship practice in fields such as carpentry, tailoring, *etc.* where the learner acquires skills while working on real tasks and products, so too the cognitive apprenticeship approach where the teacher is able to model processes involved in solving real-world problems. Figure 3.1 illustrates the position and relationship of the cognitive apprenticeship model to other pedagogies. Practice from current educational theory credits one of the strengths of the model is due to the use of real-world situations as the source of the training tasks and this becomes less effective when information is taught outside of a real context: "Situated learning does not mean 'no abstractions' but rather reconnecting formal education to everyday life" (Clancey 1982). The learner is then able to observe the teacher's approach and solution to problems and attempts to reproduce these behaviours. The teacher provides coaching support as the learner attempts the task with feedback, hints and reminders to tune the learner's performance towards a more proficient approach to solving the task. The learner is expected to repeat the tasks many number of times with the amount of support from the teacher reduced as the learner becomes more proficient in a process known as fading. The cognitive apprenticeship model tends to lend itself to computer automation and should also encourage the use of AI technologies and intelligent tutoring systems. In a report in 2001, Woolf and colleagues examined the importance of intelligent tutoring systems in supporting sophisticated interaction, adaptability and focused problem solving as a remedy to the limitations of simpler computer aided educational tools that leave the learner passive and an uninvolved participant in the process (Woolf *et al.* 2001).

3.1.1 Methods of the model

The cognitive apprenticeship model is divided into six main teaching methods which are divided into three major classes of skills: cognitive skills covered in the modelling, coaching and scaffolding methods, development of problem-solving skills addressed in the articulation and reflection methods and autonomy which is encouraged in the exploration method (Collins *et al*. 1989). A detailed explanation of activity for each method is given below:

(i) *Modelling:* In modelling the expert performs a skills task while the student observes the practice involved. The modelling can belong to two strategies: behavioural and cognitive modelling. In behavioural modelling a demonstration of how the task is to be performed is given by the instructor whereas in cognitive modelling the instructor articulates the reasoning that the learner should use in performing the task. Current teaching practice for programming can make use of both modelling strategies with behavioural modelling giving way to cognitive modelling as time and student competences progress. When the teacher articulates their reasoning it is to indicate to the learner what factors are used to guide the decision making during the task. When the learner articulates their reasoning they explain their understanding of the task and their approach to solving the problem.

(ii) *Coaching:* For this step the expert observes the learner performing the skill and offers hints, feedback, and reminders to help them. In addition, if necessary, extra support may be provided by scaffolding, remodelling and goal setting for subtasks. The learner would be expected to crudely follow the steps learned in the modelling phase and, through repetition with support at each stage, to refine their performance and/or their outcomes. The role of the coach is inexact and can be complex but they would be expected to provide motivation, analyse the performance, provide feedback and promote reflection on the task. As coaching has a social context the learner would be expected to seek help or confirm their approach at various times and would also expect the unsolicited help and encouragement from the teacher. The context of the coaching is necessarily driven by the performance of the learner and the literature outlines a number of strategies for effective coaching (Laffey *et al.* 1998). These include the ability to relate the importance of aspects of the task to the learner and to provide reasons for the learner to remain engaged with the task. The coach should work to boost the learner's confidence as they progress. Motivational prompts that are important at the beginning of the coaching can be faded as progress is made.

(iii) *Scaffolding:* For this step activities are organised at the level of the learner's current skills to encourage the learner to progress to subsequent levels where the amount of support is withdrawn. This will be provided by the structure of the course with a series of practical exercises, tutorials and assignments. The structuring of the tasks with increasing levels of complexity allows the student to be able to build on previous lessons and incorporate new knowledge into what has already been learned. The fading of support from the teacher is to encourage the student, during coaching, to tackle tasks using their own resources. The method of the fading could take two formats, either through the quantity of the support with changes in frequency or proactive offers of help, or through a change to the quality of the help using more general guidance or Socratic help to encourage the learner's reasoning.

(iv) *Articulation:* The use of articulation requires the problem solver to explicitly express their reasoning and understanding of the process at the time they are performing the task and while being observed. As a teaching tool articulation should provide additional insight into the expert's view of the domain. The teacher can be made aware of errors, misunderstandings and incorrect assumptions in the student's model of the domain and offer coaching support. Articulation can take three forms with the aim of encouraging the student to self-monitor and to explore the strategies and actions employed: 1) inquiry teaching where the teacher asks the student to answer questions that articulate and refine their theories about the domain's knowledge, 2) articulate thoughts: the teacher can also ask the learner to explain their reasoning as they problem solve and 3) critique or monitor peers in cooperative tasks.

(v) *Reflection:* In reflection the learner is encouraged to critically evaluate their own performance against that of the experts. Expert practitioners tend to have expectations of the results of various activities in a task and can adjust actions to improve outcomes. Learners need to be able to not only apply similar actions, but also to understand if the expectation has been met or how to recover if it has not. There are various suggested techniques for doing this that can recreate the expert's post-mortem of the processes involved and their effects on the problem-solving task. Reflection also allows for the use of audiovisual recording tools.

(vi) *Exploration:* For this attribute the student is encouraged to pursue general goals to tackle problems independently. Exploration requires the questions posed to be made challenging and interesting enough to encourage the student's participation. The major exploration technique is for the teacher to set general goals for the student but to encourage them to concentrate on specific sub goals. The method even allows students to refine the general goals in order to pursue areas of particular interest.

3.1.2 Constructivism

The Constructivist based family of pedagogies share a characteristic with the cognitive apprenticeship model of a learner centred approach to teaching where the emphasis is on the learner to construct his or her individual model of new knowledge rather than being simply a passive recipient of the information presented by the teacher. Constructivism itself is concerned with the learner's actual act of creating meaning (Brooks 1990). The constructivist model argues that the learner’s mind actively constructs relationships and ideas, rather than simply labelling objects that exist in the world; hence, meaning is derived from negotiating, generating, and linking concepts within a community of peers (Harel & Papert 1991). In constructivism, knowledge of the world is constructed by the individual through interacting with the world and the testing and refining of cognitive representation (Boyle 2001). Tom Boyle identified five major principles of constructivism as related to computing science from a list of general principles as:

1) *authentic learning tasks*: learners are better able to learn if they can see the relevance of knowledge;
2) *interaction*: allows learners to construct their own models of a domain;
3) *ownership of the learning process*: rather than the teacher as a taskmaster the learner selects the problem they work on;
4) *experience with the knowledge construction process*: learning how to learn, how to construct and refine new meaning;
5) *meta-cognition*: to allow the learner to monitor and direct their own learning and performance.

Constructivist theory argues that it is impractical for teachers to make all the current decisions and simply "download" the information to learners without involving the learner in the decision process and utilising the learner's abilities to construct knowledge. A major component of constructivism is its emphasis on making meaning through shared cultural, historical, social and political experiences through collaborative activities. While an agent system may be able to simulate some of the social skills in mentoring, to actually share experiences would be beyond the perceptive and reasoning capabilities of the agent.

3.1.3 Scaffolding

In addition to being a pedagogy in its own right Scaffolding is also a method within the Cognitive Apprenticeship pedagogy, by which a tutor provides temporary support to the learner until help is no longer needed. The help can take many forms e.g. explanations, examples, direction, etc. but the help is guided by the learners activity in the subject so the learner is required to be an active participant in the learning process rather than a passive recipient of information. Scaffolding allows learners to attempt things they would not be capable of without assistance. It is similar in essence to a number of other pedagogical strategies such as guided practice, apprenticeships and double-fading support but differs in detail. For example, in the classroom guided practice usually looks like a combination of individual work, close observation by the teacher, and short segments of individual or whole class instruction. In computer based or Internet based learning, guided practice has come to mean instructions presented on the learner's computer screen on which they can act. This action may be to perform some task using a program that is running at the same time, or it may be to interact with a simulation that is embedded in the program or web page. One study of computer-based Scaffolding was carried out into its use in teaching the design of concept maps (Chang *et al.* 2001). The research compared the learning outcomes of constructing concept maps using Scaffolding, termed 'construct on scaffold', against unstructured learning, called 'construct by self' and a non-computer based method, 'construct on pencil-and-paper'. The 'construct on pencil-and-paper' was used to measure for any effect of using computers in learning. Via a series of test results and feedback from students, the results of the study were able to demonstrate that the 'construct on scaffold' concept mapping had a better impact on learning than the other two methods. The results were also able to show that although those students who worked on computers were more positive about the learning there was no significant difference between the results of those groups who learned without Scaffolding.

3.1.4 Double-fading support

Another noteworthy pedagogy is double-fading support (DFS) it is a pedagogical technique that has particular application for teaching of complex software applications with minimal instructional support (Leutner 2000). When learning a new application the learner is locked out of various areas of functionality and provided with detailed guidance (the doubled component) that is gradually removed during training. Leutner and Vogt developed DFS in 1989, as an application of ACT-theory to improve software usability and in practice it is similar to the scaffolding method. To test the effectiveness of the DFS method Leutner monitored the learning outcomes of 208 university students learning how to use a CAD application in two series of experiments. The results indicated that students who learned using the initially reduced software outperformed the control group learning on the fully functional system (Leutner 2000). They were also able to measure that students who were made aware of features that were unavailable to them (e.g. inactive icons and buttons) performed less well than those students where the inactive controls were not visible. Double-fading support appears to be similar to scaffolding but suited to learners in a computing environment. Its major difference is that in the practice of utilising DFS the learning environment is under the control of the tutoring system with components being made available to the learner as they progress.

3.1.5 Anchored instruction

The Anchored instruction pedagogy is a form of situated learning that involves the use of multimedia tools to pose and solve complex realistic problems (see figure 3.1). The developers' goal was to create interesting, realistic contexts that encourage the active construction of knowledge by the learner. The stories presented were designed to act as anchors, sometimes called situated contexts, for the learner to explore rather than a series of lectures. The primary research application area of anchored learning was for the development of reading and mathematical skills at the elementary learning level and although related to apprenticeship pedagogies it is separated by not implementing the methods of cognitive apprenticeship.

3.1.6 Traditional vs. Cognitive apprenticeships

Apprenticeships were the way that skills were traditionally taught; its use predates the development of school-based education. There are differences between traditional and cognitive apprenticeships that impose considerations on teaching of non-traditional subjects (Collins *et al.* 1991). The authors outlined three major differences:

1) Traditional apprenticeships are usually grounded in physical tasks that culminate with a product. The teacher can therefore make their activities easily observable. For cognitive applications the teacher must ensure that mental processes are made visible to the learner.
2) As traditional apprenticeships produce tangible finished products the steps of manufacture are more easily understandable, *i.e.* the avoidance of some subprocess or subcomponent is likely to produce a measurable deficiency in the final product. So for cognitive tasks the challenge is to situate abstract tasks in contexts that make sense to the student.
3) Traditional apprenticeships have skills that are specific to the tasks, i.e. the craft of turning a piece of wood on a lathe is particular to carpentry and it is different and non-transferable to the skills used by, for example, a baker. The cognitive skills developed in the cognitive apprenticeship model need to be transferable; the elements of reasoning and problem solving may have application across many fields.

3.1.7 Expert areas of knowledge

The developers of cognitive apprenticeship, Collins, Brown & Newman (1989) and Collins, Brown & Holum (1991), identified four target areas of expert knowledge that are essential for the learner to gain a true understanding of a field. They then highlighted the limitation of the traditional schooling model in that the focus of the teaching concentrates primarily on the domain knowledge area to the exclusion of the others. The four knowledge areas are explained below:

1. *Domain knowledge*: These are the facts, concepts and relations that exist within a topic that encompasses the knowledge of that subject. Domain knowledge can be thought of as the information that is conveyed in the books and literature about a subject. It

forms the basis and the extent of the knowledge that can be taught by traditional, learning-by-rote, classroom based methods.

2. *Problem solving strategies*: These are the techniques that allow users to achieve tasks within the domain. This is the kind of information that is not obvious from the domain knowledge alone but required to make use of the knowledge. The problem solving strategies might incorporate experiential knowledge and heuristic knowledge "rules of thumb" that expert practitioners might use.
3. *Control strategies*: These are the techniques for recognising and selecting the most appropriate problem-solving strategy for the situations that may arise. This skill involves being able to monitor and diagnose features of the domain and then to select the correct remedial activity to achieve goals for a given state of the domain.
4. *Learning strategies*: are strategies to learn the types of knowledge that are present in the domain and described in the strategies above. Different techniques may be employed for which a rudimentary knowledge, from the techniques above, would be required in order to place it into a proper context.

3.2 A Review of Systems that apply CA

Other researchers have subsequently used the cognitive apprenticeship model for other fields including the teaching of programming languages (Chee 1994, Clancey 1992). There has been much research and use of cognitive apprenticeships in training and education. The model appears to be more applicable to secondary and tertiary education, with papers describing its use in business, law, mathematics, software engineering, research, nursing and medicine, but no examples were found for use in primary education. This may be due to the learner having to acquire a core set of skills in order to benefit from the model's learner centred prerequisite. Another reason might also be that traditional apprenticeships have continued in the age of formal school education but primarily to prepare people for work towards the end of childhood, although early developers of the pedagogy did describe its application to the teaching of reading writing and mathematics in a secondary level schoolroom environment. Table 3.1 below provides an overview of the domains and features of a number of systems that deploy the cognitive apprenticeship pedagogy.

The different teaching systems have all implemented the cognitive apprenticeship model in different ways, from some systems, such as UNCLE (Wang & Bonk 2001) and SIPLeS (Chee 1997) automating most of the methods to others that automate only one or two methods, such as the Cognitive Peedy assistant (Tholander & Karlgren 2002) or CABLE (Chen Mow *et al.* 2006). As cognitive apprenticeship is defined only in terms of its six methods with no constraints on what may or may not be automated, the subject specialists appear to have applied technology based on their individual requirements while remaining within the structure of the theory. Other intelligent tutor systems (ITS) such as PAT (Koedinger 1997), Adele (Shaw *et al.* 1991), Autotutor (Wiemer-Hastings *et al.* 1989) and Steve (Rickel & Johnson 1998) were also considered but not included in this analysis as their developers' evaluation made little consideration of their application to any one pedagogy.

The following systems considered were designed with the aim of aiding learners in diverse domains and have all used the cognitive apprenticeship model differently. UNCLE, an acronym for "Using Notes for Case-based learning Environments" was designed to teach business skills and management (Wang & Bonk 2001). The CABLE system is an examination into the influence of the cognitive apprenticeship to model a learning environment for teaching computer programming in Java (Chen *et al.* 2006). In a similar domain the SIPLeS system was used in the teaching of object-oriented design in Smalltalk (the description of SIPLeS includes the second version, SIPLeS-II. They are a development of an older ITS system called SmallTALKER all by the same author). The Instructional Planning Assisting System (IPASS) provides a multimedia tool to help inexperienced teachers to visualise how a lesson works and a systematic guide to the use of the specific standard and to provide the skills and knowledge to begin their careers. The cognitive apprenticeship has been used as a method for teaching clinical practice to pre-registered nurses by making use of multimedia technology (Woolley & Jarvis 2007). The authors did not report the system as having a name but described the training environment as the clinical practice suite (CPS). SHERLOCK was a computer-based coaching environment employed by the Air Force for training aviation technicians in a realistic context (Lesgold *et al.* 1992). It differed from other intelligent tutoring systems in that it did not model the student but was instead driven by responding to student questions. Evaluation of SHERLOCK demonstrated that subjects who used the system showed an increase in competence over non-users and a troubleshooting ability expected of technicians with four years of job experience.

Table 3.2 below illustrates how the various systems make use of computer automation to implement methods of the cognitive apprenticeship pedagogy. The modelling method is implemented in different ways by each system but the systems often attempt to represent the expert reasoning graphically as part of the user interface. In UNCLE the learner reads the text of an example case study prepared by a domain expert. The modelling is supplemented by the exercises in the later methods, but the initial reading of the case is a manual exercise, albeit one carried out on line. In SIPLeS the learner plays the part of a junior programmer in a software engineering team. The type and nature of the problem is selected from a computer and the problem scenario delivered by a multimedia presentation. Multimedia tools were also used for modelling in the clinical practice suite (CPS) and web-based cognitive apprenticeship systems. The domain modelling for the pedagogical assistant was given by more traditional human based interaction. The web-based instructional planning system also provides a multimedia presentation to supplement more traditional reading materials. To model expert reasoning in Cognitive Peedy a computerised step-by-step account is provided of how a domain expert solves various modelling tasks (Lusk & Atkinson 2007); the information the student has to follow links from decision to decision and questions and difficulties are made explicit.

The coaching method is concerned with modelling, selecting the problem solving tasks, providing hints and feedback on performance (Collins *et al.* 1991). With the exception of the Cognitive Peedy assistant, all of the systems use computer or electronic media to provide coaching support. Different strategies are used depending on the requirements of the domain. CPS records the exercises for later review, the others provide various levels of email feedback from experts or peers while SIPLeS makes use of a case-base archive to determine the feedback to the learner. The UNCLE system is designed to provide coaching through online discussions and feedback from more able peers and teachers. The use of email by these systems works to reduce the constraints of space and time in the access to expert knowledge. The Sherlock system provides coaching in the form of advice when prompted by the user. The advice is slightly different to the hints provided by other systems in that it can indicate what option to pursue next or even indicate what conclusions may be drawn from various factors.

Scaffolding selects the appropriate level of problem task and the fading of the support. Students on the UNCLE system undergo a series of online tests stored in the system's library. The results are then diagnosed and the experts are able to direct learners to additional materials from the library or work with the learners to address difficulties. In SIPLeS the learner is allowed to select their role in and as part of a programming team scenario, from an online menu system, the designer expectation is that the learners would undertake different roles over time. Cognitive Peedy provides a computerised design tool where expert pattern models are presented and may be adapted and modified for use by the learner.

Name of system	Authors	Year	Subject taught	Computerised methods	Human centred methods	Medium	Application level
CABLE	Mow I. C., Au W. & Yates G.	2006	Java	modelling coaching reflection	scaffolding articulation exploration	Network discussion and emails	Tertiary Education
Clinical Practice Suite (CPS)	Woolley & Jarvis	2007	Maternity clinical skills	modelling coaching scaffolding articulation reflection	exploration	Multimedia CDs	Pre-service nursing
Cognitive Peedy	Tholander J.	2002	O.O. modelling	modelling scaffolding reflection	coaching articulation exploration	Interactive agent	Tertiary Education
SHERLOCK	Lesgold, A., Lajoie, S., Bunzo, M., & Eggan, G	1992	Electronics	modelling coaching scaffolding	articulation reflection exploration	Computer simulation	Air Force Technicians
SIPLeS-II	San Chee	1997	OO in Smalltalk	All	None	Custom SW package	Tertiary Education
UNCLE	Wang F.K & Bonk C.J.	2001	Management decision analysis	coaching scaffolding articulation reflection exploration	Modelling	Network discussion	Post graduation business
Web-based instructional planning (WIP)	Lui, T.-C.	2005	Lesson planning	modelling coaching scaffolding articulation	reflection exploration	Web-based Multimedia	Pre-service teachers

Table 3.1. Comparison of Cognitive Apprenticeship systems

Name of system	**Modelling**	**Coaching**	**Scaffolding**	**Articulation**	**Reflection**	**Exploration**
CABLE	Web-based hosting of demonstration notes	Systematic periodic email feedback			Weekly email question/answer session with tutor	
Clinical practice suite (CPS)	Multimedia DVD presentation of the practice	Working in small groups with a teacher, sessions recorded for review		Think aloud while carrying out the exercise	Review of the recorded session and answering questions	Consider adaptation of skills in different scenario
Cognitive Peedy	Step-by-step guide		Pattern Library of problem cases		Interactive animated assistant	
SHERLOCK	Simulation environment over expert model	Advice options or reflections when prompted	Student data used to determine level of support		From system advice	
SIPLeS	Multimedia demonstration of the scenario	Automated advisor returns contextual feedback	Relevant code fragments from a case-base are presented	Optional questions are posed to the learner	Questions and multimedia expert answers are presented	Access to full development environment
UNCLE	Learner reads online example cases	Expert help by email, network conferencing with peers and experts	Online testing and diagnosis	Online questioning and answering	Comparison of learner solution with expert and/or peers solutions	Varied representation of problem, use of hypermedia
Web-based instructional planning (WIP)	Learner reads textbook chapter and views multimedia presentation	Web-based discussion forum with expert teacher	Plans are designed using electronic tool		Learners write their reflections about their design and the results	

Table 3.2. How each pedagogical system implements cognitive apprenticeship

In articulation the student is encouraged to show their understanding of their processing of the task- the social way is to provide a commentary as they address the problem. In UNCLE articulation is partly covered by the activities in the scaffolding, but in support the experts are able to pose additional scenarios and questions in computer conference sessions to challenge the learners. In the CPS the students are required to comment on the task they are performing during stages of the exercise. This is not only to help consolidate their knowledge but provides material to compare and contrast in the reflection method.

Reflection encourages the learner to evaluate their reasoning and think of ways of tuning their future performance to be ever closer to that of the expert practitioner. The UNCLE system encourages the learner to compare their solution with that of peers and experts to gain multiple perspectives on processes and solutions. Reflection in the web-based instructional planning system is a predominantly human centred task. The pre-service teachers write their own reflections on their plans and the demonstrations, which are reviewed by the experts where suggestions may be made. Cognitive Peedy is able to encourage students to reflect on their work by issuing a series of context sensitive questions requiring them to justify their decisions. The authors categorised three types of question that were prompted with no deep critique of the students' work but that were still able to solicit reflection.

Exploration builds on the understanding developed throughout the earlier methods and allows the learner to see how problem-solving skills may be adapted to new situations and across domains. Most of the systems do not explicitly implement tools for exploration but rather allow unstructured access to their tools and libraries for exploration. The main mechanism of encouraging exploration in UNCLE is the availability of the tools and case library outside of the availability of the expert teachers. Other systems such as CPS merely ask students to consider how the skills learned may be adapted or applied to new situations.

3.3 Summary

The cognitive apprenticeship pedagogy has been developed from a traditional and tested method of teaching. As traditional apprenticeships are usually for physical based skills the developers emphasised the new pedagogy was designed for thought based skills and contains activities to promote the cognitive engagement, such as articulation and reflection. The need for the pedagogy arose from the need to address some of the deficiencies of traditional didactic classroom teaching, which have been demonstrated as insufficient to produce expert practitioners in a field. The developers structured a model around the way skills are deployed and used by subject experts and encoded methods by which those skills might be developed.

Method	Agent activity
Modelling	The expert performs the activity while being observed by learner also includes lectures, workshop exercises and assessed pieces of work.
Coaching	The learner repeats the task observed by the expert who provides hints, tips and reminders to aid them.
Scaffolding	The learners activities are tuned to the current level of their skill and the level of support is gradually withdrawn as the learner becomes more proficient
Articulation	Both the expert and the learner are requires the problem solver to explain their reasoning and understanding as they perform the task, to provide expert incites or learner misunderstandings.
Reflection	The learner is encouraged to critically evaluate their performance against the experts to adjust and improve outcomes.
Exploration	To promote active participation the learner is encouraged to additionally set and pursue their own goals and tackle problems independently.

Table 3.3. The methods in the cognitive apprenticeship pedagogy mapped against the agent activity

The cognitive apprenticeship method therefore specifies six methods of practice to be carried out between teacher and learner that encapsulate the pedagogy, they are modelling, coaching, scaffolding, articulation, reflection and exploration and are summarised in table 3.3. The main activity of the model is to develop the apprentice's skills by repeatedly setting them challenges of increasing difficulty, coaching their activities and encouraging the apprentice to become an independent practitioner. This section examined some of the environments that use cognitive apprenticeship methods as a basis for their teaching model. Although the domains and implementation of the model and even the amount of the model addressed were all different, all of the implementations made use of computer based technologies. Cognitive apprenticeship has a number of features that made it an attractive choice for use in this research. Firstly the pedagogy maps to the practice used in teaching programming. The major exception was that the lectures, used in teaching, made for a poor modelling method. This was addressed by emphasising more working demonstrations and examples of practice in lecture materials. Secondly, cognitive apprenticeship provides a structured framework with separate methods where the aims and outcomes of each method may be considered in isolation and easily measured for any evaluation. The third feature of the pedagogy is that the methods may be implemented in different ways (e.g. by exercise, reading material, a discussion, etc.). This flexibility allows for the use of technology for some or all of the pedagogy. One of the main strengths of cognitive apprenticeships is that it accommodates the use of multimedia and intelligent computer-supported learning environments especially in the coaching and scaffolding methods of the pedagogy.

Chapter 4:

Intelligent virtual agents

4.1 Introduction

This chapter is an introduction to intelligent agent systems; it examines the capabilities of agents and the characteristics of the agent environment. An examination is then made of the concept of intelligent virtual agents (IVAs), what they are, their architectures and environments, how they interact with users and the application areas for IVAs with emphasis on their use in education. The agent systems considered do not necessarily conform to the cognitive apprenticeship model or any one pedagogical theory but can be a useful vehicle to demonstrate the value of the IVA model.

4.1.1 Intelligent Agents

An intelligent agent is a self-contained software system that performs some useful action. Intelligent agents are usually viewed as software assistants that take care of specific tasks on behalf of a client or owner. Agent systems need not necessarily exhibit intelligent behaviour and have been researched and used for areas such as communication and networking, so that different authorities make different claims for the capabilities of agents. Wooldridge gives the definition that is usually adopted for intelligent agents as: "An encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" (Wooldridge & Jennings 1995).

Therefore an intelligent agent would be expected to be capable of autonomous decision-making based on the agent's experience and the current situation. The agent should be responsive to events that occur within the environment and the agent is expected to have an ongoing relationship, one that persists over a period of time with that environment. In order to satisfy the design objectives for intelligent agents, Wooldridge and Jennings defined a set of characteristics that the agent should be expected to demonstrate:

- Autonomy – operate without direct control or intervention of a user;
- Social – capable of communicating and negotiating with other agents (or humans) in the environment;
- Reactive – perceive changes in the system and respond in a timely manner;
- Proactive – make decisions and take action based on long term goal-seeking behaviour.

In considering the intelligent agent one also has to consider the characteristics of the environment in which they are situated. Knowledge-based system design traditionally pays little attention to the environment in which systems operate. This was because applications, such as expert systems were not autonomous; the main interface to the environment was only concerned with interacting with a human user. As agents are designed to act autonomously within an environment a definition of the properties of the agent's environment is integral to its design. Different domains impose different constraints on agent systems. Russell and Norvig (1995) provide an analysis of various types of environment, depending on the domain agents may be expected to operate where one or more of the following constraints apply:

a) The environment is not being fully visible to the agent (at any one time) but can be detectable through sensors (and changeable through effectors);
b) The environment may contain other mechanisms (simple machines) and agents (machines of similar capability) and users (agents who may set goals);
c) The environment changes over time (outside of the control of the agent) as a result of other agents or mechanisms present;
d) Changes that occur in the environment are not always predictable (non-deterministic).

e) The environment may pose differing types of requirement to the agent.

Russell and Norvig (1996) specify five properties used to characterise agent environments that influence the required capabilities of a resultant agent. These are:

- *Accessibility vs. inaccessibility:* whether the whole of the environment is detectable by the agent or parts remain unavailable;
- *Deterministic vs. nondeterministic:* does the next state of the environment depend completely on the current state or can the environment change in unexpected ways;
- *Episodic vs. non-episodic:* with the agent's experience divided into separate sensory episodes, can decisions be made based on the experience of a single episode or are the occurrences of previous episodes required;
- *Static vs. dynamic:* if the environment remains in the same state while the agent deliberates then it is static; if the environment can change then it is dynamic;
- *Discrete vs. continuous:* if there are a limited number of clearly defined perceptions in a state then the environment is discrete; if the perceptions are variable then the environment is continuous.

A chess-playing environment for an agent would then be described as being accessible, deterministic, non-episodic, static and discrete while a medical diagnosis expert-system environment would be inaccessible, nondeterministic, non-episodic, dynamic and continuous (Russell & Norvig 1996). Intelligent agent systems therefore provide a mechanism to combine the processing requirements of reactive and deliberative systems to allow decision making to continue even when the state of the environment may not always be apparent.

4.1.2 Animated pedagogical agents

Traditional agent research makes no assumptions about an agent necessarily possessing a physical form. In fact, an agent may be anywhere between a Unix demon process (an automatic task that is executed as a background process) to a controller for robot systems that are able to interact with humans in natural ways such as speech and gesture (Kopp *et al.* 2005). However, believable agents usually exhibit an anthropomorphic form and so present physical representation within a domain; they range from 2D human shaped animated graphical objects on a screen to 3D entities in virtual reality environments that look and behave like humans.

The major aim of believable agent research is the production of the 'illusion of life' in computational systems that allow human observers to suspend disbelief and invest the agent with human-like personality. That is to say the agent is attributed with having feelings, thoughts and desires. In a study by the OZ project group the audience's expectations when observing obviously artificial characters were analysed to measure the effects of realism on believability. The study was able to define a character's believability as: "A believable character is one who seems lifelike, whose actions make sense, who allows you to suspend disbelief. This is not the same thing as realism. For example, Bugs Bunny is a believable character, but not a realistic character." (Mateas 1997: 5-6).

The film industry, especially in terms of animators, has addressed the issue of believability as a factor for engaging with audiences. Their findings have informed various research projects such as the OZ project. The researchers studied the writings of renowned animators, such as Chuck Jones, describing their experiences in creating and presenting effective characters. They provide a set of requirements for believability to include the following (Mateas 1999):

1. *Personality* – what makes characters interesting are their unique ways of doing things;
2. *Emotion* – characters exhibit their own emotions and respond to the emotions of other personalities;
3. *Self-motivation* – characters don't just react to the activities of others. They have their own drives and desires;

4. *Change* – characters grow and change with time in a consistent manner to their personality;
5. *Social-relationships* – characters interact with others in a consistent manner to their personality;
6. *Illusion of life* – requirements such as pursuing multiple, simultaneous goals, movement, perception, memory, language and reactivity.

Of the characteristics listed above the animators were said to have emphasised the appropriateness of emotions, or the ability of a character to reflect the emotion of the observer for a given situation, as the most effective technique to aid the willingness of an audience to suspend disbelief (Mateas 1997). As a result many research projects have been carried out into simulating and expressing appropriate emotions for agents as they interact with users (Krämer 2005, Aylett *et al*. 2007).

Intelligent virtual agents are agent systems that provide an animated character as part of the interface to intentionally solicit an anthropomorphic response from a human user. Various sources describe them as embodied, animated, believable or lifelike agents. In addition to presenting an animated character virtual agent systems offer the opportunity to communicate with the user in ways that model human interaction. Depending on the sophistication of the agent features such as voice input, speech output, natural language parsing, physical gesture, facial expression and dialogue may be used to communicate in ways are more natural for human users and to support a believable persona. Animated pedagogical agents are a subset of intelligent virtual agents and an extension of Intelligent Tutoring Systems where the primary area of application is in providing an educational tool. Early intelligent tutoring systems were little more than multimedia presentations where the learner was a passive recipient of information. Pedagogical theories such as the cognitive apprenticeship model advocate the learner as a performer of activities in the domain as a method of developing deep knowledge (see chapter three). Animated pedagogical agents are used to encourage the participation of the learner by interaction during the teaching session. The current paradigm for desktop computer interfaces is by direct manipulation of windows and icons on the desktop. The effect of this is that in most instances the computer will only do something if the user explicitly tells it to, for example in a browser application clicking and dragging the scrollbar to display a particular page of a document. However when humans work with each other on a task they are able to bring some prior knowledge and understanding to proactively help each other in problem solving. The idea behind interface agents is to have the agent take the initiative in certain recognised circumstances to proactively assist the user to achieve the task rather than await explicit instructions on what to do. The effect of this is that the agent works alongside the user to cooperatively achieve goals rather than acting solely as a servant. This method of agent interaction is referred to as expert assistants or interface agents (Maes 1994).

Current research into the use of animated pedagogical agents is based on examining how effective these tools are for teaching (it is possible that in holding the user's attention they are only a piece of entertainment) and how they might be best used. Studies by Lusk and Atkinson (2007) found that the degree of embodiment for a pedagogical agent did have an influence on how much information learners were able to retain. In a series of tests they compared both how much of the virtual agent was displayed and the degree of animation the agent was able to depict. Results showed that students working with more embodied agents with a sophisticated repertoire of animation were able to outperform their peers who worked with static and disembodied agents in retaining information from the exercise. Research suggests that the use of embodied characters makes (or will make) the interface to computers much easier because it more closely reflects the way humans interact (Cassell *et al.* 1999). In a study exploring the ease of human-computer interaction researchers found evidence that users interact in a more human-like way, that is to say as though they were interacting with another human, when interacting with an anthropomorphic agent interface (Krämer 2005). Measurements of the user's behaviour such as the number of words they used, use of non-verbal gestures and the length of interaction were shown to increase the more human-like the interface.

Empirical studies into Intelligent Virtual Agents (IVA) indicate that users have a greater ease interacting with the IVA than a real person for similar tasks across similar media. Users were said to find a greater degree of trust in the information conveyed by an anthropomorphic agent interface (Sproull *et al.* 1996, Rickenberg 2000), although research into the use of virtual agent in retail applications found an exception when the agent was used in a banking application (McBreen *et al.* 2000). As well as preference for an anthropomorphic agent interface (which may be the result of interest in a novel technology), research indicated that the presence of an anthropomorphic agent improves a user's cognition even when the agent does nothing in a phenomena is known as the *persona effect* (Lester *et al.* 1999). Lester suggests four educational benefits of a pedagogical agent:

1) As the agent appears to care about the learner's progress it may convey that the learner and agent are "in it together" encouraging the learner to care about their progress;
2) An emotive agent that is sensitive to the user's progress may help alleviate the user's frustration before they begin to lose interest;
3) An emotive IVA may convey enthusiasm for the subject that may be reciprocated in the learner;
4) An agent with a rich and varied personality may make the learning fun.

Although the specific reason why the presence of a virtual agent may have an effect on the learning process is not known, the evidence from the studies demonstrates that there is a measurable effect.

4.1.3 Intelligent Virtual Agent Systems

In this section a brief survey is given of a number of pedagogical virtual agent systems; though the list is not exhaustive the systems chosen are as a representative range of the types of environment used to interface with and provide teaching to the user.

Steve

The Soar Training Expert for Virtual Environments (Steve) is an agent designed to assist in the training of tasks within a virtual environment (Rickel & Johnson 1998). It was developed as part of a program to explore the use virtual reality environments as a training tool at the USC. Steve is an autonomous, animated character that operates within a virtual world alongside the student (see figure 4.1). The Steve agent continuously monitors operations within the domain and can manipulate objects within it through virtual motor actions. In operation he can demonstrate tasks, explain his actions (through a text to speech interface), monitor the student's performance of tasks and provide help when it is required. Steve can also respond to verbal questions such "What should I do next?" or "Why" via an interface to a commercial speech recognition application.

The architecture of Steve consists of two major components. First, the high-level cognitive processing module responsible for interpreting the world, making decisions and developing plans, which is implemented using the Soar cognitive architecture (see chapter five, section 4). The second component is the sensory-motor process that interfaces Steve to the virtual world, allowing the cognitive processing module to perceive the state of the environment and changes to it, and activities carried out by the student, etc. The process also sends messages to the environment to affect actions that Steve may take and to control the animation of Steve. The researchers have experimented with several graphical representations for Steve but have made little comment on the merits of different representations. However Steve often appears as a head (including an articulating face), a torso and one or two hands to manipulate objects. The researchers plan to extend Steve's capabilities to allow for non-verbal communication and the expression of emotions to increase the agent's ability to motivate students.

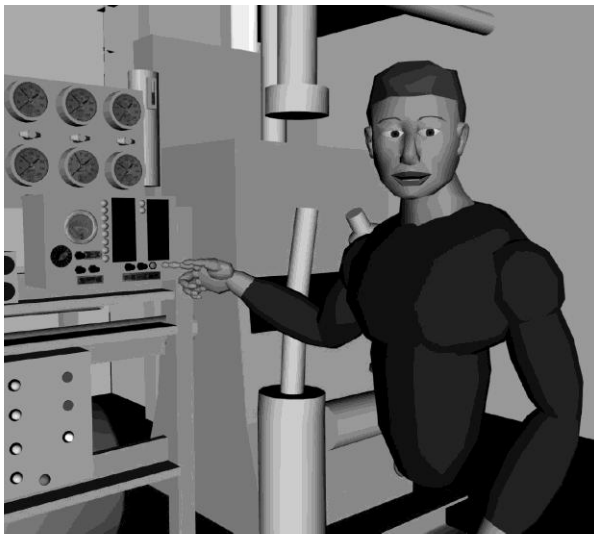


Figure 4.1. The Steve agent demonstrating a control panel – (*Courtesy of Johnson & Rickel 2000)*

The Steve agent has been tested on training simulations for the US Navy for a number of operational procedures. Steve has the knowledge to operate several consoles that control naval ship engines and to carry out inspections of the air compressors for the engines. The Steve agent, in theory, is not limited to performing only in the navy ship domain. As all instructions and knowledge is coded as declarative domain knowledge moving Steve to a different domain should be a matter of producing a different knowledge base.

BodyChat

The BodyChat system (Vihjalmsson & Cassell 1998), developed at MIT, allows the presentation, animation and communication with agent controlled avatars in a virtual reality environment. The system was researched to help in the design of behaviours for avatars. The focus of the research was based around the behaviours that accompany communicating language. Rather than programming each and every action the avatar has to perform, the avatar is controlled by an autonomous agent with a core set of movement and behaviours. The avatar is able to demonstrate behaviours and interact with the user based on the context of a situation. As with other believable agent systems, the user is only expected to use (relatively) few high-level parameters that represent the agent's intention.

The BodyChat system is implemented as a distributed virtual environment with a central server accessed by individual avatars implemented on separate client computers. Each avatar is represented as a 3D model representing the head, with an articulating face, and torso of a humanoid character. The lower body is not rendered (as it probably has little impact in communication). The avatar animations are able to gesture with their arms (wave, salute, etc), produce head movements, the face can articulate mouth movements when speaking, blink and produce eye movement. Special facial expressions can be produced to emphasise various utterances when speaking. Users are able to navigate the virtual environment via their avatar and interact with other users/avatars they encounter.

A four-tiered agent model is used to distribute the processing used to control the BodyChat avatar. Events detected in the environment are passed through the layers to elicit the appropriate response from the avatar. The lowest layer reacts to events in the environment and decides how the agent responds to a given situation. If the event requires a more involved response then a knowledge structure from the second layer that the researchers call a Conversational Phenomena is used to assess the situation and to select an appropriate communication behaviour object from the third layer, which in turn makes use of an associated package of animation gestures from a fourth layer.

PPP

The PPP persona is an animated interface agent that presents multimedia material to the user. While the user views the presentation the agent can comment on particular parts and highlight them through pointing gestures. The agent supports a repertoire of gestures for expressing things such as approval, disapproval, warning, recommendation, etc. The PPP persona system is different from other believable agents featured so far in that the agents do not occupy a virtual environment, but are instead part of a multimedia document. As such the agent exists on the plane of the document as a 2D character (see figure 4.2 below) and their behaviour is described as being similar to a lecturer commenting on a slide presentation (Andre 1999).

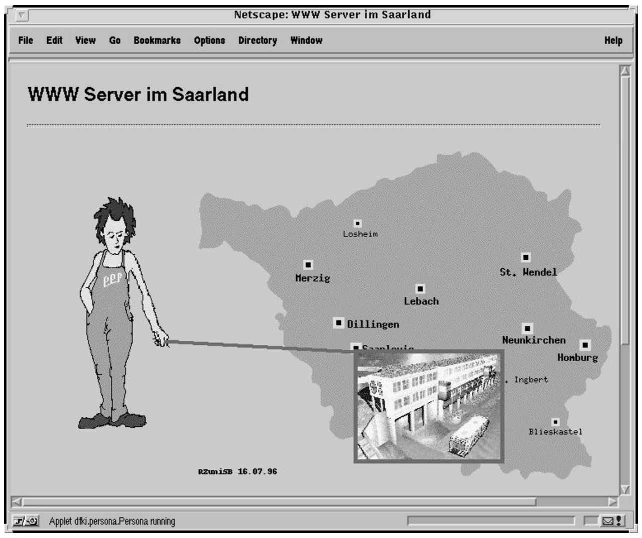
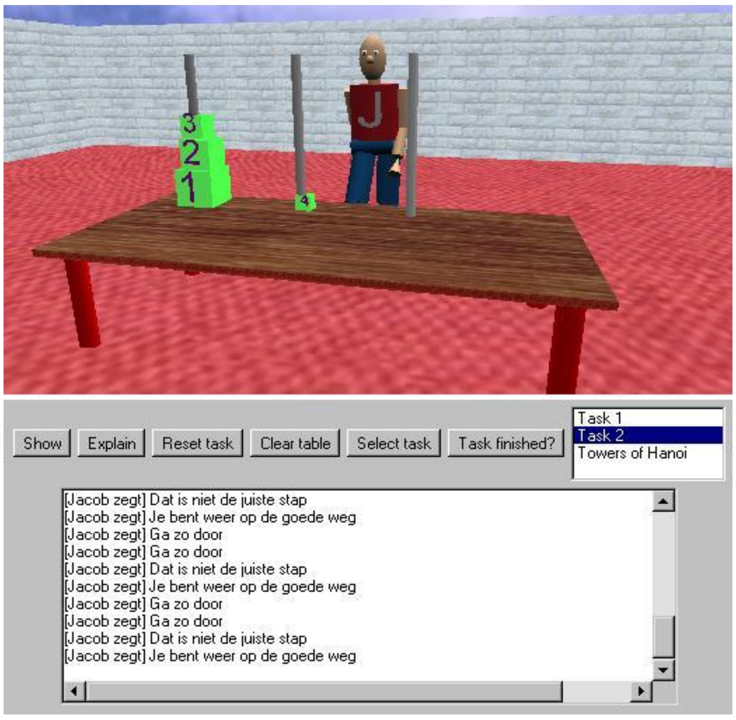

Netscape: WWW Server im Saarland
File Edit View Go Bookmarks Options Directory Window Help
WWW Server im Saarland
Losheim
Merzig
St. Wendel
Lebach
Dillingen
Neunkirchen
Homburg
Ingbert
Blieskastel
PPP
R2uniSB 16.07.96
Applet dfki.persona.Persona running

Figure 4.2. A PPP agent presentation – (*Courtesy of Johnson & Rickel 2000)*

Studies were made with the PPP agent involving subjects watching a series of presentations both in the presence of the agent and sometimes without, on technical and non-technical topics. Comparisons of the results of the comprehension of subjects indicated that the presence of the agent made no significant difference. However most of subjects indicated a preference for the presentations with the agent. The subjects reported that presentations for the technical topics were more entertaining and less difficult to follow in the presence of the PPP agent.

Jacob

The Jacob agent was designed to provide training and assistance within a virtual environment (Evers & Nijholt 2000). It was developed as part of a virtual reality (VR) project called VR-valley developed at the University of Twente, Netherlands. The Jacob agent is represented as a human-like three-dimensional figure that resides in a virtual reality environment. It is able to provide instruction to students to carry out tasks within that environment and assist them if they become stuck. The prototype version was able to instruct users on how to solve the Towers of Hanoi game by manipulating various blocks and pegs within the environment. The Jacob project attempts to answer research questions such as how different interaction modalities (e.g. natural language, gesture) can be integrated into a VR environment and an examination of the use of technologies required to produce the agent. Jacob is intended for interaction and use via a standard web browser. The VR-valley project is based on representation of environments in VRML 2.0 (Virtual Reality Markup Language), which is also used to design Jacobs's form; the agent's reasoning is implemented in Java. The agent's architecture consists of two main modules: the task module that is used to encapsulate knowledge about the task being performed, which objects need to be manipulated, how the task is to be performed, what errors a user can make, etc. The second module models the instructional knowledge. The researchers postulate that the techniques of instruction are independent of the specific task; the module is able to adapt the agent's behaviour depending on the actions and progress of the user. The Jacob agent is similar to the Steve research, the major difference is that Steve is based on custom technologies and provides an environment in which the user is fully immersed, whereas Jacob makes use of commonly used computing applications, such as web-browsers, and standards.



```
[Jacob zegt] Dat is niet de juiste stap
[Jacob zegt] Je bent weer op de goede weg
[Jacob zegt] Ga zo door
[Jacob zegt] Ga zo door
[Jacob zegt] Dat is niet de juiste stap
[Jacob zegt] Je bent weer op de goede weg
[Jacob zegt] Ga zo door
[Jacob zegt] Ga zo door
[Jacob zegt] Dat is niet de juiste stap
[Jacob zegt] Je bent weer op de goede weg
```

Figure 4.3. The Jacob agent teaching the tower of Hanoi – (*Courtesy of Evers & Nijholt 2000)*

An image of Jacob is shown in figure 4.3. As a model of a mentoring agent it is able to demonstrate a number of interesting features. The user and the agent share a common environment in which the user attempts to solve a task, the user is the primary problem solver, the agent merely provides help, only if asked for, when the user becomes stuck, the agent is able to confirm when a correct solution has been reached.

FatiMA

The FearNot! affective Mind Architecture (FatiMA) is an agent architecture designed to operate characters in a virtual storytelling education application for pre-teenage children (Aylett *et al.* 2007). The FearNot! (Fun with empathic agents reaching Novel outcomes in Teaching) system presents a 3D cartoon-like environment (see figure 4.4) in which interactive dramas are played out. For example, where one or more of the characters bully another character in various situations and between episodes the victimised character is able to receive advice from the user. FearNot! was designed to allow children to explore the actions and outcomes of various bullying scenarios in safety and without inducing feelings of victimisation in the child users (Aylett *et al.* 2005).

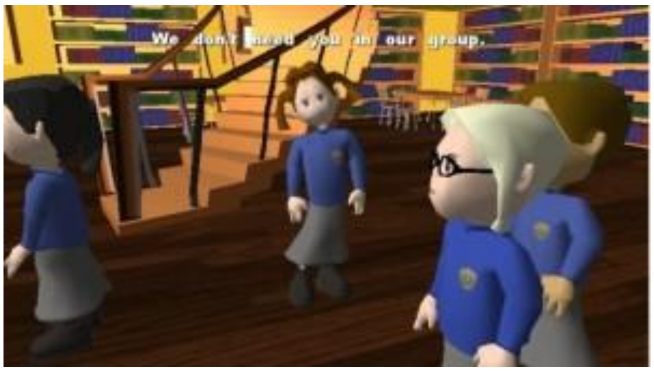
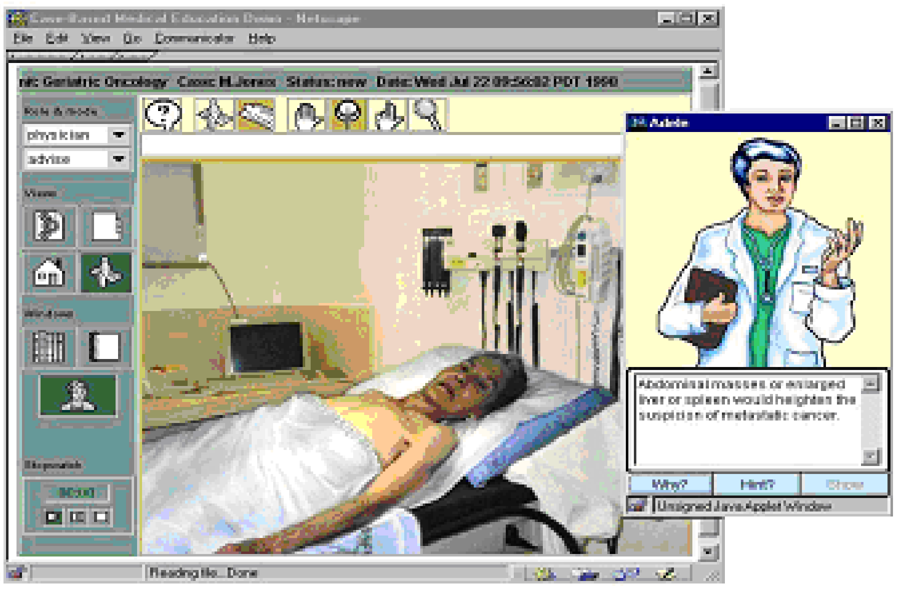


Figure 4.4. FatiMA characters in FearNot! story – (*Courtesy of Aylett 2007)*

FearNot! allows for a phenomenon called emergent narrative as the author of the scenarios only set up the initial premise and background information for the characters – there is no pre-determined sequence of events or ending to a story and the story unfolds as the characters interact driven by the FatiMA agent architecture. Each character in the world perceives information about events and objects in the world and is able to carry out actions through actuators. The FatiMA agent architecture is based on a system called OCC (named after its designers Ortony, Clore and Collins), where decision making is not only based on deliberative and reactive reasoning but is also influenced by a simulation of the emotional response to stimuli. Upon receiving a perception the FatiMA agent evaluates its significance and produces an emotional response and if the event activates a goal intentions are also set up to be achieved. FatiMA agents support autobiographical memory so that past interactions with the user can affect decision-making.

Adele

Adele (Agent for Distance Education – Light Edition) is a 2D animated pedagogical agent, which is implemented as a web-based Java applet. Adele presents a software personality to assist medical and dental students in working through course materials. Adele was part of the ADE (Advanced Distance Education) project, which researched the use of artificial intelligence in the creation of adaptive courseware that may be delivered via the Internet (see Figure 4.5). Adele works by presenting case-based diagnostic situations to the learner highlighting relevant and salient parts. The learner can ask questions of the case, specify tests to be performed and receive the results. Adele then monitors and provides appropriate feedback to the learner.


File Edit View Go Communicator Help
physician
advice
Adele
Why?
Hint?
Show
Unsigned Java Applet Window
Reading file...Done

Abdominal masses or enlarged liver or spleen would heighten the suspicion of metastatic cancer.

Figure 4.5. Adele explains the importance of palpating the patient's abdomen. – (*Courtesy of Johnson 1999)*

The virtual agent personality is maintained through animating the character representation by swapping frames in and out of the applet window. It is unclear from the literature frame whether the effect is a series of static images or at a rate to provide continuous movement, but the authors assert that users find the amount of the personality acceptable (Johnson *et al.* 1999).

Adele consists of two components: the case simulation engine and the pedagogical agent. The agent contains a reasoning engine that is able to follow the student's decisions and monitors the state of the simulation. Knowledge in the agent is represented in the form of hierarchical plans that include preconditions and effects for each action. The decisions made by the agent are based on the student model, a task plan for the medical case, an initial state and the agent's current mental state, which is updated as a student works through the case.

4.2 Summary

Intelligent agents are self-contained software systems that perform one or more useful operations on behalf of a client or user. They are characterised as being autonomous, sociable, reactive to changes in the environment and able to pursue long-term goals. This allows agents to operate in environments where changes may occur unexpectedly or all of the information to allow decision-making may not be conveniently available to the agent. One aspect of agent research is the implementation and application of intelligent virtual agents, which is the presentation of the agent as an anthropomorphic character for social interaction with users. Other applications for IVAs include a range of educational applications where the agent may carry out a range of activities such as demonstrations, question answering or making assessments of the learners work. There are different strategies for implementing IVA applications in education; many exist in 3D virtual environments but 2D IVAs may also be used on desktop or web-based applications. The use of an animated virtual agent interface offers two facilities to an agent based mentoring system. Firstly, the opportunity to use modes of communication that appear more intuitive to the learner avoids the additional cognitive load of having to learn an application interface to access the mentoring knowledge. Secondly, the phenomena of the presence of an anthropomorphic character for in increasing performance in computer users should also aid the learner.

Chapter 5:

Reasoning in agent based systems

5.1 Introduction

In the previous chapter the concept of intelligent virtual agents and their application were introduced, however little examination was made of the reasoning mechanisms used to produce the intelligence. The architecture of an agent describes the components that determine the processing and reasoning capabilities of the system. A number of different solutions arising in the field of Artificial Intelligence (AI) have been used to provide the reasoning capabilities of agents, such as rule-based production systems, Bayesian belief network, fuzzy logic, *etc.* to equip the agent to respond to its environment. This chapter examines the two knowledge-based reasoning technologies that are the basis of the decision making in the developed mentoring agent, MRCHIPS, described in later chapters. The technologies are the Beliefs Desires and Intentions (BDI) planning and Case-Base Reasoning (CBR). The architecture of an agent describes the arrangement of its component parts, which eventually determines its behaviour and capabilities. The design of the architecture is determined by the required behaviour of the agent, the environment in which the agent acts, interactions with the environment, knowledge representation, the way in which information is processed and the communications interface.

5.2 The BDI agent architecture

The most common architecture in use for agent systems is based on a method of reasoning called the Belief-Desires-Intentions (BDI) model, see figure 5.1. The BDI agent model was developed from a theory of human practical reasoning originally proposed by the philosopher Michael Bratman in the mid-1980s. In the BDI architecture the software structures are used to represent mental attitudes about the world (beliefs), the goals the agent is to pursue (desires) and a set of behaviours (intentions) the agent is able to perform. The architecture allows for the agent to make decisions based on its mental attitudes, an interpreter makes use of the data structures to select given behaviours (intentions) to achieve certain goals (desires) in response to particular attitudes (beliefs) formed from sensing information about the world. It is the combination of the BDI structures that allows the agent to reason both reactively in response to its environment and deliberatively to pursue goals (Bratman, 1987). Many BDI systems also include event data structures when interfacing with the environment and a library of plans as part of their reasoning mechanism. Events usually lead to the formation of new goals and thereby contribute to the desires mechanism, while the library of plans defines "recipes" for the actions the agent is able to perform. Although BDI systems are primarily deterministic in that they maintain an internal representation of their domain, plans can be made to act reactively by having a simple sense and response structure. Research carried out at the University of Michigan describes how a BDI based system (PRS – the Procedural Reasoning System) was used to control the activity of a robot to navigate a path and correctly cope with unexpected obstacles in its path (Lee *et al.*, 1994), demonstrating its deterministic reasoning in the path finding and reactive reasoning for obstacle avoidance. More modern BDI systems are able to demonstrate similar capabilities in real-world activities as well as for virtual environments such as games and simulation (Baillie, 2004).

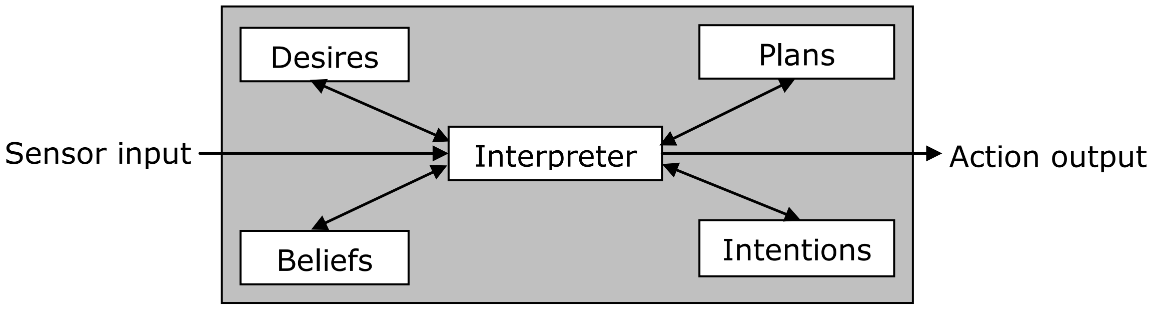


Figure 5.1. The generic BDI architecture

The BDI agent architecture is a mature and widely used model for agent reasoning, it is used to resolve the problem of choosing the appropriate action in a changing environment and based on a model of the mental attitudes of people as agents in a defined environment. A BDI interpreter maintains a number of data structures for coordinating the execution of plans to achieve goals while remaining responsive to new events (see Figure 5.3). The mental attitudes that are modelled in BDI are:

- *Beliefs* - these are informational structures that reflect the current state of the world. Although similar to facts in a knowledge base, beliefs model knowledge that is based largely on an agent's perceptions from the environment, other beliefs may be inferred or the result of a communication. Beliefs are very dynamic; they are changed constantly during an agent's execution and can contain inaccuracies such as out of date facts.
- *Desires* are the motivational structures for the agent that specifies the objectives to be accomplished. Agents may have multiple desires to achieve; during execution only a subset of desires need to be active at any one period; the activity of goals can be switched on or off depending on the context indicated by the agent's beliefs therefore the set of goals may be unrelated, complementary or even incompatible with one another. Desires and goals are generally treated as synonymous terms in the literature, but some authorities (Rao and Georgeff, 1995) do make a distinction between them and distinguish goals as tasks to be accomplished, while desires are states to be maintained.
- *Intentions* are produced in response to desires; these are the structures that represent the selected course of action of an agent to achieve a desire within a current set of beliefs. They are dynamically generated paths of reasoning indicating the current state of the agent's deliberation and can be used to backtrack if necessary. The deliberation in most BDI systems is guided from a library of plans.

Practical BDI implementations also maintain additional data structures that support the core deliberative and reactive structures.

- *Plans* are maintained in a library by the agent and encode the agent’s capabilities. The mental attitudes of a BDI system are manipulated by a series of planning rules in a plan base to produce the behaviour of an agent. Each plan describes the processing activities required to achieve one or more goals. Although some BDI implementation may contain more elaborate structures plans are basically comprised of three structures, (see figure 5.2).
    1. An invocation condition, the head of the plan, which is a goal to be matched against goals in the desires;
    2. A context sometimes called a guard condition which contains one or more beliefs that must hold for a plans to be activated; and
    3. A body which contains a sequence of primitive operations or subgoals to be executed and is placed on the intention stack if the plan is activated.

    More elaborate plans may also contain exception structures that specify operations to be carried out on the failure or successful completion of the plan body.

Goal ← Context | Body

(a)

```
plan:
    goal    : awakeAgent,
    context : avatar:isVisible(false),
    body    : {
        write('== maximise Agent'), nl,
        avatar:isVisible(N),
        if N \== true then {avatar:show()}
    }.
```

(b)

Figure 5.2. Logical structure of a plan (a) alongside a practical working example (b)

- *Events* are dynamic goals or beliefs that may be added or deleted to trigger or alter the activation of plans. Events may originate either externally from the environment to the agent or internally from the agent to the agent. External events are the signals or messages from the environment that trigger a response or thread of reasoning in the agent. Internal events are the subgoals generated during the execution of the body of active plans.

One of the earliest practical implementations of BDI was the PRS developed for NASA and used in fault diagnosis for space shuttle systems. A later system, DMARS, was a re-implementation of PRS in C++ (PRS was written in Common Lisp). Both PRS and DMARS were considered a general-purpose model of BDI but they were said to lack portability and sufficient explanation of their runtime reasoning (Chen 2003). Other BDI systems such as JACK, Jadex and JAM are also based on PRS. They are said to be based on an engineering approach (Ancona *et al.* 2005) and are implemented in Java. Knowledge in these systems is usually represented in a highly procedural Java-like notation with great emphasis on easy re-use of code libraries and integration with the external environment and system. Other BDI developments concentrated on establishing a closer link between the theoretical aspects and a practical abstract interpreter that could be used to implement real systems, which led to the definition of systems such as AgentSpeak(L) architecture (d'Inverno & Luck 1998) and Jason, a Java implementation of AgentSpeak(L). More recently a version of PRS was implemented in Python by Stanford Research Institute, called SPARK (SRI Procedural Agent Realization Kit), with features to address the issues of formal properties and application development (Morley & Myers 2004). The theoretically derived BDI systems such as Jason (a popular open source implementation of AgentSpeak(L)) and 3APL are also implemented in Java but represent and process knowledge in a declarative Prolog-like programming language, 3APL makes use of a clearly distinct embedded Prolog engine while Jason tightly integrates the unifier and resolution process into its plan interpreter.

5.3 Case-Based Reasoning

The second major reasoning subsystem of MRCHIPS is the Case-Based Reasoning (CBR) engine. Like BDI, case-based reasoning is a method of problem solving inspired from a theoretical model of how humans reason. It is the CBR module of the agent that is concerned with the domain knowledge about the Python language, novice errors and where identification of anomalies with novice errors is done. In general symbolic reasoning mechanisms can be classified into two general categories: deductive reasoning and inductive reasoning approaches. Although there are many varied types of deductive reasoning the most established method is typified by rule-based reasoning as used in production systems. A rule-based reasoning system uses existing domain knowledge in the form of rules to make inferences about new problems and is considered an effective reasoning mechanism when the theory of the underlying problem domain can be well defined and easily encoded into rules. Deductive reasoning in a rule-based system works by progressively rewriting the problem state in working memory so it more closely resembles the solution space (Jackson 1999). The major weakness with using rules is the relative expense of the knowledge elicitation process for developing the rule-base.

One alternative to rule-based reasoning is Case-based reasoning, which records knowledge in terms of entire diagnostic situations and reasons inductively to draw inferences for new cases based on the experiences learned from previous encountered cases – if the experience is not quite sufficient for the new problem then they are often able to make adaptations to likely strategies to achieve their goals. Case-based reasoning attempts to solve problems by making analogous links to similar problems that may have been encountered before. In CBR knowledge is represented in schemas of information containing attributes and values known as cases and reasoning is performed by comparing cases against each other to find similar historical episodes. CBR systems are concerned with finding the best match to a solution rather than an exact match and cases are selected by searching for an appropriate match to a current problem. Once one or more candidate cases have been selected various attributes of the case may be adapted to make it more appropriate to the problem being addressed. However rather than being a single solution CBR describes a family of information processing techniques that attempt to solve new problems from prior experience rather than first principles.

A CBR system involves storing and recalling previous examples of similar problems. New cases (that have proved to be successful) may be stored and can be used for solving later problems; this is in effect a form of machine learning. The primary reason for using CBR is that it appears to be particularly well suited to representing the knowledge about the learner errors; the literature describes a number of typical characteristics for suitable problems for case-based systems:

a) Problems for which the domain knowledge is broad but shallow;
b) Where the primary source of information is based on experience rather than theory;
c) For problems where the requirement is for the best available solution, rather than a guaranteed exact solution;
d) For domains where solutions are reusable, rather than unique to each situation;
e) The search space for cases has a mechanism that draws similar cases together;
f) The similarity metric (for the domain) exploits this similarity.

Case-based problem solving is acknowledged to be an attractive alternative to rule-based solutions if the knowledge available is already organised in cases (Tanimoto 1995) and where there does not exist any accepted set theory or set of rules that can be used to solve new problems directly.

Irrespective of the details of implementation a case-based reasoning system consists of two components: a library of prior/historical cases, which forms a knowledge base – the case-base, and a reasoning mechanism to select and apply the most applicable case. Cases are defined as being a complete description of a diagnostic situation; they contain a description of the symptoms of the case, information about the failure or cause of the failure and a strategy to repair the case. The reasoning mechanism consists of a means of using the key elements of the present problem to find and retrieve the most similar case (or cases) from the case library. This is called indexing, a method for modifying the selected solution to make it applicable to the current problem and finally a mechanism for storing the modified case in the case-base.

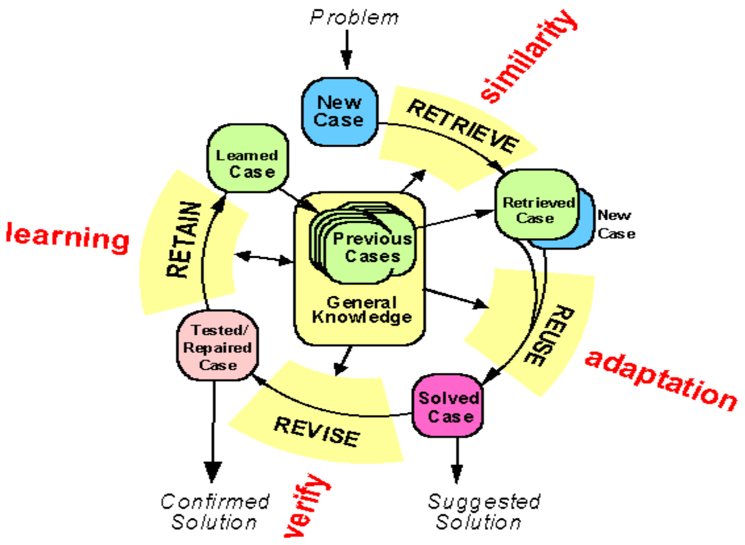


Figure 5.3. The CBR process cycle (Aamodt & Plaza 1994)

Aamodt and Plaza defined the reasoning mechanism in CBR as a four major step process called the four REs (Aamodt & Plaza 1994), as illustrated in figure 5.3. They are:

1. RETRIEVE the most similar case(s);
2. REUSE the case(s) to attempt to solve the problem;
3. REVISE the proposed solution if necessary; and
4. RETAIN the new solution as a part of a new case.

The retrieval process is concerned with using the features of a case describing the current problem to help select the best matching previous case. In retrieval the case engine identifies features of cases to make a comparison, matches features against other cases and selects the closest match.

It is likely that the problem case and the selected case still contains differences. The reuse process is concerned with making a copy of the selected case and adapting to apply to the current problem. Different strategies may be used to transform the selected case depending on the requirements of the CBR. A domain-dependent model such as rule-base is used to govern the transformation of the solution into a new case. The effect of the adaptation process may be to adjust parameters values in the solution, to reorder the sequence of operators or insertion or removal of operators.

After reuse the case is applied to the domain but there are circumstances where the selected solution fails, so CBR provides a mechanism to assess the effectiveness of the new case and revise it if necessary. Both the assessment of the solution and the revision itself may require the intervention of a human operator or the use of some external reasoning system. The use of revision allows CBR to make use of failure as a learning mechanism.

The final process of the reasoning cycle allows the CBR to automatically add to its knowledge base by retaining the new case. One of the major tasks of the retain process is to ensure the new case is indexed on features that allow the case to be selected should the appropriate problem features arise. Depending on the implementation the whole of a new case may be retained or only those parts that differ as a result of reuse and revision. It may be required to retain even those cases that failed after revision as it can be a useful indication to situations that have no solutions.

The earliest development of CBR was credited to Roger Schank and associates at Yale University in the 1980s, it was based on the proposition that when faced with a new situation humans are able to plan and make decisions based on lessons from prior experience rather than the first principles as modelled in the knowledge-based technology of the time. To model this type of reasoning they developed a frame-like knowledge representation scheme called a Memory Organisation Package (MOP), where each MOP was used to represent a concept, an entire case or some facet of a case. The MOPs could be as simple as a single value but usually represented more complex values such as a sequence of events or a relationship. The MOPs are linked together to form a network of abstract and instance data to represent the case-base. The use of MOPs as a knowledge representation scheme has the advantage that a different granularity of features of a case may be represented. The features may be easily manipulated for revising cases and MOPs may be inherited from the memory efficiency of storage and searching (Riesbeck & Schank 1989). Although efficient the difficulty with working with MOPS is that individual cases are distributed across many frames and it is not always intuitive how the granularity and hierarchy of each MOP should be representation and organised. The first CBR system based on Schank's work was a question-answering system with knowledge about diplomatic missions called CYRUS and developed by Janet Kolodner (Kolodner 1983). A number of other prototype CBR systems were developed at that time, such as CHEF, which demonstrates case-based planning in the cookery domain; JULIA, a case-based designer; CASEY, a hybrid CBR diagnostic program, using case-based and model-based reasoning; SWALE, a case-based explainer for anomalies in stories; HYPO, which provides CBR in the legal domain; and CLAVIER, a CBR used to layout composite components in an autoclave. Other knowledge representation schemes such as a flat file, relational databases and program objects have also been used to represent cases (Chi & Kiang 1991). Prodigy/Analogy is a CBR planner that, like CHEF, integrates rule-based reasoning to allow multiple strategies to be used when solving problems (Veloso 1994). At the core of Prodigy/Analogy is a domain-independent, non-linear planner that uses means-ends analysis and backward chaining to find solutions. The amount of searching performed by the planner is reduced by the CBR that records decisions, their contexts and outcomes at given points during planning to form cases. When similar circumstances reoccur the cases may be recalled to save the amount of planning required from first principles.

5.4 The cognitive agent architecture

The study of cognitive agents architectures is concerned with devising the set of principles and artefacts required for creation of general-purpose intelligent systems, rather than describing any one method of processing. Cognitive architectures are defined as theories of how the mind integrates different processes to produce thoughts and behaviours

(Stewart 2006). Over many years of AI research fields have tended to fragment into the examination of specific subcomponents that underlie intelligent behaviour, but with little concern for how components work together (Langley 2006). In his article Langley identified that the production of versatile intelligent systems such as sophisticated robotics, intelligent tutoring systems, and embedded virtual characters, require generalist intelligent reasoning resources whereas much of the field centres around pure or "niche" reasoning systems. There are three architectural paradigms concerned with how intelligent systems may be combined to produce more general reasoning resources. The oldest architecture is the *blackboard* system where a collection of independent reasoning systems (called knowledge sources in the model) tackle particular subtasks of a problem and share information on a centrally accessible knowledge-base known as the blackboard (Hopgood 2000). The blackboard allows information to be selected, added or deleted as required by each knowledge source. This allows each knowledge source to remain independent of others but does not allow for knowledge sources to use information about the capabilities of other parts of the system to route knowledge to parts of the system where it needs to be processed. The second and most widely known architecture is the *multi-agent* systems framework in which several interacting, intelligent agents work together to pursue a set of individually held goals or perform a set of individual tasks (Hopgood 2000). As with blackboard systems each agent undertakes a facet of a problem but this time communicates directly with other agents. The design of multi-agent systems makes use of the social capability of agents and is therefore concerned with how agents negotiate with one another if they wish to solicit services from each other. The third paradigm is the *cognitive agent* architecture, which was advocated by Newell (1990), where the architecture should be based on theoretical assumptions about the mind and subcomponents should be highly interdependent on one another. The work of Newell, one of the contributors to the development of the Soar cognitive architecture, was credited by Langley (1991), who extended his theories to define the four commitments for the development of cognitive agents architecture:

a) They should be based around short-term and long-term memories that store the agent's beliefs, goals and knowledge;
b) Clear representation and organisation of structures that are embedded in these memories;
c) Clear functional processes that operate on the memories for both retrieving and maintaining content;
d) A programming language that allows the construction of knowledge-based systems that embodies the architecture.

A number of intelligent agent systems encompass Newell's commitments in their implementation, such as Langley's Icarus architecture (Langley *et al.* 1991), the Soar architecture (Laird *et al.* 1987), which is used to drive the reasoning of the Steve virtual agent described in section 4.4 and ACT-R agents (Anderson 1993), EPIC (Kieras & Mayer 1997) and Clarion (Sun *et al*. 2001).

5.5 Alternate reasoning methods

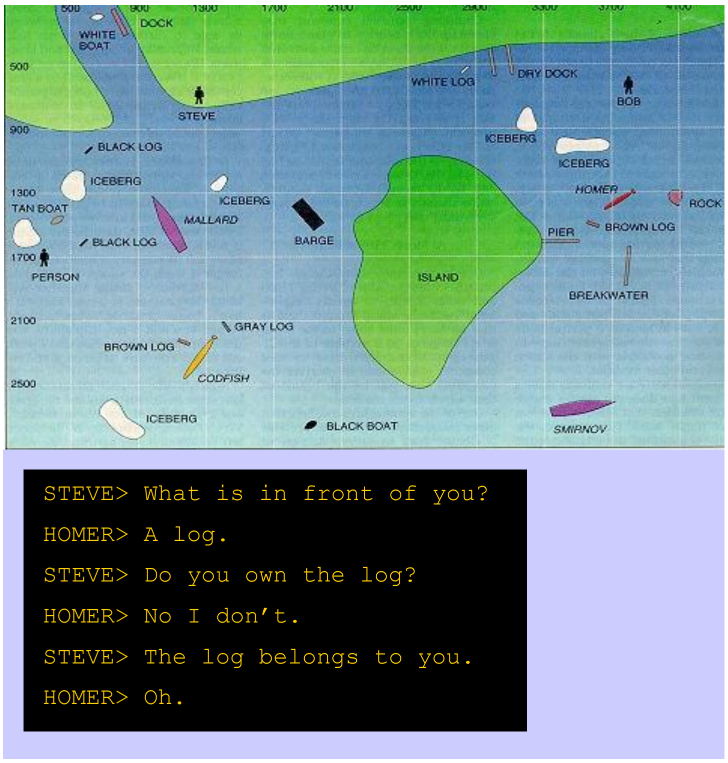
There are other AI reasoning technologies that have also been applicable to agent decision-making, these technologies offer different opportunities, capabilities or constraints to the design or reasoning of agents. A review is given for the technologies that were considered but not developed in this research.

5.5.1 Classical agent reasoning

Classical artificial intelligence systems are based on the symbolic representation and manipulation of knowledge for their decision-making process. The control of classical agent-like systems, such as SHRDLU, STRIPS and NOAH, were based on deliberative plan generation, where the problem-solving follows the sense-plan-act process, the planning problem was described in terms of the state of the world, the desired goal state and a set of operators to effect changes to the world. The knowledge bases maintained by these systems were both the agent's internal model of the environment and the application's simulation of the environment. The systems assumed that the agent had a complete and up to date view of the environment and that no changes occurred in the environment outside of the control of the agent. Little emphasis was placed on the execution of actions so manipulating items in the environment was simply a task of altering a symbolic statement in the knowledge base. In addition the information about the environment was represented as a set of highly abstract, symbolic statements about the environment. Although the systems produced positive results in their environment they suffered the limitation of being less successful when applied to real world environments.

5.5.1.1 Deliberative agents

One of the more sophisticated deliberative agents was the Homer project (Vere & Bickmore 1990), which was an attempt to construct a complete socially aware rational agent that was able to function in a simulated dynamic environment. The environment called Seaworld simulated the activity around a small harbour (see figure 5.4) containing a number of objects such as docks, islets, fish and passing boats. The agent, called Homer, operated as an Autonomous Underwater Vehicle (AUV) able to sense, make plans, perform actions, communicate in a subset of English and reflect upon its activities in the environment.



```
STEVE> What is in front of you?
HOMER> A log.
STEVE> Do you own the log?
HOMER> No I don't.
STEVE> The log belongs to you.
HOMER> Oh.
```

Figure 5.4. The seaworld environment – (*Courtesy of Vere & Bickmore 1990)*

The goal of the developers was to integrate the then technology to develop an autonomous intelligent agent. Homer did address some of the deficiencies of deliberative systems. The knowledge base for the agent and the environment were separate, the agent had limited sensory abilities so was only "aware" of its immediate surroundings and changes could be made to the world outside of the agents knowledge. To allow Homer to function in the environment its reasoning capabilities were built around specialised modules such as a temporal plan generator, an action executor, different types of agent memory for different tasks and a reflective processor. Homer also contained natural language processing modules for communication with human users, including being set goals to achieve and commenting on its activities.

Although Homer is only capable of deliberative processing it is able to react to changes in the environment by re-planning, making changes to the formulated plans in the agent memory to cope with the new information. Homer can be regarded as an advancement on the SHRDLU simulation system, where there was no distinction between the agent's knowledge base and the environment. It was developed with the engineering goal of investigating the state of AI technology by producing a complete agent artefact rather than any particular contribution to research. However more recent research by Liu and Schubert use a similar planner and reasoning engine called ME (for Motivated Explorer) to research linguistic competence in self motivated intelligent agents (Liu & Schubert 2010).

5.5.1.2 Reactive agents

Completely reactive systems are able to rapidly process real world information that is often presented as a stream of data with very little abstraction from the environment. They are said to have advantages such as simplicity, economy and robustness against failure (Wooldridge 2002). However there are a number of difficulties, for example, as decisions are based on local information they are inherently short term and there is no principled methodology for building such agents.

Nils Nilsson proposed the Teleo-Reactive as an architecture for creating goal oriented reactive programs. The Teleo-Reactive (T-R) architecture is a reactive agent control system that directs an agent toward a goal in a manner that continuously takes into account the agent's changing perceptions of its environment. T-R programs are structured as a network of decision-making elements, processing directs an agent toward a goal in a manner that continuously takes into account the agent's changing perceptions of a dynamic environment to select the agent’s action. The programs are written in and interpreted by a production-rule-like condition-action language, where conditions may specify some detectable situation from the environment condition and actions specify agent behaviours. Although rule-based reasoning is generally associated with production systems they may also be used for plan generation and execution. Rules allow agent behaviours to be executed from simple operators rather than a library of pre-coded plans typical of BDI agents. In addition to continuous feedback, T-R programs support parameter binding and recursion. In addition, T-R programs are said to be intuitive and easy to write and are written in a form that is compatible with automatic planning and learning methods (Nilsson 1994). T-R programs have been used in the control of simulated agents and actual mobile robots.

Another example of a completely reactive agent is the subsumption architecture devised by Rodney Brooks, (Brooks 1991) who wanted to explore producing intelligence without the need for elaborate knowledge representation or reasoning. The idea of subsumption is to produce intelligent behaviour from a network of interacting stimuli-response subsystem modules, each of which controls a logically single or simple behaviour. The network of modules are organised into a fixed hierarchy where modules in lower layers represent primitive behaviours such as avoiding obstacles, which are able to override or subsume the behaviour effects from other modules at higher layers that govern more general tasks such as path following. In effect a subsumption architecture forms a software circuit analogous to an electronic circuit, where the operation at any one time is determined by the state of the inputs. There are two mechanisms that allow modules to override the effects of other modules: suppression where the input to a module is blocked, hence preventing it producing a behaviour and inhibition where the output from a module is blocked. The reasoning for module behaviours are implemented as stimulus-response processes typically using condition-action rules and although computationally very simple the subsumption powered machines are capable of producing behaviours that would be regarded as sophisticated if produced by symbolic AI systems.

5.5.2 Practical agent reasoning

5.5.2.1 Hybrid agents

In the last chapter the set of required capabilities for agent systems was specified as: autonomy, reactivity, deliberation and sociability. The processing for these capabilities requires differing resources that are not always complementary. The limitation with deliberative agents is that they are not able to respond quickly to changes or unexpected events in their environment. The limitation with reactive agents is that they are not really capable of pursing a range of goals over a long term. One solution to the differing requirements is to allow different subsystems, or layers, to process the deliberative and reactive requirements separately and then combine results to provide the overall agent behaviour (Müller 1991).

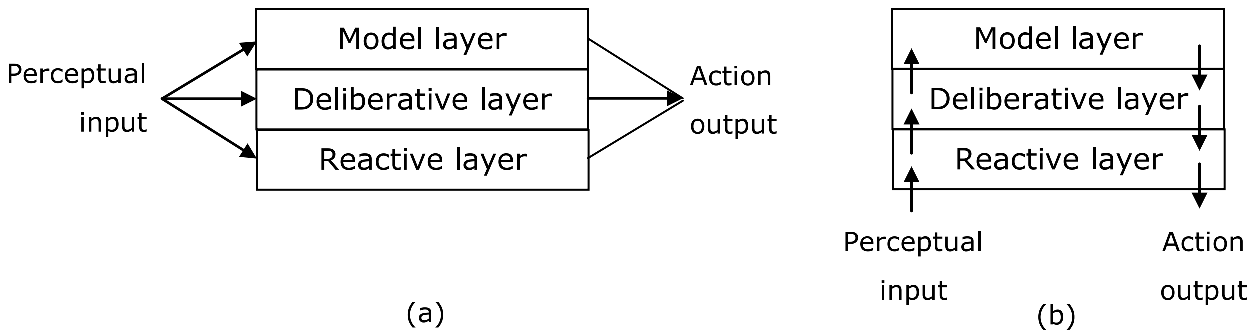


Figure 5.5 Horizontal (a) and vertical (b) information flows in layered agent architecture (*Courtesy of Müller 1991*)

This hybrid arrangement of processing layers allows the agent to produce timely responses to changes in the environment while pursuing longer-term goals. Hybrid agents such as INTERRAP (Müller 1991) and Touring

Machines (Ferguson 1992) are typically constructed with a reactive rapid responding layer, a goal seeking deliberative layer and a third domain specific modelling layer. The major difference between the types of agent is how the layers interact. In INTERRAP the layers are arranged vertically in a hierarchy. All sensory input and action output to the environment is through the reactive layer. If an input requires more processing it can be passed up to the deliberative layer and so on to the model layer, see figure 5.5 (b). If a layer is able to process an item of information the result is passed down the hierarchy where it may affect the operation of a lower layer or produce an action via the reactive layer. In Touring Machines the layers are arranged horizontally. Each layer has sensory input and action output to the environment, see figure 5.5 (a). Information in the agent is processed in parallel by each layer; because of this it is possible for layers to produce contradictory actions so each layer contains a mediation function to inhibit, or be inhibited by, other layers giving control to one layer only at any particular time (Ferguson 1992). The horizontal reasoning, Touring Machines architecture, makes use of suppression and inhibition mechanisms similar to that used in the subsumption architecture to determine which layer controls the agent's behaviour.

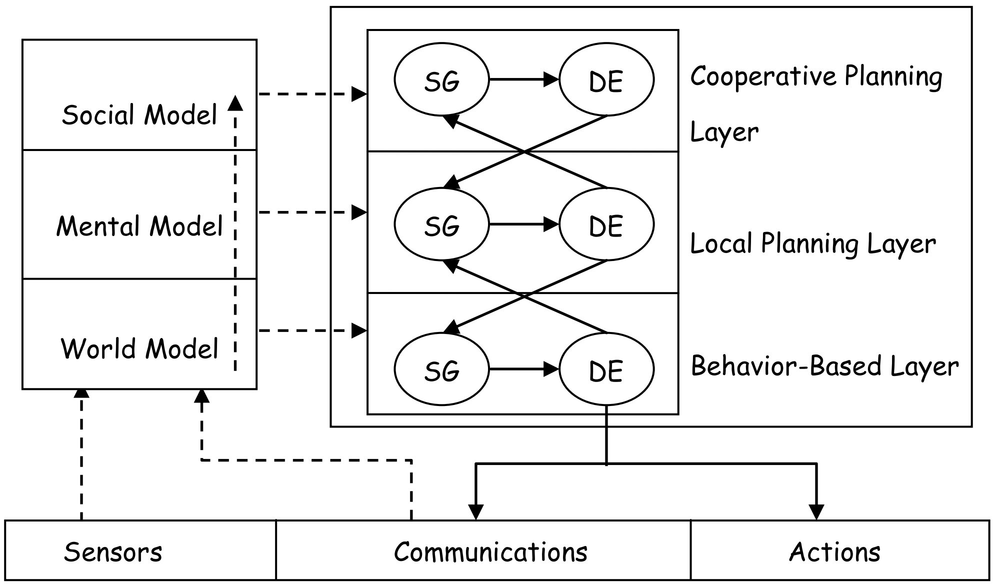


Figure 5.6 The Interrap agent architecture (*Courtesy of Müller 1991*)

The INTERRAP architecture consists of three vertically layered processing areas that each process perceptions from its environment at a different level of abstraction, see figure 5.6. Each layer consists of two processes called SG, for recognising situations and setting goals and the DE process for making decisions and overseeing plan execution. The lowest layer, called the behaviour based layer (BBL), deals with supervising reactive responses to changes in the environment. The middle layer, called the local planning layer (LPL), implements a planner to generate plans required to achieve the proactive goals of the agent. The highest layer, the cooperative planning layer (CPL), governs social interactions with other agents.

Another example of a layered hybrid system is the Prodigy/RAPS architecture developed by Veloso and Rizzo (1998). This consists of two separate reasoning layers. The upper layer is Prodigy, which is a deliberative reasoning system, although it is not clear from the authors whether or not the Prodigy planner includes the Analogy CBR engine for this architecture. The lower layer is based on James Firby's Reaction Action Package system (RAPS), a rule processor, which executes planning goals that are specified as knowledge structures similar to the reactive plans of a BDI architecture. Plans generated by Prodigy are translated into RAPS operators, as the two systems do not share a common syntax, for execution where RAPS controls the pursuit of deliberative and reactive goals without intervention from Prodigy. Another hybrid architecture, called CBR-BDI, combines a BDI planner with a CBR to address some of the limitations of BDI such as the absence of a learning mechanism, the need to recompile the agent knowledge base to add new plans and the efficiency of some implementations (Bajo & Corchado 2005). The architecture is not layered but rather implements the BDI reasoning within the CBR by mapping the BDI knowledge structures onto the cases in the knowledgebase. In a CBR-BDI a case represents the set of beliefs, an intention and a desire, which cause the resolution of a problem (Corchado & Pellicer 2005). The mapping between cases and BDI plans are for the problem component of a case to represent the beliefs, the solution component is equivalent to the intentions and the result represents the desires. Reasoning in the CBR-BDI is performed in the *four REs* process cycle of the CBR engine. It is not clear from the authors how efficiently reactive processing is supported in the architecture compared to other BDI systems, however the agent is able to reason, communicate and learn.

5.5.3 Biologically inspired reasoning methods

Another class of agent reasoning is the reasoning technologies inspired by processes found in nature such as neural networks or genetic algorithms. Rather than representing and manipulating knowledge in the form of symbols as a method of reasoning these systems reason by mimicking biological processes. The systems tend to be self-organising so acquire knowledge by a process of learning rather than from a knowledge base. A genetic algorithm reasons by an evolutionary process of repeated manipulation and evaluation of a population of strings to optimise a search towards a solution. An artificial neural network (ANN) is a programming structure that consists of many simple processing units interconnected in layers to produce specific outputs in response to particular inputs. The ANN is said to mimic the way the brain processes information (Schalkoff 2011) and is very useful for pattern matching and predicting trends in data. A more comprehensive treatment of technologies is available in Schalkoff (2011), Russell and Norvig (1995), and Hopgood (2001). There has been some use of biologically inspired reasoning systems for agent decision making used in applications such as for the control of embodied agents in virtual reality environments (Florian 2003), crowds of people and flocks of birds simulations (Stanley *et al.* 2005).

5.6 Summary

This chapter introduced two reasoning technologies, agent systems and case-based reasoning, which form the basis of the mentor agent system. Agent systems combine different methods of reasoning to satisfy the requirements to be autonomous, to be social, reactive to changes in the environment and able to pursue long-term goals. Although agents may be implemented in different ways those based around the BDI architecture are the most developed and popular. BDI reasoning is a form of planning that provides a method of reasoning that supports both reactive and deliberative processing; it makes use of a library of hierarchical plans to achieve goals. The architecture provides a mechanism for handling the differing requirements from a learner in the desktop environment. The agent has to reconcile information from multiple sources on the desktop, make inferences about the learner's activity, control the agent's interface, coordinate information from the different knowledge sources and respond to commands from the learner. The second technology, CBR, stores records of complete diagnostic situations and provides mechanisms to select and adapt historical cases to supply the closest solution possible to new cases. CBR is analogous to the way humans solve problems by recalling past experience and therefore is used for domains where there are large example sets of decision making data. Traditionally CBR systems are used by a consultation process, where a user presents the properties of problem for diagnosis and a solution is returned. In later chapters these technologies will be brought together to form a cognitive agent architecture where the different reasoning and knowledge sources are integrated to produce the agent mentor. By combining BDI and CBR the BDI will manage the presentation of problems to the CBR making its diagnosis resources available to the learner.

Chapter 6:

The challenges of learning Python – Case Study

6.1 Introduction

This research focuses on the programming language Python, which is the language taught by the researcher at his university, and provides a useful case study as the researcher has access to his students' work and their difficulties. Python is not only a good introductory programming language to first year students but also provides an ideal situation to test the proposed mentoring approach and validate the results. This chapter begins with an analysis of the nature of the errors produced by novice learners and a classification of the programming errors encountered by novice programmers in Python. It is followed by a brief overview of the features of Python to explain why it is used as a teaching tool. There is then an explanation of the different schemes that may be used to characterise programming language errors before a detailed examination of the observed learner errors is given within the scheme chosen as the most appropriate.

6.2 Difficulties in learning to program

Although this chapter is concerned with the domain of python programming errors it is worth examining whether errors occur irrespective of any particular programming language. In section 2.2.1 a review was made of the literature related to the psychology of the novice programmer and why errors are made. The literature summarised the source of novice errors as from two causes: *fragile knowledge* where the learner is aware of the required information but fails to see the opportunity to use it and *neglected strategies* where students do not use techniques to gain further understanding of the problem they are solving. Both these causes are related to the difficulties of understanding the semantics and the logic of code, and independent of the syntax of any particular language. However, the syntax of a language has an influence on how easy it is for a programmer to introduce errors.

As will be explored in sections 6.4 and 6.5, syntax errors account for most of the errors made by novice programmers. As the design of a language influences the range of real-world developments it may be used for there are many non-scholarly Internet debates comparing the design of programming languages and the influence of different syntax on error rates. More scholarly sources have examined novice errors while learning a range of prominent programming languages such as BASIC (Mayer 1981), LISP (Gray et al 1988), Pascal (Ueno 1998), Smalltalk (Xu and Chee 1999), LOGO (Glezou and Grigoriadou 2007), C/C++ (Kummerfeld and Kay 2006, Gobil, et al 2009), and Java (Jadud 2004, Traynor and Gibson 2004, Thompson 2006). One of the scholarly sources McIver and Conway (1996) examined the design of programming languages suitable for teaching and summarised three types of syntactic and semantic constructs they termed "grammatical traps" that impede the novice programmers. They are:

- Syntactic synonyms – in which two or more syntactic forms are available to refer to a single construct,
- Syntactic homonyms – a syntactic form that has two or more semantics depending on context and
- Elision – the optional inclusion of a syntactic component.

The researches also identified other language design issues such as:

- Hardware dependence – where programmers have to specify storage class of data (often merely for the convenience of the compiler writer),
- Backward compatibility – including features for historical reasons,
- Excess of features – languages support many more features than required for teaching that are used for real-world application development,
- Excessive cleverness – features that cause misunderstanding at the novice level but are considered obvious to experienced programmers and

- Violation of expectation – there is no reason why the protocols of a programming language should appear obvious or natural to a novice.

While tools such as syntax highlighting editors, reviewed in section 2.3.1, are shown to aid the productivity of experienced programmers evidence of a similar increase with novice programmers is unclear (Green 1989). The reason why a given language has relatively little effect on the types of novice errors observed is because any programming language is essentially a protocol for communicating commands to a computer. The differences between programming languages are influenced more by their purpose and method of evaluation within the computer. Novice programmers face two major obstacles in learning a new language: firstly, there are no everyday intellectual activities that are analogous to programming and secondly, programs operate on a notional machine (albeit in a physical machine) whose function and operation remains opaque to the learner (Rogalski & Samurcay 1990). Novice programmers face the same difficulty with the syntax of any programming language as they do with the semantics and logic of program design that of *fragile knowledge*. They will often have yet to acquire required information *missing knowledge*, lack the experience of when to use information *inert knowledge* or use what they have in the wrong context *misplaced knowledge*. The difficulty is further compounded by having to learn the multiple skills of the syntax, semantics and logic of program design in parallel, each reliant on the other to produce error free code.

6.3 The properties of Python

Python is an object oriented scripting language developed by Guido Van Rossum in the 1980s with the aim of being easy to learn and easy for rapid application development. The Python programming language is the main development tool used to teach programming to the students in the "Foundations of programming" module for the Information Sciences course at the University of Northampton. A more detailed explanation of the Python programming language is given in appendix-A. There are a number of features of Python that make it an attractive choice as a software development tool and a suitable language for teaching:

1. Support for multiple coding styles, i.e. scripting, procedural programming, object-oriented development;
2. Automatic memory management;
3. Dynamic data typing;
4. Simple syntax, few keywords and indentation for block delimitation;
5. Rich set of data types – integers, floats, strings, lists, association lists, sets, etc.;
6. Interactive interpreted (compiles to byte-code) programming environment - suitable for rapid application development;
7. Large set of third party code library;
8. Widely used in the networking and computing industry.

In terms of programming languages Python is conventional in many ways. The most distinguishable feature of Python is its use of indentation to mark the beginning and end of sections of code, which coupled with its dynamic data typing, avoiding the need to declare the data type for variables when writing code, provides for a brevity in its notation. The general impression given of Python code is as a kind of executable pseudo-code; in the book *Artificial Intelligence for Games* (Millington 2006) the author acknowledges the similarity of the notation of the pseudo-code examples given to Python. The computer scientist Peter Norvig wrote on his web site of a similar observation when converting lisp programs to Python for his book, *Artificial Intelligence: a modern approach* (Russell and Norvig 1995). It is the pseudo-code like features that make Python easy to read and easy for non-programmers to learn. The step from a design to implementing code reduces the cognitive load of the learner having to remember large amounts of detailed punctuation, such as where to place a semi-colon or a bracket.

The professional computing community has used Python in many applications, often as a configuration or prototyping tool, but also in deliverable products. The Python web site lists about 60 such applications written wholly in Python or using Python to drive or configure the application. It is worth noting that there are some limitations with Python. Although semi-compiled and executed in a virtual machine it is relatively slow compared to rival programming languages such as Java, Perl and Lua, it does not produce easily portable compiled object code like Java class files and compatibility is not supported between different versions of the Python run-time environment. Therefore Python applications often include the entire run-time environment when distributed.

6.4 Observation of novice errors

The novice learners were students from a year one undergraduate university course who had to complete an introduction to programming module as a compulsory component of a business computing degree course. The module was designed to offer the students insight into the production of software applications and to develop the student's skills in areas including problem solving and working in teams. The average number of students per cohort was between 20 and 25 with ages starting from 18 years old upwards and an average age of 21. The module was taught over a twenty-four week period consisting of a weekly one-hour lecture where an introduction to some aspect of programming was examined followed by a ninety-minute practical session where supervision was given while students worked through a set of related programming exercises, to reinforce the topic introduced in the lecture. However, for many students computer programming was not the primary interest of their study and the level of motivation was variable. As programming is a skill based activity that relies on building new knowledge upon old, students who had difficulty with the beginner level concepts and exercises had even greater difficulty with the later intermediate level and advanced level exercises.

6.4.1 Method

The observations were carried out using five techniques to gather sufficient information about the errors made. Care was taken to observe ethical considerations and none of the techniques involved interfered with the learning process. The first method was to observe the learners during normal practical sessions where students carried out programming exercises. Due to time commitments and the desire to reduce classroom disruption it was not possible to make contemporaneous detailed notes, but notes were recorded at the end of most sessions. In this way two or three original (that is to say not recorded previously) errors were generated from each practical session. The second method was to run one-to-one tutorial sessions with three student volunteers from the cohort where similar programming exercises to those in the practical were carried out and notes could be made as the student worked through the problem. This approach allowed a more detailed record to be made: the chronology of how novices approached problem solving and questions to be asked as to why certain decisions were made. The third method was to review the assignment work submitted by students and categorise the different solutions used – what worked and what problems they were unable to solve properly. The fourth method was to offer an email consultation service to the students where they could email questions describing problems they had encountered and a solution returned. This allowed a record to be made of the way students think about and express problems. One final source for information on novice errors was literature from third parties; this was often in the form of error finding (debugging) hints that accompanied Python programming tutorials and allowed for different sets of problems that would occur from different types of teaching materials.

6.4.2 Categories of programming errors

The purpose of undertaking the observations of novice programmer errors was to identify the range and types of learner mistakes with the aim of finding ways to rapidly identify the source and possible solution. These observations form the basis of the knowledge for the mentoring agent and so the domain knowledge for the agent, the categories therefore needed to reflect how the errors would be used to determine the program cause.

One approach to categorising the errors observed would be to organise them in terms of the types of programming statements they represent and to have the errors treated as variations on the legitimate statement. This would allow the assessment of student code to be made by comparison against legitimate statements. This method of diagnosis can be called *source-to-source* comparison (Chee & Xu 1998) and is the method used in SIPLeS discussed in the literature. The limitation of this approach is that, assuming a mentor agent would provide assistance when the learner had produced an error. It would lead to the mentor performing a substantial amount of analysis on code that had already been analysed by the Python environment.

As Python is a loosely typed language, variables do not have a type and the data type of operations can therefore only be determined at runtime. This means that the static analysis of the syntax of a program cannot determine some types of error. The Python interpreter makes a distinction between the way it treats errors that occur when compiling the source code, syntax errors and those that occur when the program is being executed – these are as a result of the semantics of the program. This would appear to be a logical way to categorise programming faults as it is the same way the programmer experiences them and skilled programmers are able to reason about and correct faults using this level of information. There are also some errors that do not fall into the category of syntax error or semantic error, but produce an unexpected or incorrect output. These errors will be placed in the category of logical errors.

6.4.2.1 Syntax errors

The **syntax error** category is where the rules of the language have been broken so the meaning of statements and expressions cannot be properly interpreted. Syntax errors in formal languages such as those used for programming are more likely than in natural languages for two reasons: the syntax rules are less flexible and the semantics of parts of many programming languages are carried by the use of more non-alphanumeric symbols than those in natural languages. In terms of this analysis syntax errors are those that prevent the successful compilation of a Python script. There are a number of errors that will be uncovered as syntax errors in strictly typed languages that, because of the nature of Python will only become apparent at runtime in a Python development.

	Example	Description	Notes
6.1.1	**if food == "spam"**	Missing colon from end of statement	
6.1.2	**print "hello" name**	Missing comma between terms	Print can handle a single argument or a comma separated list
6.1.3	**Test = [alpha, beta gamma]**	Missing comma between terms	A list should contain comma separated items
6.1.4	**if test(max(x,6):**	Unbalanced parentheses missing)	
6.1.5	**x = 1 + 2 y = m * x + c**	Missing operator between 2 and y	These are two lines of code and should be separated by a new line or semicolon
6.1.6	**If food == "spam":**	Upper case letter used in keyword 'if'	
6.1.7	**if food = "spam":**	Assignment operator rather than test for equality	The = means "becomes equal to" in Python
6.1.8	**Ifval == 123:**	Missing space after if keyword	Words must be ended by a space or non-alphanumeric character
6.1.9	**def say_hello(): print "Hello World"**	No indentation in line after the colon ended line	Produces an indentation error
6.1.10	**day = day + 1 print "start of the weekend"**	Rogue alignment of statements	Variation of error 6.1.9 but produces a syntax error
6.1.11	**def name(arg1 * arg2):**	Illegal operator in argument list	
6.1.12	**def na me():**	Illegal space in function name	The names of items in Python must be a single word
6.1.13	**def = name(arg1):**	Illegal syntax in function definition	
6.1.14	**def name(arg1 arg2):**	Missing comma in argument list	Variation of error 6.1.3
6.1.15	**def "name"(arg1, arg2):**	Quotes not permitted around function name	
6.1.16	**def__init__(self):**	Missing space after def keyword	Variation of error 6.1.8
6.1.17	**print "please press enter'**	Different symbols to delimit string constant	
6.1.18	**import "string"**	Module name should not be a string	
6.1.19	**class = "month"**	Use of a keyword as a variable	
6.1.20	**int(calc_area(width,10)**	Unclosed bracket	Usually flagged on line following

Table 6.1 Syntax Errors

For example, because there is no variable declaration the compiler is unable to detect the incorrect spelling of a variable name. Although most novice errors originate from minor causes, such as the incorrect use of punctuation the effect can be quite critical to their progress through a problem. Most errors are as a result of *fragile knowledge* and have trivial solutions: the inclusion of a missing symbol, or the substitution of a correct piece of punctuation, etc. In table 6.1 are examples of the observed errors that prevented compilation of Python code. It is worth noting that none of the errors is particularly complex, usually requiring the addition or the changing of a single character. Some learners are able to locate and correct them by themselves, but where they are unable to, these errors greatly restrict further learning.

One of the first types of error to be observed (and one that would continue to occur regularly) was the missing out of punctuation symbols (or non-alphanumeric), characters or the format of Python code. The most commonly missed symbols were, for example, the comma separator between multiple arguments in print statements (table 6.1, error 6.1.14) and missing the colon at the end of a program structure defining line such as def, if, while, etc. (table 6.1, error 6.1.1). The comma separator was the symbol most often missed. With most of the other errors the learner could determine the fault as long as the location of the error was pointed out. This is an example of *inert knowledge*, although students were often unable to determine the cause of the error if the missing symbol was a comma, an example of *missing knowledge*. In an example of misplaced knowledge there was often confusion between the use of the equals symbol for a test for equality or to assign a value, but students were often able to correct the problem by themselves. There were no errors with arithmetic operators, however comparisons operators such as less-than and greater-than were often confused for one another, and became apparent as a logical error, (see table 6.3, error 6.3.2). The observation of the learners' treatment of symbols is that different punctuation and operator symbols carry different amounts of meaning for individuals. The four arithmetic operators posed little difficulty but after that, less familiar symbols including commas and parentheses caused some to make errors.

The other commonly occurring error was difficulty in handling the level or degree of indentation. Observations noted mistakes even when students were tasked to type in some code from a pre-prepared program code (see table 6.1, errors 6.1.9 and 6.1.10). Managing white-space characters is more important in Python than with other languages as they are used to delimit blocks of code. The most frequent error with white spaces, made by novices, is to not include them; this is probably as an attempt to avoid potential errors but is particularly unproductive. Incorrect indentation is potentially a more difficult problem to diagnose and treat because it relates to the student's understanding of how the program is supposed to work. Even when copying a piece of code some students will alter the indentation and are surprised at the level of accuracy required to reproduce the working code. This is consistent with McIver and Conway (1996) who categorise white-space block delimiting as a feature of excessive cleverness.

6.4.2.2 Semantic errors

Once a program is in a state where its code is syntactically correct the next level of errors that may occur are **semantic errors**, these are statements that are legal, but they have an error in meaning that will cause the program to fail when it is run. Semantic errors are usually generated by an incompatible operation for a particular type of data. These errors are sometimes only detectable when a program is processing data and thus are usually detected at runtime. Strictly typed programming languages provide a margin of security against some semantic errors, but Python is a weak typing mechanism (the language designers preferred the increased flexibility for its data handling in weakly typed language). Exception handling is another mechanism available to a programming language to allow the application to catch errors that occur at runtime within the application and if possible to take remedial action to deal with them. For the purposes of this analysis a limit is going to be placed on the definition of a semantic error as one that causes a runtime error such that a Python program would not be able to complete its execution. Other authorities may have a different definition of the semantic error.

	Example	Description	Notes
6.2.1	**y = 0** **result = x / y**	Division by zero error	The zero is usually arrived at by a longer calculation
6.2.2	**result = “123” – “456”**	Type error operation, subtraction, is not legal for strings	String concatenation by use of the addition is legal
6.2.3	**Sum = m * x + c** **print “the answer is”, result**	Variable name ‘result’ is not defined	Usually as a result of copying example code without adaptation
6.2.4	**current = week * 7.0 + day** **. . .** **.** **today = days[current]**	Type error as array indexes must be an integer value	
6.2.5	**noOfDwarves = 7** **. . .** **.** **boots = 2 * noOfDwarfs**	Name error noOfDwarfs is an unrecognised variable	
6.2.6	**name = graham**	Name error it is unclear whether graham is to be a variable with a value or literally the word “graham”	The line of code needs to be analysed in context as it might produce a syntax error, a semantic error or no error

6.2.7	**def foo(arg):**
	    …
	.
	foo()
		Type error exception missing argument in function call	

Table 6.2 Semantic Errors

The name error exception outlined in table 6.2 usually occurs for a number of reasons, from simple reasons such as failure to initialise a variable or a spelling mistake, to more subtle reasons like the mixing of cases (see table 6.4 below). However homophones, such as illustrated in table 6.2, error 6.2.5, where dwarves and dwarfs become confused, support the theory that novice programmers are more concerned with meaning than with representation and some novices incorrectly presume the computer capable of providing more human-like levels of interpretation. The differences can remain opaque to the novice until they are encouraged to check each spelling letter by letter.

For error 6.2.6 (table 6.2) the absence of quotes means the interpreter evaluates the word "graham" as being a variable and not finding one would cause the program to raise an exception. A run-time error message accurately reports a name error saying that the graham variable (in this instance) has not been defined but from the error message students are often unable to understand why the error has occurred and so how to proceed to correct it. It is notable that this error occurs more often when the constant value being assigned is a single word. For some reason the space in a phrase or sentence acts as a prompt for the correct delimitation. Both of the errors above indicate that, even after being shown how to create different data types, some learners tend to pay attention to the largest portion of data constants to determine the meaning. This error occurs even in the presence of editors with colour syntax highlighting, which might indicate that while syntax colouring is noted to be more of an aid to experienced programmers its purpose appears to be opaque to the untrained eye of the programming novice. Although type errors are some of the earliest mistakes made they tend to produce semantic or logical errors. The learners who have difficulty with types often mistakenly expect the programming language to have more human-like levels of interpreting meaning called Egocentrism (Pea 1986).

In addition to learning the core of the language learners are introduced to programming concepts that start to illustrate some of the purpose of programming with more real-world application examples for their practice. To do this the course introduces the student to two new concepts, which can influence some semantic errors; the concepts are:

1) *Modules:* The introduction of Python modules allows the learners to develop two new resources: first it allows for larger programs with code spread across a number of files, and second it introduces the use of third-party code libraries for access to different applications such as database access via an ODBC library

and more importantly writing GUI applications via the Python version of the TCL/Tk interface, called Tkinter. Python allows for a number of formats for the import of modules, affecting what resources are imported and how the resources are addressed. The addressing code (and data) from other modules introduces the concept of the dot notation for names, used extensively in object-oriented and object based programming discussed in the next section. Although the introduction of modules allows for many potential errors the one that students regularly make is handling the case sensitivity for the imported file names.

2) *Object-orientation:* Although object-oriented programming is optional in Python scripts and learners are not expected to develop any object-oriented programs, with the use of third party code libraries, especially the Tkinter GUI library, object-based programming, where objects are made use of would become necessary. Students were given a brief introduction to the general concepts behind object-orientation, such as encapsulation and inheritance, an explanation of the terminology, such as the difference between a class and an instance and a look at how Python implements such features. The most important feature the students needed to understand was the creation of an object before making use of its functionality; this was mainly done using the Tkinter window objects, called widgets. The use of window objects meant that changes to underlying code often produced an immediate visual effect on the application so students made fewer errors than expected (or were able to correct them without tutor intervention) even though there was a substantial increase in the complexity of the code being developed. The same degree of competence did not appear when working with database access via the ODBC library, which would lead to the inference that the visual confirmation offered from the Tkinter widgets had a substantial effect on their understanding. The most frequently occurring error appeared to be case confusion when creating Tkinter widgets, the writers of Tkinter adhere to the convention in the object-oriented programming community of spelling class names with a capitalised

first letter and all other names to begin with a lower case letter; so some learners would find their program producing a runtime error for an undefined function, say “frame” rather than having produced an object from the class “Frame”. Other difficulties arose from manipulating objects once created: first, in not creating new variable names to hold different instances of objects. So learners would call all their Button widget instances say “b1” and be unsure why only one button would appear on their application even though they had intended more. Second, the requirement for objects to be configured after creation was also a source of errors. It is not clear if this was because variables with simple data types do not require further initialisation, or solely the peculiarities of the Tkinter programming interface. The operation most often forgotten by the learner was to pack (the Tkinter name for placing) the widget into the application window.

6.4.2.3 Logical errors

The third type of programming error is the **logical error** where there are no errors in the code that prevent a program from executing, but rather faults that prohibit the production of the required or meaningful output. Logical errors can be difficult to detect from analysis of the code alone, as there often must be an understanding of the difference between the code produced by the programmers and the requirements of the problem to indicate what may be missing. For instance Python requires the name of a function to be followed by parenthesise when call is being made to it, however functions first class object, meaning the function name acts as a variable and its value (a function object) may be passed as an item of data in which case the parenthesise are not used. The use of either format is fully legal and depends on the logic of the problem and the intention of the programmer. There are, however, some attributes that can be searched for that would be expected to be in most novice level programs such as the program containing a structure where there is initialisation, processing and termination. Each phase would be expected to contain a typical set of activities such as the initialisation or input of data in the initialisation phase, a processing phase where there is a relation between the input data as some result and the termination phase where the results are usually presented to the user.

	Example	Description	Notes
6.3.1	**raw_input(“prompt>> ”)**	No destination for input value	Not an error if awaiting an Enter (often to pause a program)
6.3.2	**If a>10 and a<0:**	No value may be both less than 0 and greater than 10	The results of any test will therefore always be true (or false)
6.3.3	**for each in myList:** **print myList**	Use of wrong variable in a for statement	

6.3.4	**for count in range(len(myList)):**
	    sum = sum + myList[count]
	    ave = sum/count
		Code misplacement the calculation of average should not be in the loop	

6.3.5	**def f1(v1):**
	    if v1 > 10:
	        v2 = 2 + 3 * 4 / 7 << 3;
	        return v2
	    else:
	        return 7
	    return 0
		Unreachable code	Zero is never returned as v1 is either greater than 10 or not so there can be no third option

6.3.6	**User = raw_input**	missing brackets for a function call	Python executes this as an assignment of the identity of the function
6.3.7	**count + 1**	No destination for an expression result	The user usually means to increment count by one

6.3.8	**if 3 > 2:**
	    print “Answer is True”
		Both sides of the test are constant values	The result will always be true (or false)

6.3.9	**data = [‘string message’]**	Incorrect data type specified	Here the intention was to process a string

Table 6.3 Logical Errors

Logical errors are very difficult to define and therefore difficult to detect also. The reason for this is that the logical purpose of a program’s statements also depends on the context of its use; for example the use of the raw_input function pauses a running program and awaits some input from the computer keyboard before continuing execution. An optional prompt message may be passed to the function to specify the information requested and the result can then be assigned to a variable. However on some occasions no information need be returned from the input (for instance to confirm when the user is ready to proceed) so no variable is required for the result. The need for a destination variable for the input depends on the context of the input. The determination of its presence requires an overall understanding of the purpose of a piece of code.

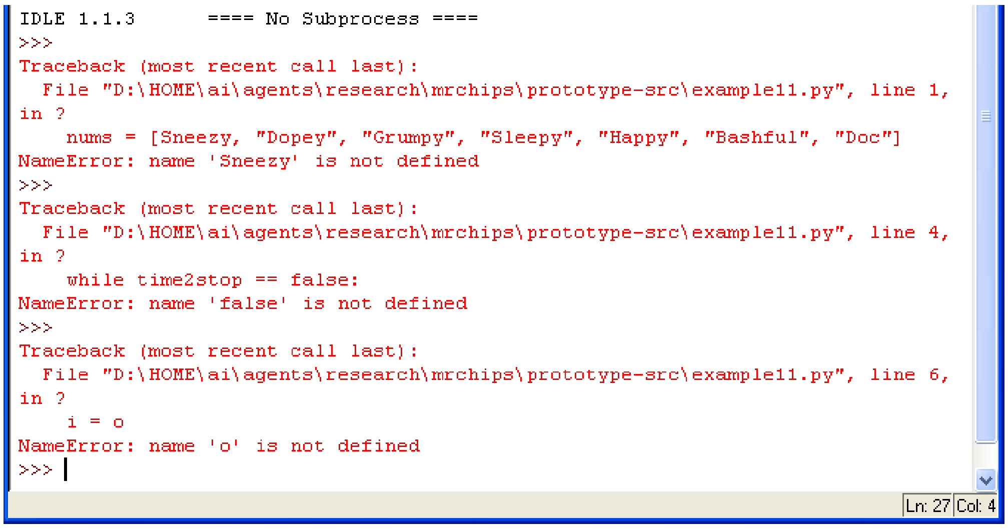
6.4.2.4 Strategic errors

There are a number of other errors observed that do not arise so much from the code written by learners but more from their approach to writing code. These have been placed in here in a category of their own and are included here as they relate to methods of cognitive apprenticeship but may not be directly addressed by the mentoring agent.

1) *Slow Rate of work:* The rate of work from an individual is consistently slower than the average rate of progress within a cohort because they do not engage with practical exercises. An individual's output in performance may vary greatly from session to session and it might mean nothing or even be an indication of taking the time to learn. However a sustained low level of output might indicate a student who is struggling or will come to struggle as they miss a proportion of the learning experience.
2) *Programming as a typing exercise:* This can be indicated by a learner who constantly finishes exercises more quickly than the average; where the individual is happy to type in and run example programs but reluctant to change or experiment. The learner presumes speed is a measure of progress, but takes little opportunity to reflect and understand. They start to struggle as scaffolding is removed.
3) *Reluctance to compile:* Novice programmers are encouraged to compile and run their programs regularly as an aid to

understanding the effect of each incremental change. Some learners may adopt the strategy of writing as much code as possible before attempting to run it and do all the corrections in a single step. While there may be efficiency in performing these tasks in a batch there is usually a penalty to pay in terms of understanding.

4) *Ignoring error messages:* In the Python environment when an exception is raised the program is halted and a record of the call stack is printed to the screen as it is unwound. This means the oldest information is printed at the top of the screen and information related to the cause of the exception is towards the bottom (see Figure 6.1). A number of novices who attempt to find feedback from stack output have been observed to read error messages from the top of the screen and often fail to make sense of the information presented because they cannot see anything relevant so are unable to determine the nature or location of the error.



```
IDLE 1.1.3      ==== No Subprocess ====
>>> 
Traceback (most recent call last):
  File "D:\HOME\ai\agents\research\mrchips\prototype-src\example11.py", line 1,
in ?
    nums = [Sneezy, "Dopey", "Grumpy", "Sleepy", "Happy", "Bashful", "Doc"]
NameError: name 'Sneezy' is not defined
>>> 
Traceback (most recent call last):
  File "D:\HOME\ai\agents\research\mrchips\prototype-src\example11.py", line 4,
in ?
    while time2stop == false:
NameError: name 'false' is not defined
>>> 
Traceback (most recent call last):
  File "D:\HOME\ai\agents\research\mrchips\prototype-src\example11.py", line 6,
in ?
    i = o
NameError: name 'o' is not defined
>>> |
```

Figure 6.1. Runtime-errors in the Python shell window

6.4.2.5 Errors arising from incorrect use of letter-case in Python

One of the particular properties of Python is that it is at the same time loosely typed and case sensitive. For this reason it is possible for some errors to cause a symptom in more than one category depending on where the error occurs in the code. For instance the incorrect use of letter case in a Python keyword would cause a syntax error. If error occurs in the name of an item of data, such as a variable name it would cause a runtime exception error. The way the errors are detected is different, but in both cases the cause of and the solution to the problem are precisely the same – the correct case should be used.

	Example	Correct form	Notes
6.4.1	**Def foo(arg1, arg2):**	**def foo(arg1, arg2):**	Produces an invalid syntax compile error
6.4.2	**import tkinter**	**import Tkinter**	Produces a file not found runtime exception
6.4.3	**if current_drive == “c:”:**	**If current_drive == “C:”:**	Representation in the data can obscure the expected interpretation

Table 6.4 Case sensitivity error types

The most frequently occurring mistake that caused errors in all three categories was caused by incorrect use of cases, illustrated in table 6.4. The choice of case sensitivity in a programming language depends on the purpose of the original language designers. Languages that are designed for teaching such as LOGO tend to be case insensitive, whereas languages used for application development tend to be case sensitive, but may still be used for teaching, Python and Java for example, although the designers of Alice thought it an important enough issue to modify the version of Python that was shipped with Alice to be case insensitive. They gave the argument:

> While we, as programmers, were comfortable with this language feature, our user community suffered much confusion over it. [...] Case sensitivity is an artificial rule that fights against older knowledge that novice users have, namely that while forward and FORWARD may look different, they should at least mean the same thing (Conway 1997).

Another type of error that is reported differently are those caused by incorrect alignment or indentation of code; failure to indent correctly are reported as a syntax errors while inconsistent indentation raises runtime errors. The way the errors are detected is different but in both cases the cause of and the solution to the problem are the same – the code should be properly aligned.

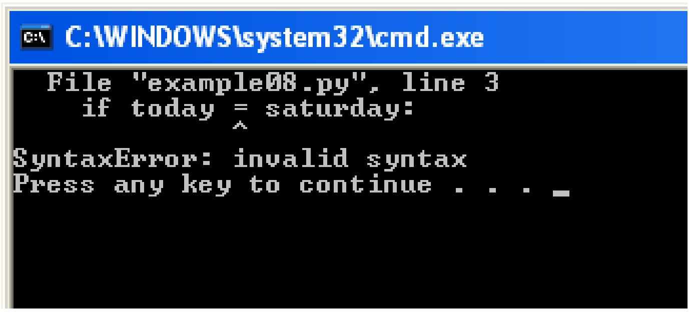
6.4.3 Recognition of errors

So given a problem the programmer must be able to categorise a sufficient number of features of the code to determine its likely cause. The first clue is when the problem occurs because that determines which strategy to use for the rest of the analysis of the problem.

```
today = raw_input("What day is it?  ")
.
if today = saturday:
   print "Hurrah it's the weekend"
```

Figure 6.2. Listing of a faulty Python to be debugged

As determined from the observations although the causes may be varied errors eventually manifest in the Python environment belonging to one of three categories; syntactic, semantic or logical errors. Due to the way Python is compiled and interpreted if a coding fault exists that may cause errors in more than one category it will always be expressed as type syntax error before type semantic error and type semantic error before type logical error. To illustrate debugging in Python the result of processing the program code above in figure 6.2 is examined. Note that only the relevant lines are shown for brevity, in most instances the lines will exist as a more substantial module of code. Presenting the code above to the Python interpreter would generate a compile error because it first violates the syntax rules for Python.

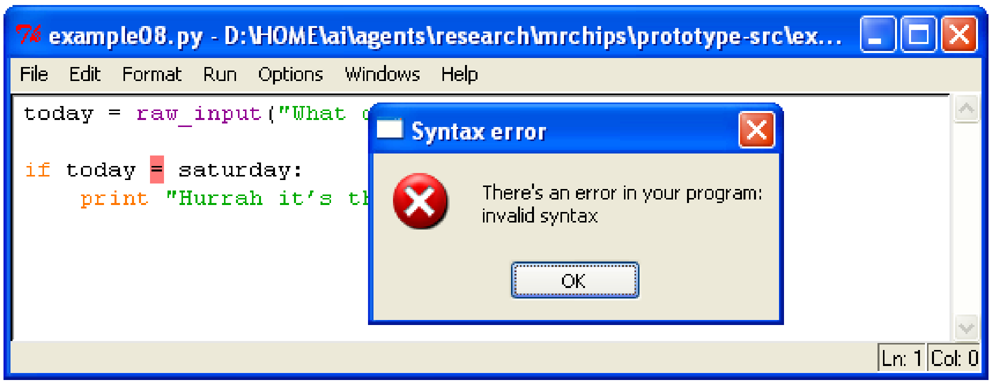


```
C:\WINDOWS\system32\cmd.exe
  File "example08.py", line 3
    if today = saturday:
             ^
SyntaxError: invalid syntax
Press any key to continue . . . _
```

Figure 6.3. Console error output for the faulty Python code

The error message would be displayed to the console as illustrated in figure 6.3 or in the case of using an IDE the editor would produce a dialog box window and highlight the symbol at fault as shown in figure

6.4. Although the error output would be enough for an experienced programmer to determine the source and a likely correction, a novice may require more guidance, which is not provided by the interpreter. The error message would indicate that the equality symbol "=", in the line beginning "if today…" is at fault.


example08.py - D:\HOME\ai\agents\research\mrchips\prototype-src\ex...
File Edit Format Run Options Windows Help
Ln: 1 Col: 0
Syntax error
OK

```
today = raw_input("What c
if today = saturday:
    print "Hurrah it's th
```

There's an error in your program:
invalid syntax

Figure 6.4. Windowed error output for the faulty Python code

As the error is with the syntax it becomes a matter of checking the code against the rules for the language. The interpreter has given the line of the error and the offending component. In this case it is an if-statement and the equality symbol. A check of the rules of the language (see Appendix A) would indicate the if-statement expects a test expression (i.e. that will evaluate to a Boolean value), that the single equals symbol in "today = Saturday" makes it an assignment statement. In Python statements are not allowed in place of expressions, and the closest similar operator used in test expressions is the double equals symbol "==" which is the test for equality. However as it is a frequently occurring error from novice programmers the source of the error can be determined without the need for reasoning from the rules of the language syntax given the clues syntax error, if-statement and the equals-symbol.

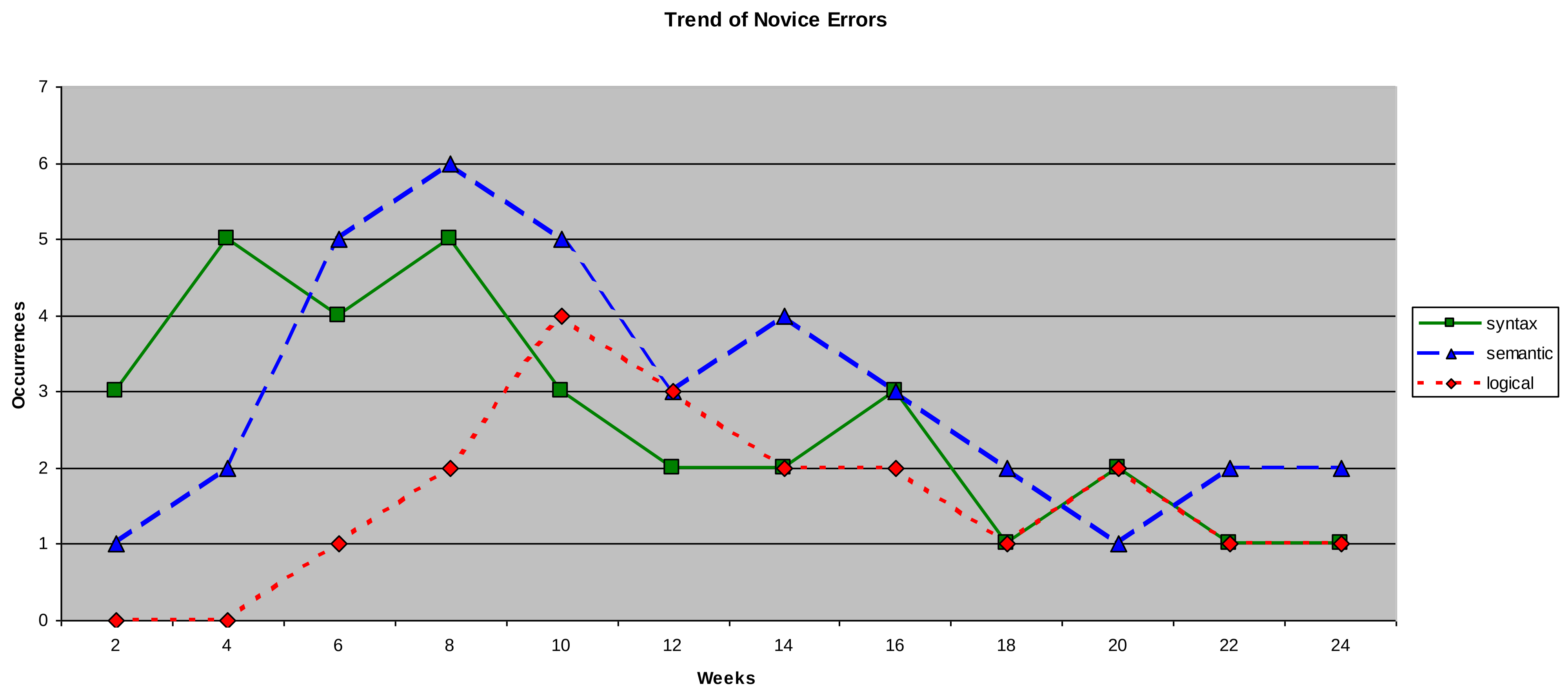


Figure 6.5. Chart of the trend of Python novice errors over one year - *based on a cohort 20 students*

Once the code has been corrected (the correct equality check has been inserted) and assuming no other syntax errors, executing the Python code will run the program until completion or a runtime error is detected. Using the corrected example code above, executing would run the programme until the if-statement line where it would produce a runtime exception indicating that 'saturday' was undefined. There are usually two sources for this type of error: first, that the offending item of data is an undefined variable. This might be due to the need to define a variable, but it might also be due to a spelling mistake including the mixing of cases in different occurrences of the name. The second reason is that the data is meant to be a literal value and needs to be surrounded by quotes. The point is from a runtime error while the source of the error is as easy to determine as with syntax errors the range of solutions for semantic errors is increased.

Detailed figures for the numbers of each type of error that occurred were not kept as the observations were carried out during the running of the computing practical classes, addressing the needs of the students had to be given priority. However a record was kept for the weeks on which errors occurred, this information can be used to give an indication of the different rate for each category error and any trend over time. From the chart, in figure 6.5, it can be seen that syntax errors are encountered first, closely followed by semantic errors. While syntax errors begin to decline after week 6, the rate of semantic errors were more persistent. The logical errors began to occur later than the others and occurred at a lower rate than the others.

6.5 Related work

Other languages have been used as the basis of research into the difficulties faced by novice programmers. Thompson (2006) identifies 4 categories of error for Java programmers, syntactic errors, semantic errors and logical errors. She then distinguishes the run-time error as a separate category of semantic error. This is possible because Java is strictly typed. The compiler is able to identify some semantic errors during compilation; those it cannot find occur at run-time. This is different in Python as all type checking occurs at run-time so all semantic errors are run-time errors. Jadud (2004) found that 60% of novice Java errors were from 4 sources (illustrated in figure 6.6): missing semicolons, spelling mistakes in variable and class names, missing brackets and illegal start of expressions usually caused by the missing brackets or semicolons errors of previous statements. Gobil, et al (2009) used C++ in their research with novice programmers, concentrating on the semantics of code for their ability to follow the path of execution in selection (if…else) statements. They also observed that the novices had difficulty dealing with basic syntax similar to those with Java (both languages share a similar syntax), but did not indicate how learners were able to progress to the semantic problem solving.

```
day++
System.out.println(“start of the weekend”);
```

(a) Missing semicolon

```
int noOfDwarves = 7;
. . .
.
boots = 2 * noOfDwarfs;
```

(b) Misspelled name

```
for (int count=0; count < myList.length; count++{
  sum = sum + myList[count];
}
```

(c) Missing bracket

```
day++;
if (day == 6)
   System.out.println(“start of the weekend”);

}
```

(d) Illegal start of expression

Figure 6.6. Examples of frequently reported novice programming errors in Java

In a series of student assessments the researchers found novices had difficulty understanding how expressions and assignments alter a program's memory, comprehending the limits of a selection statement, following the likely path of execution through a selection statement and the importance of the correct sequence of instruction.

6.6 Summary

This study has been primarily concerned with understanding and classifying the diversity of errors faced by novice learners. These errors were classified in three categories, namely syntactical, semantic and logical errors; these will provide the framework for building the animated pedagogical agent, MRCHIPS, which is introduced in the next chapter. The cognitive or psychological reasons for producing an error are likely to be less informative than the class of the error, but it is worth noting that these errors are produced by novices learning Python and similar sets of errors would also be produced when learning any other programming language. The study of the literature identified the difficulty in learning to program as a result of not having a real world analogue to the activity of programming, in learning two concepts simultaneously, in modelling problems into code and understanding what features are available from the computing language to represent the model. Student errors stem from their inexperience with the use of program code for expressing ideas and problem solving. Most programming novice students appear to understand the need to manipulate programming code to produce solutions to problems and thus appreciate that the mutability of code allows a notation for expressing many different types of solution. Unfortunately an appreciation of how code is to be manipulated is difficult to grasp with a first programming language. The source of the majority of novice errors appear to be because they are not able to discriminate between the importance of different components of a body of code. Often students will create a non-standard syntactic notation while at the same time being greatly unwilling to manipulate the components such as the names of variables and the order of statements.

Chapter 7:

An agent framework for mentoring within cognitive apprenticeship

7.1 Introduction

In this chapter the theories, problems and technologies discussed in the previous chapters are considered in relation to each other to explain the rationale behind the development of the mentoring agent, MRCHIPS, and to determine the processing requirements for its architecture. In previous chapters an examination was made of a number of Intelligent Tutoring Systems (ITS) such as SHERLOCK (Lesgold *et al.* 1992), UNCLE (Wang & Bonk 2001), CABLE (Chen *et al.* 2006) and SIPLeS (Woolley & Jarvis 2007) that had implemented cognitive apprenticeship. An examination was also made of the capabilities of intelligent virtual agents used for tutoring in systems such as Steve (Rickel & Johnson 1998), Adele (Johnson *et al.* 1999), PPP persona (Andre 1999) and FatiMA (Aylett *et al.* 2007). While the development of agent systems as a tutor is a well researched area very little attention has been made to the role of an agent as a mentor. The role of the mentor is to act as a more experienced practitioner willing to share their knowledge, guided by the concerns of the learner in comparison to that of a tutor who provides a programme of teaching material and gives feedback on the learner's performance. Mentoring includes the activity of coaching (Landsberg 1996), which provides support during practice-based learning and is a large component of learning to program. A more detailed discussion of the role and tasks involved in mentoring was given in chapter one. They are reviewed in the sections of this chapter each followed by an *Agent capability* section used to accumulate the requirements for the MRCHIPS agent.

The cognitive apprenticeship pedagogy is used to provide the theoretical underpinning for mentoring as they share the activities of coaching and support in terms of scaffolding. The requirements for a cognitive agent based mentor can therefore be determined by examination of the pedagogy alongside the other subjects introduced in previous chapters of programming theories, the programmer's environment, the observed novice errors, the capabilities of virtual agents and architectures for intelligent reasoning. This chapter describes the mentor agent named MRCHIPS (Mentoring Resource a Cognitive Helper for Informing Programming Students), explains how it interacts with the learner and determines a set of capabilities for its operation.

7.2 Handling errors in Python

When errors occur in software the programmer is faced with two tasks to determine the location of the code that is at fault and to devise a solution to correct the fault. Locating the fault includes both identifying the position in the code and determining the component of the code that is the cause. For syntax errors and simple semantic errors (those in the order of misspelled variables or unquoted strings that would normally be detected by the compiler in languages like Java or C++), identifying the code component at fault usually identifies the required correction. For more complex errors a redesign of the code, such as initialisation of data, the order of statements or additional operations might be required. For an experienced programmer the type of the error, the content of the error message and a reading of the relevant section of code are usually all that is required to determine the cause of an error. From the analysis in chapter six it was shown that the programming errors produced in Python could be grouped into three categories and that these groups were based on how the programmer experienced the error. The categories are syntax errors, semantic errors and logical errors. The reason for the distinction between the categories was to account for the dynamic typing of Python where some decisions on the nature of the operation to be carried out on data can only be determined once the program code is being executed. The categories also reflect how the learner is encouraged to diagnose errors and attempt to correct them on their own.

Learners do progress while learning to program within the normal teaching curriculum, making fewer errors and solving more complex problems over time. The trend of the results for the observation of student errors, summarised in chapter six, figure 6.5, shows that the occurrence of errors decreased over time. As students continue to learn the main purpose of the agent is to supplement the process and provide mentoring in the form of additional diagnosis resources when errors are produced. To provide additional mentoring support this thesis proposes an intelligent animated agent to sit alongside the learner on the desktop and provide support within the framework of cognitive apprenticeship by supplementing coaching and scaffolding methods. The reasons for an agent-based solution are:

1) An agent would avoid an ITS environment where the learner would have the additional cognitive load of having to learn the interface of the additional application.
2) Working alongside the Python IDE and Windows desktop produces a dynamic environment with differing requirements, such as monitoring applications and diagnosing errors, challenges that are suited to agents' reasoning.
3) The capabilities of agents may be extended by interfacing with other code libraries and tools.

Evidence from the psychology of programmers (chapter three) indicates that programmers do not solve programming problems from first principles but rather recall experience to apply to new situations that may arise. This method of reasoning is analogous to case-based reasoning, which is embedded in the mentor agent activity in the following way:

1) To diagnose the learner's errors the mentor agent, which has a profile of the learner's previous errors can retrieve the closest solution match from similar past cases, or adapt the candidate case to the new problem and offer the solution to the learner.

2) Each of the novice programming errors can be combined with its given solution to correct the error. The various error-solution combinations form individual diagnostic situations or cases, which may be used as the knowledge base for the CBR.
3) The solution contained in each case is a plan to address the problem of the case. When the agent executes the plan it causes the agent to appear on the desktop and prompt the learner to identify a cause for the error.

7.2.1 Agent capability

a) The agent must be able to monitor and detect the placement and content of desktop windows in general and those related to the Python development environment specifically to help determine learner's activity.

b) *The agent must be able to monitor the Python development environment and sample the learner's code to determine the context and cause of errors.*

7.3 The Cognitive Apprenticeship approach to learning to program

Cognitive Apprenticeship pedagogy, which describes a structure for teaching practice-based subjects such as law and medicine, underpins the approach adopted by the mentor agent. It consists of six methods that describe the activities to be undertaken by teachers and learners, as was discussed in chapter three and summarised in table 7.1 below. From the literature it was found that researchers have used cognitive apprenticeship as a teaching framework for their ITS and a number of the systems were also reviewed. Each of the ITS systems used different approaches and selected those methods in their implementation of the cognitive apprenticeship to suit their teaching requirements. In this research the main methods of cognitive apprenticeship that are considered for the agent are those concerned with mentoring activities of coaching, scaffolding and exploration. The assumption is made that other parts of the pedagogy are available as part of the normal course of a university based programming module where the agent supplements the teaching.

Method	Agent activity
Modelling	The teacher as the subject expert would carry out this activity in the form of lectures, workshop exercises and assessed pieces of work.
Coaching	This stage is one of the major tasks of the automated agent; it would provide support to the learner based on its database of prior similar cases.
Scaffolding	The agent will implement scaffolding by tuning the level of feedback to the learner and fading the level or amount of support as the learner becomes more proficient
Articulation	No explicit support is being provided for this method of the pedagogy.
Reflection	The agent will eventually be able to provide a summary of the users performance
Exploration	This should be available to the learner by virtue of the agent learning support working within the standard development environment.

Table 7.1 The methods in the cognitive apprenticeship pedagogy mapped against the agent activity

The only constraint that cognitive apprenticeship makes on the identity of the coach is that they have expertise in the subject. In the normal course of events a teacher would provide coaching, however if a teacher were not available the expertise could be encoded in a knowledge-based agent system. For a knowledge-based system to be able to provide teaching assistance to a learner it has to fulfil a number of requirements. The agent has to be responsive to the user, it needs to monitor and react to changes in the environment, it has to be able to reason about problems in the subject area, communicate the results of its reasoning and monitor the outcomes. Some of the steps are available to automation in the programming field by intelligent software. The mapping of each step of the pedagogy is shown in table 7.1 alongside the behaviour of the intelligent system.

The primary methods of cognitive apprenticeship that are addressed by the mentor agent are coaching and scaffolding, where the agent supports the learner in practical exercise sessions when the learner attempts to reproduce the activities of the expert. This supports the initial description of the agent as a mentor as opposed to a tutor; the agent does not introduce course material to the learner rather it provides a mechanism to help assimilate new knowledge. The major activity of the agent in mentoring will be in support of the coaching method. At this stage the agent system sits alongside the learner's development environment to monitor activity as they write code and alert or advise them of errors and problems in a format suited to the requirements of the novice. The mentor agent has to monitor the user's activity, analyse the nature of a user's problem and provide effective responses. Other methods are addressed as part of the normal teaching curriculum. A human teacher following a course of lectures, demonstrations and set exercises still provides the modelling method. The scaffolding method is shared between the teacher and the agent following the requirements of the curriculum in setting the level of the tasks and support provided and the mentor in selecting the level of support in responses within individual exercises.

Although the cognitive apprenticeship methodology does not have an explicit mentoring method it does include a coaching method. The assumption made for this research is that there is no significant difference between mentoring and coaching. The term mentor was chosen for this research to emphasise a passive role for the agent's assistance and make a clear distinction from a tutor. Cognitive apprenticeship defines coaching as the learner repeating the task observed by the expert who provides hints, tips and reminders to aid them, while the dictionary defines mentoring as advice from a wise guide or counsellor. However both words are synonymous and it is arguable that the perceived difference between the two words is primarily a matter of context. Mentoring students involves explaining the code component at fault and providing a solution for novice learners. As the students become more proficient the level of help required reduces to the level required by experienced programmers in locating and rectifying the error. Scaffolding is provided as a result of the activity of the learner, as the learner progresses they reduce the number of errors made or become more able to correct them before seeking help.

As mentoring is largely a coaching process the main activity is to allow the learner to reproduce the expert’s activity under observation, to provide hints, tips and reminders. In terms of the agent’s behaviours this means:

- Reproducing the expert’s activity means allowing the learner to practice writing code in the environment;
- Under observation means the agent needs to monitor and assess the learner’s work; and
- Provide hints, tips and reminders means under certain circumstances providing appropriate feedback to the learner.

The result of this is that the learner programs Python and interacts with the normal development environment for most of the time while the agent remains out of the way, but the agent observes the learner’s activity and is activated (that is to say becomes interactive) to provide hints, tips and reminders in specific circumstances.

Two situations were chosen for agent activation:

- An error occurs and the learner has been unable to correct it after a set period of time;
- The learner makes a request for the agent to become active.

A third situation where the agent would give a positive message after a successful run was considered but not pursued. The idea was that the message would encourage further reflection, but there was no evidence for this. It might be possible that the agent’s approval would signal the end of development to the learner, where there may still be logical errors to uncover and correct.

7.3.1 Agent capability

c) The agent strategy will be to respond to errors found by the development environment rather than lead the process. This will avoid the agent presenting information to the learner that they might not be ready to receive and to allow the learning activity to remain driven by and centred on the learner.

d) *When the Python interpreter finds a syntax error an additional parser in the agent will compare the learner's code against the BNF for the language to identify the location in a statement that is the element of code at fault before being used to index case retrieval in the CBR. The need for the additional parse is because each syntax error message covers multiple types of coding error.*

e) *The agent will use runtime error messages directly from the Python interpreter to index case retrieval to determine the cause of semantic errors, as the range of error messages compared to its cause is small.*

f) *Logical errors will be addressed by a natural language consultation where the learner can pose questions to the agent and causes or solutions suggested.*

7.4 The agent interface

One possible solution to automating coaching support would be to implement a specialised ITS environment on which the novice programmer can practice and be provided with a more detailed breakdown of mistakes and errors. While this approach may provide an environment where teaching material could be presented in a systematic and highly controlled way the limitation would be to break the principle of cognitive apprenticeship for the learner to use the real world tools of expert practitioners. Another approach might be to provide a programming environment that would be the same as that used by practitioners, where areas of the environment and the language code could be shielded from the novice as they begin and with the restrictions fading as the learner progresses. This would be a more attractive approach than a specialised ITS application and would allow the learner to work in an environment closer to a real world context. While the restrictions would offer a scaffold to the learner the limitation of this approach would be in how to accommodate the cognitive apprenticeship method of exploration. The restrictions would provide the learner with fewer opportunities to become familiar with aspects outside of the sequence of the fading scaffold. A solution that allows a closer adherence to cognitive apprenticeship would be to allow the learner to train in the real world environment of the programming language and have knowledge-based software mentor the learner by providing expertise within the same environment. The knowledge-based tool designed for work in a given environment is the intelligent agent and its behaviour would be to fulfil the main functions of coaching support. Although an agent approach provides a solution that should integrate within the cognitive apprenticeship pedagogy with little alteration to the environment or method of learning the greatest challenge is the unstructured nature of the environment and interaction with the novice.

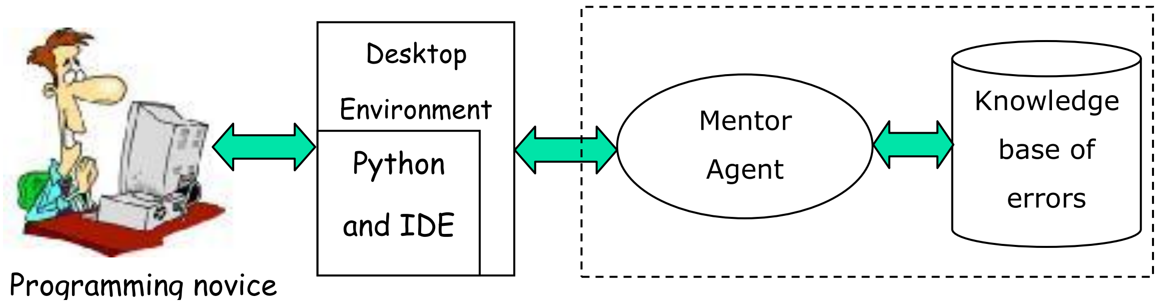


Figure 7.1. Sketch of the novice and mentor agent interaction

One of the main principles arising from the cognitive apprenticeship pedagogy for this research is for the learner to work on real-world problems in as near as possible to the environment as used by expert practitioners. The theory does not directly speak of cognitive load for the learner but addresses it in the principle of scaffolding and the fading of support as the learner becomes more competent. The Python programming language provides a real-world software development tool. It also supports the reduction in cognitive load by way of a small language core, a simple syntax, optional inclusion of module and object libraries and other support tools such as colour syntax highlighting editors and code profiling tools. To develop an agent with a solely conventional application interface (i.e. buttons, text fields, icons, etc) would provide yet another tool to learn that would add to the cognitive load. Research form intelligent virtual agents, as explained in chapter four, indicates that animated virtual characters allow users to communicate in modes that are a closer analogue of the real-world, such as natural language, speech, embodiment and gesture. In this way the rules of interaction are already largely known by the user and the requirement to learn an additional application can be avoided. An anthropomorphic interface, where the agent maintains strategies for communication and dialogue with the learner, provides an easy interface to the agent's knowledge by avoiding the requirement for the learner to learn how to use an additional interface and so would not add significantly to the cognitive load of the learner.

7.5 The agent environment

To fully understand the capabilities of any agent system consideration needs to be made of the environment in which it operates. The design of MRCHIPS was influenced by the opportunities and constraints imposed by the nature of the environment. Opportunities include factors such as the message passing nature of a computer desktop environment, whereas constraints are features such as the set of development tools. The agent's environment is the Windows desktop of any PC variant of the Microsoft Win32 operating system such as Windows ME/2000/XP/Vista; although Windows 7 has limited support for the Microsoft agents engine, it is used for the animated character interface and explained in chapter eight. It is no longer shipped with the operating system and needs to be obtained from the Microsoft website.

In terms of Russell and Norvig's five properties used to characterise agent environments, as explained in chapter five, the Windows operating system imposes the following constraints on the capabilities of the agent. These are:

- *Accessibility vs. inaccessibility:* the privacy and security issues of the Windows environment means it is not accessible;
- *Deterministic vs. nondeterministic:* as Windows is a multitasking operating system it is nondeterministic;
- *Episodic vs. nonepisodic:* as the accessibility is limited on Windows the agent episodes have to be retained across perceptions;
- *Static vs. dynamic:* the Windows environment is dynamic as it changes constantly outside of the control of the agent;

- *Discrete vs. continuous:* as Windows may support an unlimited number of configurations the environment is continuous.

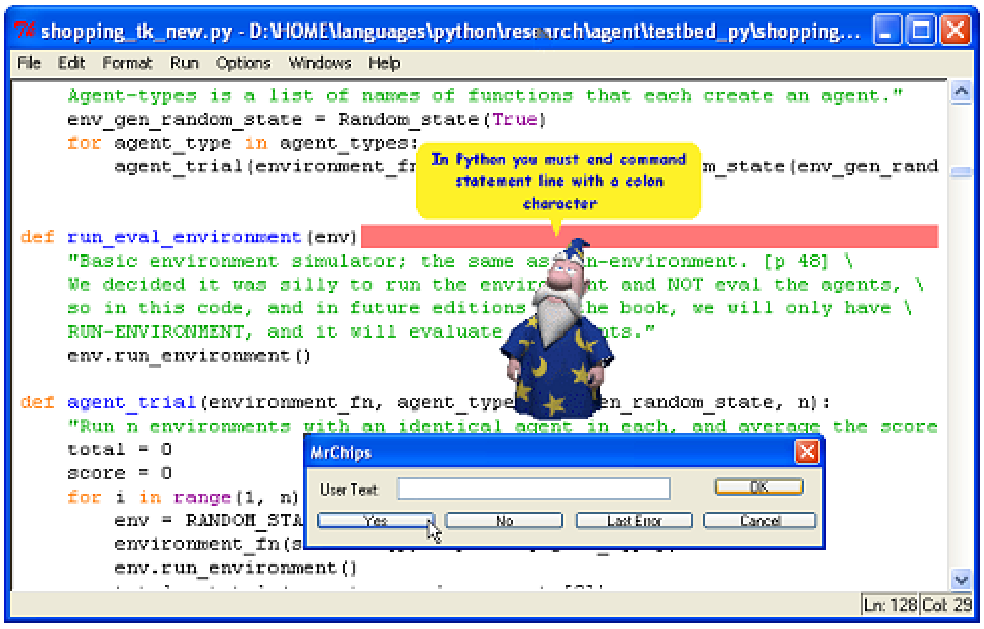
These properties inform the eventual capability of the agent and constrain the overall design of its architecture.

7.5.1 Agent capability

g) The agent must operate on a Desktop environment, notably Win32, as this is the platform used to teach programming in the university.

7.6 A mentoring scenario

To illustrate the use of an agent-mentoring assistant, consider a student, called Oscar, working on a desktop environment to develop a Python program. Alongside him but not visible is the agent, the Mentoring Resource a Cognitive Helper for Informing Programming Students (MRCHPS). Oscar has been asked to make use of an insert swap program that is able to sort a list of numbers and then to sort a list of names, of the seven dwarves, "Sneezy", "Dopey", "Grumpy", "Sleepy", "Happy", "Bashful" and "Doc", into alphabetical order. He has successfully used the program to sort ten numbers and works out that he must put the names in place of the numbers. Oscar edits his program and enters the names, however when he attempts to run the program it reports an error of type 'name error' for an unidentified variable on the line specifying the names of the dwarves. The error alerts the MRCHIPS agent, which has been monitoring the learner's activity in the desktop environment. The agent then reads the code from the Python development environment along with other values that are used to help select the closest matching cases from its knowledge base. It is possible that the learner is able to correct the error by him or her self so MRCHIPS places a small transparent window on the desktop informing Oscar that help will be provided in thirty seconds to see if he is able to correct the error. MRCHIPS notes that Oscar's activity makes no change to the faulty code, as a result the MRCHIPS interface, the Merlin character from Microsoft agents, becomes visible on the desktop (Figure 7.2), with an introductory message offering help.


shopping_tk_new.py - D:\HOME\languages\python\research\agent\testbed_py\shopping...
File Edit Format Run Options Windows Help
In Python you must end command statement line with a colon character
MrChips
User Text
OK
Yes
No
Last Error
Cancel
Ln: 128 Col: 29

```
    Agent-types is a list of names of functions that each create an agent."
    env_gen_random_state = Random_state(True)
    for agent_type in agent_types:
        agent_trial(environment_fr                          m_state(env_gen_rand

def run_eval_environment(env)
    "Basic environment simulator; the same as  n-environment. [p 48] \
    We decided it was silly to run the envir   nt and NOT eval the agents, \
    so in this code, and in future editions  he book, we will only have \
    RUN-ENVIRONMENT, and it will evaluate       nts."
    env.run_environment()

def agent_trial(environment_fn, agent_type       en_random_state, n):
    "Run n environments with an identical agent in each, and average the score
    total = 0
    score = 0
    for i in range(1, n)
        env = RANDOM_STA
        environment_fn(s
        env.run_environment()
```

Figure 7.2. The mentor agent's advice to a learner

The agent character, and an input dialog box are placed alongside the Python code Window, and offers to help. As Oscar is unsure why the error has occurred he accepts the help and MRCHIPS gestures toward the code while providing the explanation from the selected case that the variable name is likely to be data and should be surrounded by quotes to prevent evaluation. MRCHIPS continues to monitor Oscar as he makes the correction. Once done the new case is recorded. Working in this way MRCHIPS provides mentoring support by undertaking the behaviours of coaching in providing immediate feedback to the learner, that is context/task sensitive, and the guidance offered is to support performance improvements (Laffey *et al*. 1998).

7.6.1 Agent capability

h) The Agent output to the learner will be directed via an animated, anthropomorphic character as produced by the MS Agents interface or similar system. Use will be made of the text to speech for voice generation if available.

i) *The agent will make a delay between detecting an error and providing assistance to allow the learner a chance to solve the problem on their own and therefore to encourage learning.*

j) *Input to the agent will be via text input, with a simple natural language parser to interpret inputs. While speech input may be possible it will not be considered a requirement as the natural mode for programming input is already via the keyboard.*

7.7 A Cognitive Apprenticeship agent framework

The scenario above illustrates that the development goal for the mentoring agent is to produce an agent architecture with the range of behaviours and the available knowledge to provide mentoring support to novice programmers. Rather than producing a single or pure reasoning technique the architecture is a collection of reasoning techniques that integrate to produce a cognitive architecture based on the definition asserted by Langley (2006). Whether the design is based around a multiple-agent systems framework, a blackboard system, or a cognitive architecture, all of the systems maintain some form of reliance on short-term and long-term memories, the representation and organisation of structures within these memories, the reasoning that is able to operate on the structures and a programming language that allows the construction of knowledge-bases (Langley 2006).

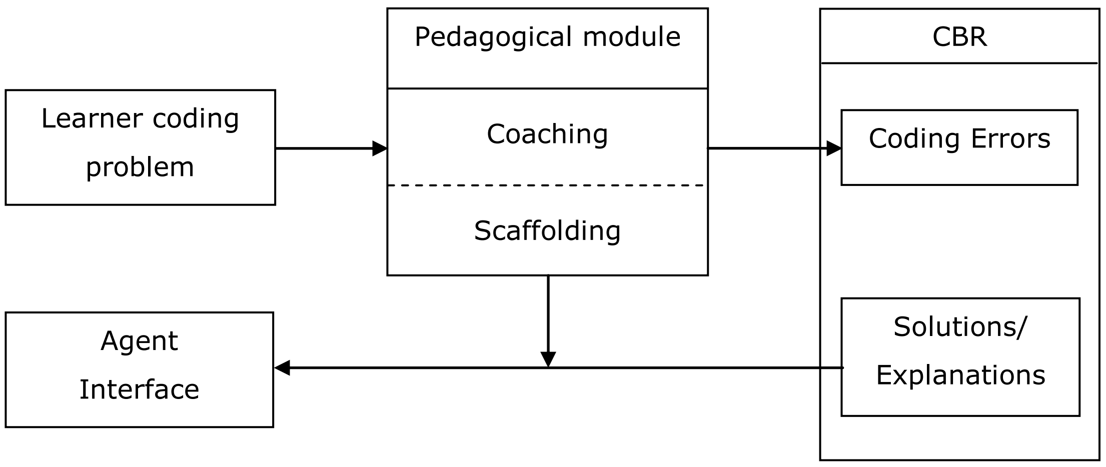


Figure 7.3. The agent framework

The architecture chosen consists of a number of processing subsystems that coordinate the activities of the agent. The two main reasoning modules are the Beliefs-Desires-Intentions (BDI) and the Case-Based Reasoning (CBR) subsystems. The two subsystems coordinate the different levels of reasoning required to provide the different capabilities of the agent, as in figure 7.3. The BDI subsystem provides the processing required to interface the agent with the environment. It coordinates the control of the agent interface, the speech, emotional expression, gestures, its position and orientation. The reactive and deliberative capabilities of the BDI allow the agent to sense various computer resources such as the filing system, the Windows desktop, the content of Windows of interest, keyboard activity and mouse clicks. By tracking the user's activity this layer will also be able to make inferences about user activities and select suitable responses for the agent character. The CBR subsystem maintains specific domain knowledge about programming errors, techniques for diagnosing errors and the strategies for communicating solutions to the learner. Although both the CBR and BDI subsystems use different internal representations suited to their individual processing requirements, they are both capable of sharing each other's knowledge-bases by including mechanisms to access the different knowledge-bases from within the plans of the BDI and the cases of the CBR. Other research has also proposed the combining of BDI-CBR agent systems for intelligent web searching and a tourist guide agent (Corchado and Pellicer 2005). These systems have primarily been concerned with adding learning capabilities to BDI and have in different ways used CBR to implement BDI agents. The MRCHIPS agent works differently from these systems in that the BDI and CBR subsystems are structured to reason in different ways on different aspects of a problem domain and combine their results to provide solutions where a single reasoning method would be insufficient. In this way MRCHIPS is similar to cognitive architectures such as Homer (Vere and Bickmore 1991), Prodigy/Analogy (Veloso 1994), Fatima (Aylett *et al*. 2007) and the challenge set for generality in intelligent systems by Pat Langley (Langley 2006).

7.7.1 Agent capability

k) *The agent will be implemented as a processing shell. The domain knowledge, behaviours, plans, rules and cases will exist as a knowledge base script that will be interpreted by the agent.*

l) *The rules and plans of the BDI engine will coordinate the interaction with the learner and processing of other modules within the agent.*

m) *Domain Knowledge concerning learner errors and corrective action required naturally form cases and will be processed by the CBR engine.*

n) *The knowledge base content will be expressed as a sequence of predicate calculus clauses. A Prolog-like parser will read the content and individual modules will extract data into an internal representation as required by each module.*

7.8 Agent requirements for MRCHIPS

The MRCHIPS architecture can be characterised as an implementation of a cognitive architecture produced by an integration of a plan-based agent system and a case-based reasoning system to address a problem that would be difficult to solve using a single reasoning system. The architecture satisfies the commitments of a cognitive architecture explained in the literature review. The agent is implemented as a number of integrated interpreter subsystems that can be programmed via various knowledge sources to produce the required behaviours. Through access via various Win32 programming resources the agent is able to share a desktop programming environment with a programming novice to allow learning to continue in a real-world context. MRCHIPS is able to detect and analyse Python syntax errors and semantic errors and to provide help via an animated assistant when programming errors are produced. At the present time the agent still lacks the component to provide support for logical errors, but the plan is for a natural language parser to drive a question-answer subsystem to access the cases concerned with logic errors. However with the MRCHIPS agent able to address issues of syntax and semantic programming errors it has enough functionality for experimentation with programming novices and testing of the hypotheses.

7.9 Summary

In this chapter the idea for a mentor agent to assist novice programmers was analysed in terms of the problems of learning to program, the cognitive apprenticeship pedagogy and agent theory. The literature in chapter three was concerned with the psychology of learning to program and identified the causes of these errors as neglected strategies and fragile knowledge. In chapter four it was shown that cognitive apprenticeship was an effective pedagogy for teaching subjects where the learners had to understand how to apply the acquired knowledge. The analysis of novice errors, from chapter six, confirmed the observation that learners make programming mistakes in a variety of ways from a set of misunderstood concepts. The results of the analysis from the students' errors are collated with the literature from cognitive apprenticeship pedagogy to determine the type of reasoning required to provide mentoring support to novice programmers. So the decision was made to concentrate the agent's assistance on the coaching and scaffolding methods of the cognitive apprenticeship pedagogy and the agent behaviour on diagnosis of syntactic and semantic errors. It was also decided to implement the agent behaviour in a BDI planner as this provides a mechanism for handling the differing requirements from a learner in the desktop environment and a CBR for the diagnosis of learner errors as the combination of a programming error and its solution correlates to a diagnostic case. The Microsoft agent engine will be used to produce the animated character for the virtual agent as this provides most of the facilities for an anthropomorphic interface.

Chapter 8:

Implementation of MRCHIPS

8.1 Introduction

This chapter describes in detail the design and implementation of a cognitive diagnostic agent that is able to provide mentoring support to novice programmers and the interface to its environment. The information from the analysis of student errors, which was examined in chapter six, has led to the development of an agent given the name MRCHIPS (Mentoring Resource a Cognitive Helper for Informing Programming Students). In section 8.2 a description is given of the agent's operation as it interacts with and helps diagnose student errors, illustrating its reasoning and mentoring capabilities. In section 8.3 the constraints of the programming environment and windowing desktop are highlighted. In section 8.4 a description of an overall design of a mentoring agent is provided, followed by a more detailed view of its architecture, subsystems and components. Finally section 8.5 explains how the prototype MRCHIPS agent is implemented.

8.2 The MRCHIPS cognitive architecture

The agent, MRCHIPS, is an implementation of a cognitive agent architecture, as discussed in chapter three, which combines a plan-based agent system with a case-based reasoning system to address the complexity of the learners' problems and the reasoning required for mentoring purposes. To satisfy the methods of the cognitive apprenticeship model the system's architecture included the following components:

a) The long-term and short-term memories are addressed by the case memory and belief-base.
b) The memory based knowledge structures are represented in the form of Prolog clauses.

c) The processing is distributed across the Python Agent Language (PAL) engine, the Case-Based Reasoner (CBR), the agent-user interface and the Backus-Naur Form (BNF) parser.
d) The PAL language and Prolog are used to configure the knowledge bases in the agent.

The agent is implemented as a number of integrated processing subsystems that can be programmed via various knowledge sources to produce the required reasoning. By accessing various Win32 programming resources the agent is able to share a desktop programming environment with a programming novice to allow learning to continue in a real-world context. MRCHIPS is able to detect and analyse Python syntax errors and semantic errors and to provide help via an animated assistant when programming errors are produced. An overview of MRCHIPS showing its logical interface to the environment is shown in figure 8.1.

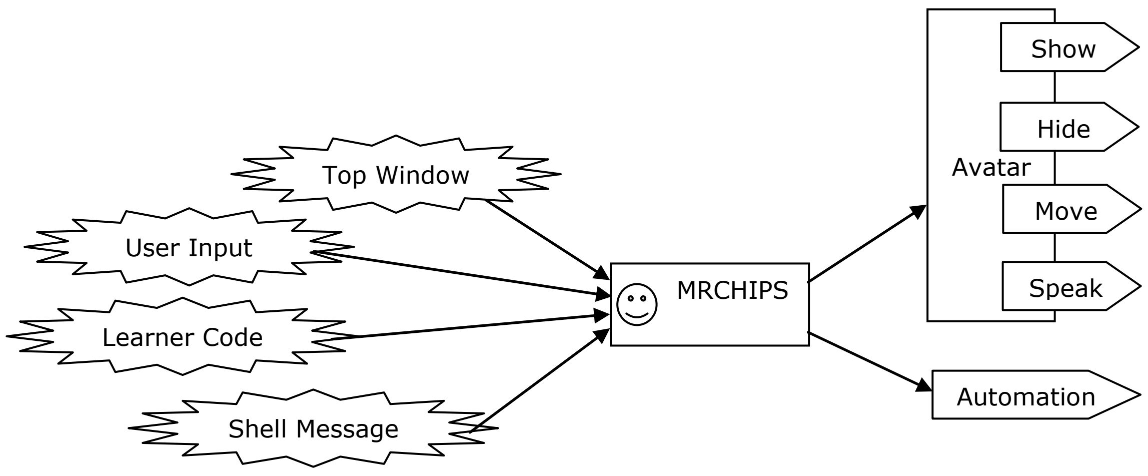


Figure 8.1 MRCHIPS System overview diagram

8.3 The subsystems of MRCHIPS

The MRCHIPS architecture consists of four major subsystems: the BDI, the CBR, the BNF parser and the agent interface (see figure 8.2) that coordinate the various behaviours of the agent and provides the domain specific reasoning. At the heart of MRCHIPS is the Python Agent Language (PAL). This is an implementation of a plan interpreter that combines reactive and deliberative reasoning. MRCHIPS has to be able to monitor and respond to changes on the desktop as well as performing diagnostic activities and maintaining interactions with the learner. It is the presence of PAL that co-ordinates all these activities within the agent.

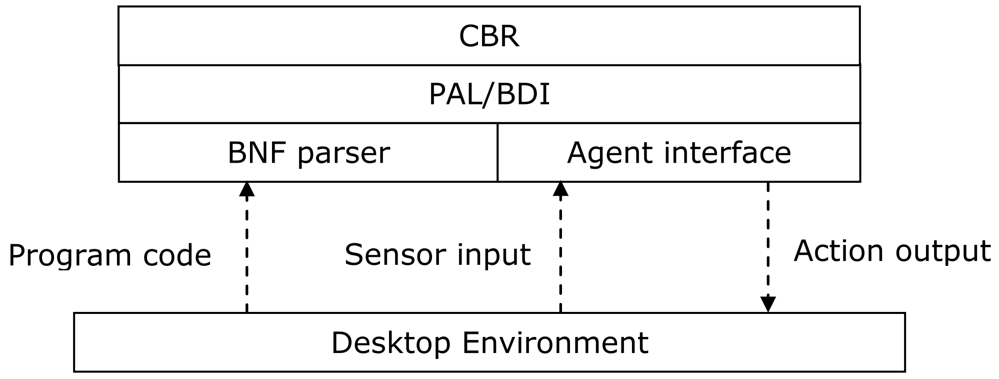


Figure 8.2. The mentor agent's architecture

An early design for the agent considered a design based on the Interrap architecture as discussed in chapter five. The design involved insertion of the case-based reasoning layer between the deliberative planning and communications modules to produce a four-layered architecture. However this design was not pursued as it was thought the real-time requirements for a desktop environment did not require separate reactive and deliberative processing layers. The capability of the BDI to combine deliberative and reactive processes was considered to be adequate for the requirements of a desktop environment.

8.3.1 Reasoning in MRCHIPS

The main cognitive processes in MRCHIPS are shared between the BDI planner, the CBR subsystem and the Python syntax pre-processor. There is also a very limited natural language parser to allow interrogation of the agent's knowledge base as part of an interactive reasoning facility for analysis of logical errors, but this was not fully developed and the agent can only respond to a limited set of queries. To aid the easy integration of knowledge between the systems all of the knowledge bases, plans, cases and rules are encoded as Prolog predicate calculus clauses. However once parsed they are processed in different ways by the relevant parts of the agent.

The Prolog database, normally used to store facts and rules in a Prolog program, is used as the beliefs knowledge base of the agent. No distinction is made to distinguish agent beliefs from Prolog clauses. This strategy allows MRCHIPS to access beliefs that are implemented as demon processes, for instance to retrieve the current day of the week. The behavioural capabilities of MRCHIPS are to monitor the desktop and Python shell, profile the user's program code, select the closest matching case and inform the learner of any solution. In general plans are domain independent and used to encode the agent behaviours and capabilities, although some of the capabilities are domain related in terms of the way the Python development environment works. The knowledge used for the cases and syntax rules is domain specific and although not unique to Python is directly related to issues arising from learning Python.

8.3.2 The BDI reasoning subsystem

The PAL interpreter at the core of MRCHIPS is based on the BDI (beliefs-desires-intentions) family of agent architecture, which is a customised system largely based on the AgentSpeak(L) agent architecture and incorporating language features from other agent systems such as 3APL and JAM (Ancona *et al.* 2005).

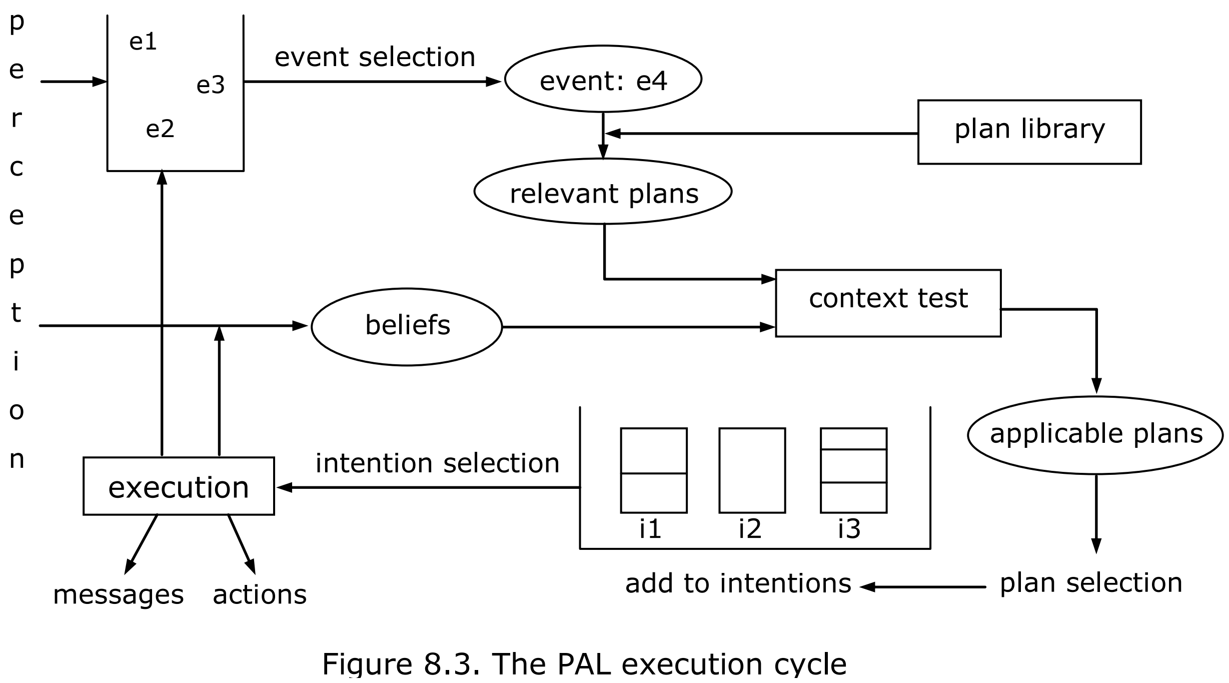


Figure 8.3. The PAL execution cycle

A similar evolution of the AgentSpeak(L) architecture was implemented by Flake and Geiger in their CASA agent system (Flake & Geiger, 2000) and used in the simulation of character interaction. The execution cycle for PAL is shown in figure 8.3 it is a modified version of the execution cycle given for the CASA agent system. The strategy and decision-making of MRCHIPS is coordinated from its plan-base. Plans are a specification of a list of actions to be performed in response to events and to fulfil an intention. Plans consist of three components: the trigger event, a guard condition that specifies the applicable context of the plan and finally the body of the plan is the set of actions to be performed.

```
plan:
        event   : awakeAgent,
        context : true,
        body    : {
                  write('== maximise Agent'), nl,
                  avatar:isVisible(N),
                  if N \== true then {avatar:show()}
                  }.
plan:
        event   : checkHelp(RtErr),
        context : [agentMode(idle)],
        body    : {
                  eval(currentFile(File)),
                  if profileCode(File,RtErr) then
                          {wait(offerHelp)}
                  else
                          {wait(pause(0.5)),
                           wait(getDeskTopApps)}
                  }.
```

Figure 8.4. Two example plans in MRCHIPS

The structure of two BDI plans are illustrated in figure 8.4. Events can be a single symbol or a clause. If the clause contains variables they can be used to pass values in a similar manner to a function call. Plans in PAL support Prolog-like variables. They are indicated by symbols beginning with an uppercase letter and can be instantiated by unification. The context guard condition is optional if it is not required, meaning the plan is universally applicable and then the context can be set to true. Otherwise the context is a list of clauses that must be available in the belief base or evaluate as true. Using context conditions within each plan allows varied modes to be specified, either by a flag value or testing some value in the belief knowledge base. The effect of this causes some collections of plans to govern particular behaviours, while other collections remain inactive. The body of the plan is a list of statements that are evaluated by the PAL interpreter or passed to the Prolog interpreter, which executes primitive operations. The syntax for the body of PAL plans is shown in table 8.1 in BNF notation (for clarity keywords, operators and functions have been highlighted in bold).

Block	'**{**' block-content '**}**'
Block-content	statement ['**,**' block-content]
statement	if-statement **|** while-statement **|** assignment **|** belief-modifier **|** plan-call **|** primitive-statement
if-statement	**if** condition **then** block [**else** block]
While-statement	**while** condition **do** block
assignment	Variable **is** expression **|** variable '**=**' expression
belief-modifier	**assert**(expression) **|** **retract**(expression) **|** **eval**(expression)
Plan-call	**wait**(expression) **|** **achieve**(expression)
Primitive-statement	atom **|** primitive-function
Condition	*atomic formula*
Expression	Variable **|** primitive-function
primitive-function	*prolog clause* **|** *external function*

Table 8.1. BNF syntax for body of PAL plans

Braces are used to delimit blocks of procedural code. The PAL interpreter supports the while loop and the if-then statement and plans may be chained together by lists of trigger event of other sub-plans that are called in sequence. If the event is within a wait function the calling plan is suspended until the sub-plan completes. External primitive functions are indicated by a colon separated function name, where the left of the colon identifies a subsystem of the agent and to the right the function in that system. From the point of view of programming the agent there are five subsystems that are addressed directly through function calls. They are shown in the table in table 8.2 below.

bdi	Exposed functions PAL interpreter
cbr	The case-based reasoning subsystem
avatar	The MS-agent and general outputs
sensor	The Perception subsystem
epi	The journaling subsystem

Table 8.2. Table of addressable agent subsystems

8.3.3 The case-based reasoning subsystem

Cases in the agent are implemented in a frame-like data structure where each frame represents a complete case. It was decided to represent one case per frame for simplicity rather than a distributed structure. Each case is implemented as a Prolog clause with two fields. The first field holds a unique name for the case, the second holds a nested hierarchy of attribute-value pairs that define symptoms and a solution for the case (see figure 8.5). Each case situation is described by four attributes that contain lists of clauses; they are the initial description, the solution, the final state and the success of the case.

```
case( upperCaseKeyword, [
        initial - [errortype(compiletime),
                   start(P),
                   keywordCase(P)],
        solution - [Opt = ["The " + P + " should be all lower case",
                     "Python is case sensitive so " + P + " should be lower case",
                     "Check the case of your " + P + " keyword"],
                   bdi:selectOne(Opt,I2),
                   avatar:speak(I2),
                   bdi:rememberCase(upperCaseKeyword,I2)],
        final    - [checkResult([lower(P),_])],
        success  - true]).
```

Figure 8.5. Example of a case in MRCHIPS

The initial attribute contains clauses that identify the nature of the error, for most cases the information about the error messages produced by the Python development environment. This strategy allows MRCHIPS to be guided by the context of what the learner is working on and avoids the risk of providing information on unrelated problems that might only have the effect of confusing the learner. The solution attribute lists the set of actions the agent is to carry out to achieve the state of the final attribute. The success attribute indicates the desirability of the outcome. Both solution and final attributes are not currently used by MRCHIPS but included for future expansion. The agent's case-base contains records of typical novice level errors based on information gathered from observations from cohorts of learners explained in the literature and analysed in chapter six. Although the errors observed were as a result of different types of coding problems the ultimate action of the agent is the same in each case, to provide additional information – the difference occurs in meaning of the information provided.

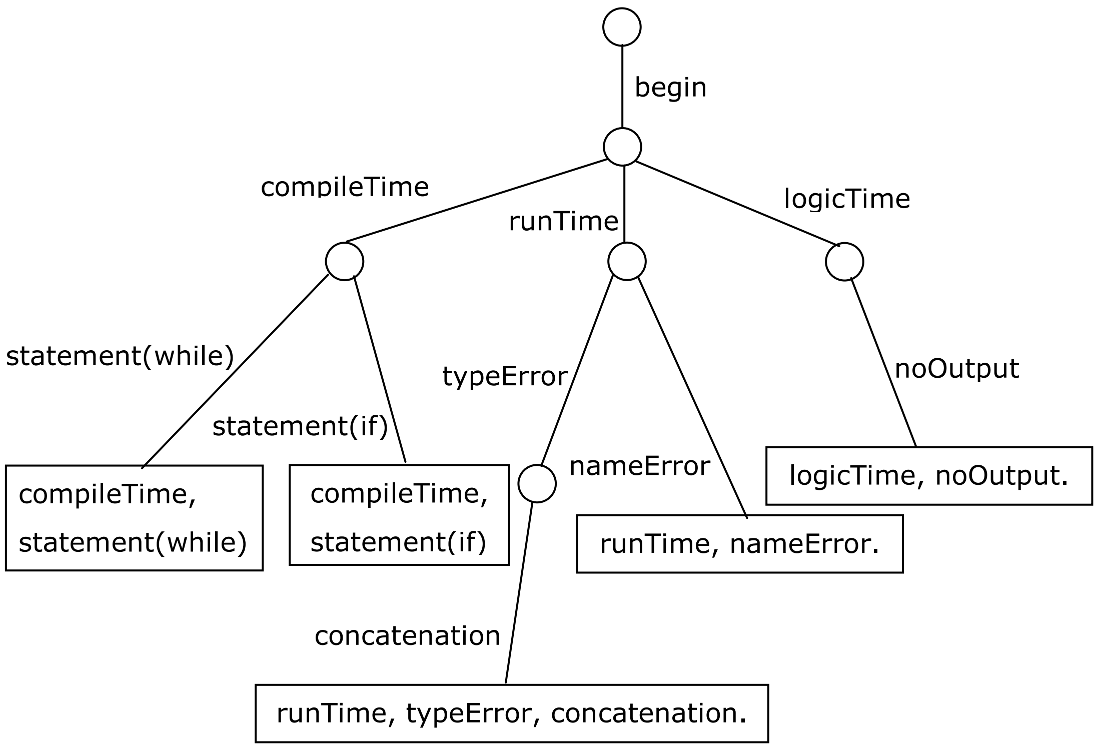


Figure 8.6. Fragment of a discrimination tree as a case-base index

The cases in MRCHIPS are indexed and stored using a discrimination tree (Charniak *et al.* 1987), also called a discrimination network, which provides an efficient method to access the case-base. A discrimination tree is a branching network data structure used for storing and retrieving large numbers of symbolic objects. The principle behind a discrimination tree is to recursively partition a set of objects where each partition divides the set based on a particular property and properties that are similar by some measure are shared in memory. The effect of placing data in the network is to cluster together items that are similar. As a side effect of the clustering a discrimination network is also able to discriminate between cases. For instance as illustrated in figure 8.6, MRCHIPS cases are initially partitioned based on the class of error, so some cases belong to the set of compile-time errors, others to the run-time errors and others to a third set of the logical-errors. Each internal node is a question that subdivides the items of data that are stored below, where each item is a different answer to the question. Case retrieval is performed by using the features of the problem case as a map into the discrimination tree to similar cases and a complete case is stored at the terminal node of each branch of the tree. The algorithm for searching a discrimination network is based on a simple loop shown below. The main work of the search is contained in the strategy for matching nodes.

```
Let N = top node of tree
Repeat until N is a case:
        Ask question at N of the input
        Let N = subnode with the answer that best matches the input
Return N
```

Figure 8.7. The search algorithm for a discrimination network

Incomplete data, indicated by a non-ground expression returns all of the sub-cases of a branch. If a variable is encountered in the problem case during retrieval it is matched against the corresponding field in the discrimination tree and all of the branches of the tree below that may have a valid value for the variable remain in the search. Ground value, occurring later in the problem case can be used to discriminate the branches at a later iteration of the retrieval. If the variable is in the tree it can be matched against any corresponding fields in the case and the search continued. A measure is kept of the degree of match for each clause selected from the discrimination tree and the solutions returned in a sorted list if more than one exists. The degree of match is given by the expression:

$$\mathbf{D = 3 * NE + NV}$$

Where:

D	= degree of match
NE	= number of symbols that match exactly
NV	= number of items matched by a variable

Figure 8.8. Expression for the degree of match in the CBR

The case-base contains a default case, called defaultError, that has a single variable value for its index pattern and therefore gives the value 1, the lowest degree of match permitted. This means that the case will always be selected but with the lowest possible priority compared to other cases. If as the result of a search there are no appropriate cases for a particular problem the default case is selected and reports a general warning message. The initial information gathered from the exercises observed from novice programmers revealed some thirty types of programming error, but no claim is being made for a complete coverage of all types of novice errors. The indications are for the number of cases to be in the order of many tens (possibly hundreds), rather than thousands and even if the case-base were to grow the discrimination tree based search would still work with these numbers. A more detailed treatment of discrimination networks, their use in deductive information retrieval systems and implementation can be found in Charniak *et al.* (1987).

Although the domain knowledge is described as case-based reasoning MRCHIPS does not fully implement CBR. The cases in the agent describe stereotypical rather than particular error situations. They may contain variable fields rather than fully grounded clauses and although the programming structures are present to revise and retain newer cases they have not been developed in this agent implementation. But it is to be considered as a recommendation for the future. The case structures might more accurately be described as contextual schema, such as developed for the MEDIC and Orca knowledge based systems (Turner 1994). Schema-based reasoning is a generalisation of case-based reasoning that extends cases to generalized situations by allowing cases to contain variable fields and saving the effort needed to transfer knowledge from an old case to a new situation. The variable field allows for more approximate matching and can exist in the problem case and in the case-base. Case selection is performed by tracing the content of the initial attribute only against the discrimination tree for the best matching case. This is because the initial field contains the symptoms of a problem and it is that data that is used to identify similar cases. Once a case has been selected a copy is taken and it is adapted to the new problem. This is achieved by unifying the problem case with the new case and instantiating variables to produce a fully grounded data structure, the new case is then asserted into the agent's beliefs knowledge base, where it becomes available for further processing.

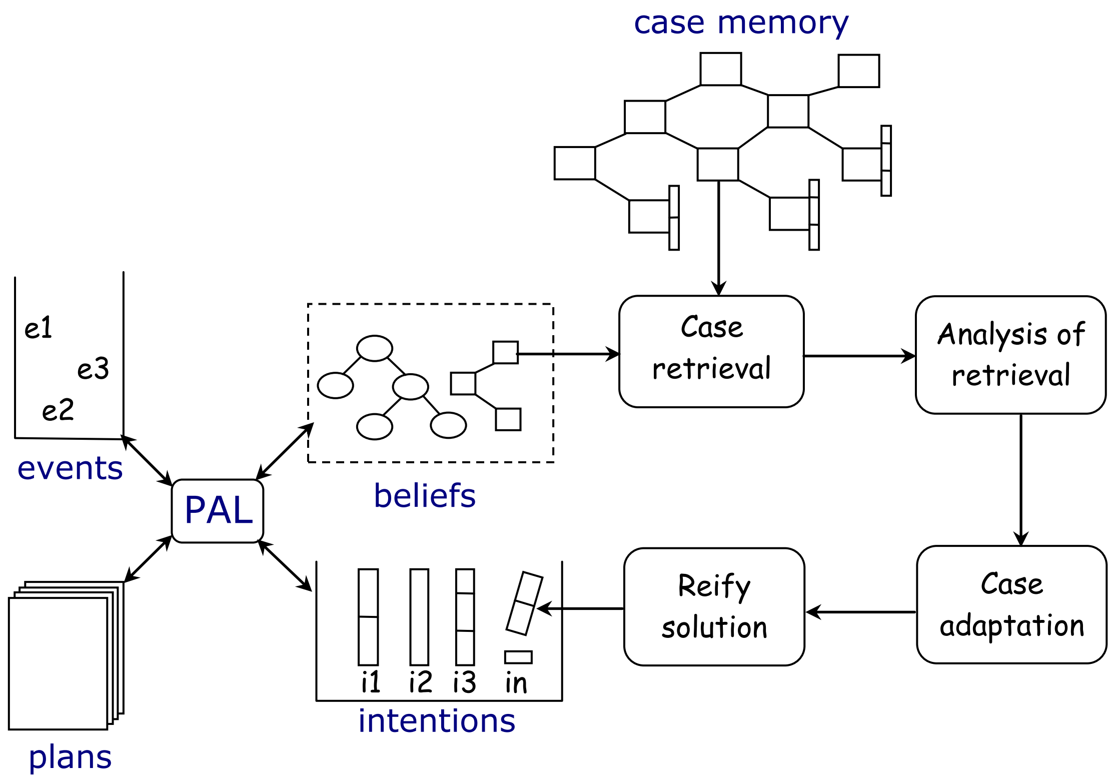


Figure 8.9. The mentor agent's architecture

The reasoning in MRCHIPS is shared between the BDI and CBR subsystems and linked associating BDI beliefs and intentions with case symptoms and solutions respectively. See figure 8.8. To link the subsystems the agent is able to use some of the information in the beliefs knowledge base as symptoms for case selection. The symptoms are constructed under the control of the plans as this assists the indexing process, which can be suppressed if additional knowledge is available or the format adjusted if the requirements change. When the most appropriate case is selected it is activated for use within the agent by placing the solution, which is merely a plan to address the symptoms, into the BDI system's intentions stack for execution. The activation is again under the control of the plans so case activity may be subsumed.

8.3.4 Additional agent subsystems

MRCHIPS makes use of additional subsystems to allow the core reasoning components to integrate with its environment. As the Prolog interpreter makes use of a Python hash table data structure to store all of its built in functions, this method was chosen to allow for a relatively fast access to functions and because the table can be dynamically added to. Each of the additional agent components extends the capability of MRCHIPS by adding access to their functions via the function table in the Prolog interpreter.

8.3.4.1 The BNF parser

The BNF parser is a Definite Clause Grammar (DCG) parser that contains the Backus–Naur Form (BNF) rules for Python code. MRCHIPS makes use of the BNF parser to locate the cause of syntax errors. The output from the Python parser generally only specifies the location of errors and the category in broad terms. In a DCG the rules of grammar are coded in first order logic and when a legal phrase is processed a parse tree or semantic statement of the phrase can be returned or, if the phrase is not legal, the point at which the error occurred. As DCGs are powerful enough to be used to parse natural languages, parsing an artificial programming language is relatively simple. Using a DCG allows the Prolog engine to analyse each token of a Python statement in turn.

Figure 8.9 illustrates two of the BNF rules of the DCG for parsing a while-statement and a def-statement and the need rule that first checks for an item and if it is not found reports it missing if it belongs to the set symbol (not shown) or as unexpected for any other item.

```
statement([while|Z0],Z,Err,while(Test,Do)) :-
                test(Z0,Z1,Err,Test),
                next(':',Z1,Z2),
                statement(Z2,Z,Err,Do).
statement([def|Z0],Z,Err,def(name(Name),Args,Stmt)) :-
                next(Name,Z0,Z1),
                need('(',Z1,Z2,Err),
                arglist(Z2,Z3,Err,Args),
                need(')',Z3,Z4,Err),
                need(':',Z4,Z5,Err),
                statement(Z5,Z,Err,Stmt).

need(A,[A|R],R,_) :- !.                    %% progress
need(A,_,[],missing(B)) :- symbol(A,B),!. %% report error
need(_,[B|_],[],unexpected(B)).            %% report error
```

Figure 8.10. Fragment of the DCG for the BNF parser

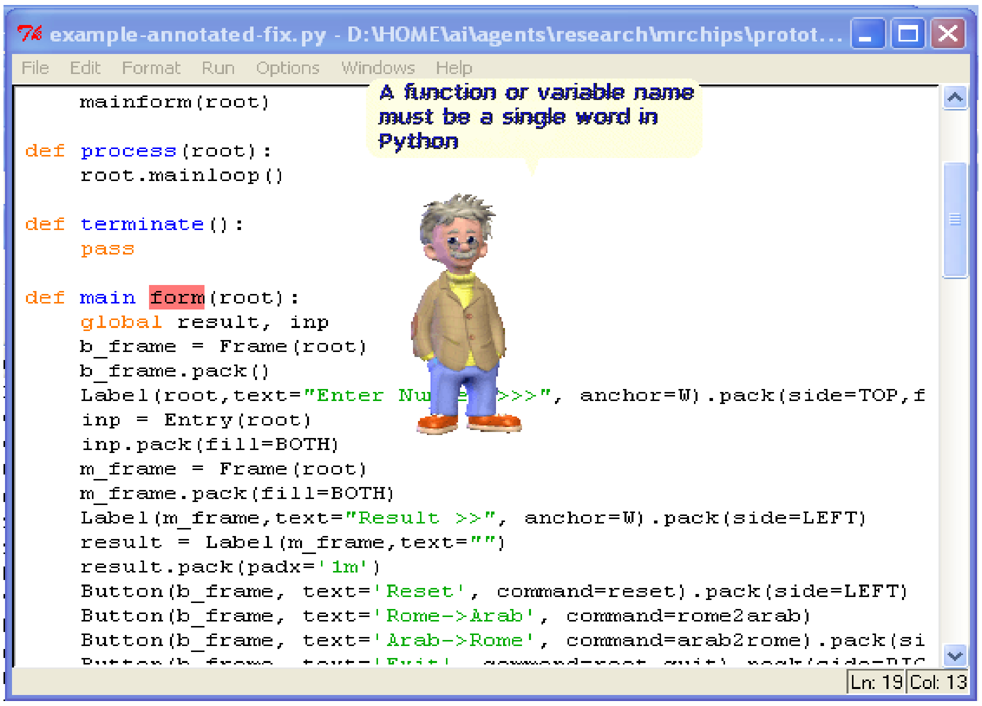
Language keywords are used to identify the type of the statement, variables and constant data isolated, operators and punctuation symbols checked and when an unknown or unexpected token is found the details are returned.

8.3.4.2 Perception

The MRCHIPS agent monitors the Windows desktop to make inferences about what the user is looking at. It looks for the presence of the Python development environment and then clues to the occurrence of errors. MRCHIPS is fairly “short-sighted”. It is able to directly sense the content of its environment in terms of the position of the windows on the desktop. It is able to identify if a window is in plain view, minimised or covered by another, the title message of a window can be read and with some effort the textual contents of editor windows may be sampled. As the Python development environment also runs as a process within the operating system and also makes use of the display to present a collection of Windows and components for interaction with the user. By monitoring the Windows display and sampling the contents of Windows in the development environment the agent is able to infer the behaviour of the learner, examine any source code produced and make appropriate responses. As stated earlier, windowing systems use message passing to allow applications to communicate. For reasons of stability and security typical Windows applications are only aware of their own message queue. It was possible to monitor the Windows message queue globally to intercept messages for other applications such as those for keyboard and mouse inputs, but after investigation this was decided against due to the volume of messages and level of noise. It was found that attempts to filter system messages via the Python interpreter would cause the Windows interface to slow down noticeably. Inferences are therefore made from the arrangement of windows on the desktop and scanning of contents of windows concerned with the Python development environment, by examination of the source code and error messages the appropriate agent response may be selected.

8.3.4.3 Actuators

Actions are the means by which goals can be achieved in the environment. All actions in MRCHIPS are controlled via the avatar subsystem. The results of the cognitive processing of the agent are presented to the world mainly via a Microsoft Agent character, a 2D anthropomorphic animated figure that is able to gesture and perform a repertoire of actions under program control. A mock-up of the agent using the Microsoft Agent interface and working in the Python development environment is shown in figures 8.10 and 8.16. In addition to the animated gestures, the agents's main output method is speech via a speech bubble window that pops up and down as required and is accompanied by audio speech, if a text to speech engine is available on the computer. The texts of the messages are taken from the adapted case selected as a solution to the error. The agent community treats communication as an important facet of an agent's capabilities to help pursue its goals (Wooldridge 2002).


example-annotated-fix.py - D:\HOME\ai\agents\research\mrchips\protot...
File Edit Format Run Options Windows Help
A function or variable name must be a single word in Python
Ln: 19 Col: 13

```
    mainform(root)

def process(root):
    root.mainloop()

def terminate():
    pass

def main form(root):
    global result, inp
    b_frame = Frame(root)
    b_frame.pack()
    Label(root,text="Enter Nu    >>>", anchor=W).pack(side=TOP,f
    inp = Entry(root)
    inp.pack(fill=BOTH)
    m_frame = Frame(root)
    m_frame.pack(fill=BOTH)
    Label(m_frame,text="Result >>", anchor=W).pack(side=LEFT)
    result = Label(m_frame,text="")
    result.pack(padx='1m')
    Button(b_frame, text='Reset', command=reset).pack(side=LEFT)
    Button(b_frame, text='Rome->Arab', command=rome2arab)
    Button(b_frame, text='Arab->Rome', command=arab2rome).pack(si
```

Figure 8.11. MRCHIPS driving the Victor agent character

The work of John Austin and later John Searle in the 1960s attempted to categorise the classes of natural language communications in a field called speech act theory (Russell & Norvig 1995, Wooldridge 2002). The later AI research based on speech acts as a plan or rational action does not really apply to MRCHIPS because the agent makes no choice in whether to communicate or not – if the agent finds a case, it provides an answer as its pedagogical action. In terms of Searle's communication categories MRCHIPS mainly communicates in the form of representatives, informing the learner of information known by the agent. The sentences of the pedagogical actions are structured into three different types: *Explain, Suggest, and Show*. *Explain* actions state what is wrong in the program statement but do not offer a solution. *Suggest* actions offer answers in the correct the form of the line but do not state the cause of the error. *Show* explanations say what is wrong with the suspect line and the form to which it should be corrected. Outputs to the standard Win32 API are used to create dialog box controls and windows, the input control to the agent and the popup window that provides a countdown to the arrival of the agent while the learner attempts to solve the error on their own.

Program line	Type	Pedagogical action
if test = 123:	Explain	A single equal sign '=' means set value to
	Suggest	The symbol for equality should be a '=='
	Show	You need to replace the set value symbol '=' with the equality check '=='

Figure 8.12. Table of the different types of pedagogical actions

Other output from the agent is used to manipulate the windows desktop using a technique called windows automation, the process of injecting messages into the message queue of windows belonging to other applications to simulate key presses and mouse clicks. Automation is used by MRCHIPS to control which window is in view and to scroll to the appropriate line of code when giving error advice.

8.3.4.4 Journaling

MRCHIPS contains a journaling system to record particular events and actions taken. The journaling system keeps a record from the time of its start up to shut down, the identity of the application window that has the user's focus if it is Python, the location of the Python source file, errors detected and the solutions offered by the agent. Each entry in the journal is written to a file in backing storage as the entry is made, so in the event of an abnormal termination the journal is preserved, as well as being preserved in the agent. At the present time the journaling system does nothing that would aid the agent's cognitive processing but the output file is used to analyse the learner's activity and that of the agent. With some adjustments the agent can be made to make use of the historical record in the journal and therefore to access an autobiographic memory (Tulving 2002). Autobiographic memory is an entity's personal history of the events and activities it has experienced; it allows an agent to remain situated in time and able to make higher cognitive decisions, such as reflection (Nuxoll & Laird 2004). Autobiographic memory might become more important for modelling the learner's understanding over the long term, but the facility has not been implemented for the current agent.

8.3.4.5 Reading code

The Python environment produces outputs in two different formats in a windowed environment: syntax errors are detected as the program is compiled and the error message is displayed in a dialog box. Semantic errors are produced at runtime as the code is executed and error messages in the form of runtime exceptions are displayed to the Python shell window. In reality all messages from the Python interpreter are routed to the process output console, but the development environment intercepts the messages and routes them appropriately. When the dialog window for a compiler reported error is detected MRCHIPS locates the source code file from the title bar of the editor window and sends the file through the agent's internal parser. The Python executable carries its compiler alongside the runtime systems, which is why it is more accurately described as an interpreted language, whereas systems such as Java are described as compiled because the compiler and runtime are separate, even though both languages produce object code that is executed in a virtual machine. The output from the parser reproduces the same error message as displayed to the user in a data structure that specifies the type of error, its location and the line of code in question. The message output by the parser does not provide enough information to determine the cause of the error for the novice programmer, so the suspect line is passed to the BNF parser, which further analyses the Python line and isolates the unexpected syntax. The parser operates as a pre-processor to the case-based reasoning system when analysing compile-time errors, it is able to parse the keywords and operators in a line while ignoring the details of data items. When an error is encountered in the form of an unexpected component the parse ceases and an error message returned. For some errors involving a missing component, such as for example a closing parenthesis, comma or colon, the expected component is specified in the error message. The type of the error, the type of the statement and the unexpected component are then used to construct the index, which is sent to the case-based reasoning component.

As the BNF parser is based on a DCG it is able to parse statements containing syntax errors, isolate the program structure that contains the error and in some cases provide information on what the expected structure should be. For the abstraction of 'if' statements, as illustrated in figure 8.12, each level of the hierarchy may have the same meaning but contains different levels of detail. For a CBR system each statement would require a different case to account for that pattern and layers 0 and 1 would require additional cases for expressions involving different data types, different operators, calls to functions, etc. When a runtime error is produced the message is output to the Python shell window and to detect them MRCHIPS monitors the window on a two second cycle for the presence of an error message. The agent is not directly able to read the contents of the window but does so via the Windows clipboard. This is accomplished using Windows automation (see section 8.3.4.3 concerning MRCHIPS actuators) to select and copy the contents. It is then available to be read by the agent for analysis.

5					**ifstatement**				
4			**If**		expression		**colon**		
3		If		**term**	**operator**	**term**		colon	
2	if	**Data**	Operator	**data**	operator	**data**	operator	**data**	colon
1	if	Identity	**Equal**	string	**Or**	identity	**equal**	string	colon
0	**if**	**X**	**==**	**"one"**	**Or**	**X**	**==**	**"two"**	**:**

Figure 8.13. Instruction hierarchy for an if-statement

The last line and the third from last line are parsed to provide the type and location of the error. The information in a runtime error message is fairly detailed. A major problem faced by novice Python programmers is with interpreting its structure and relating the information to a location in the source code, so no pre-processing is performed and the runtime error message alone is used as an index to the case-base.

8.4 Implementation details

The MRCHIPS agent is implemented in Python, but its execution is run as a separate process to any of the code run by the students, that is to say the development environment and tools used by the student do not rely on any service from MRCHIPS and would still run in absence of the agent. The main reason for choosing to implement in Python was due simply to the availability of the Python environment with a known set of libraries on the computers at the University. Other languages such as Java, Pascal or Prolog can also be applicable, but Python's support for rapid prototyping development, abstract level processing, modular and object-oriented development, while allowing support for low level interface to the operating system resources made it an attractive choice for a large experimental program. The suitability of Python for developing AI software has been demonstrated by the development of knowledge-based systems such as the Sherlock expert system shell (Lutz 2001), porting of Lisp examples as demonstrated in the book *Artificial Intelligence: a modern approach* (Russell & Norvig 1995) and similar research investigated by the author (Case 2000).

```
procedure run():
    while number of PAL.Intention > 1:
        foreach stack in PAL.Intention:
            step(stack)

procedure step(stack):
    loop 8 times:
        interpret(stack)
```

Figure 8.14. Algorithm of the PAL top-level execution cycle

Other advantages of using a Python application to analyse Python code, such as access to the internal components of the compiler came to light later in the development. The whole of the agent is encoded in 35 classes across 12 files of Python code, with a knowledge base of 37 plans, 25 cases and 50 Prolog rules and it incorporates the winGuiAuto by Simon Brunning and Tim Couper for driving the Windows automation (explained in greater detail in section 8.6). MRCHIPS runs in Python version 2.4 upward and requires the PyWin32 library to allow access to the Win32 API. Installations in version 2.4 also require installation of the ctype library. One of the first decisions of the design was how knowledge would be represented. The Prolog horn clause was chosen because it provided a rich notation to express ideas and could be directly manipulated by the Prolog engine in the PAL interpreter.

```
Procedure interpret(code):
   instruction = code.pop()
   if instruction == [if, Cond, then, Action]:
      if evaluate(Cond) is true:
         code.push (Action)
   else if instruction == [while Cond, then, Action]:
      if evaluate(Cond) is true:
         code.push (instruction)
         code.push (Action)
   else if instruction == [achieve(Event)]:
      getAlternatePlans(Event, Plan)
      code.push(Plan)
   else if instruction == [assert(Clause)]:
      prolog_assert (Clause)
   .
   .
   else if getAlternatePlans(instruction, Plan):
      PAL.instances.push(Plan)
   else:
      prolog_prove(instruction)
```

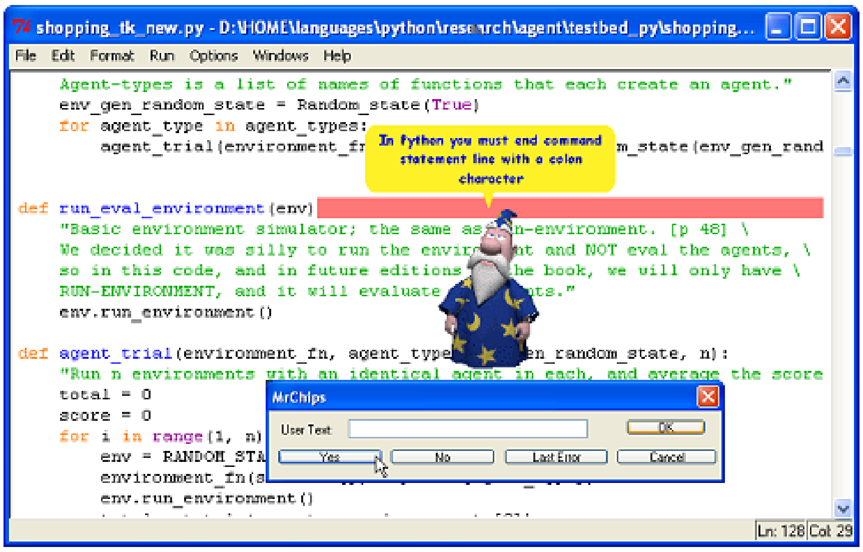
Figure 8.15. Algorithm of the PAL instruction interpreter

It is the PAL interpreter that drives the MRCHIPS agent; it implements the execution cycle described in figure 8.3. The execution cycle is driven by interpreting the instructions contained in one or more of the intention stacks. The interpreter removes an instruction one at a time from a stack and determines how it is to be executed, as illustrated in figure 8.13. Each stack can be thought of as a different execution thread and when a stack is to be executed it is passed to the interpreter, see figure 8.14. PAL itself is implemented as single threaded Python code but performs multithreading by switching the execution between the different intention stacks. If the instruction is a built-in PAL command it is dispatched and executed there. This usually involves manipulation of the stack and controlling the next instruction to be interpreted. If the instruction is unrecognised as a PAL command it is checked against the plans in the agent knowledge base to see if it is the trigger event for an agent plan in the getAlternatePlans function. If the instruction is neither a PAL command nor a plan event it is passed to the Prolog interpreter to execute if it is a recognised Prolog clause. The getAlternatePlans function shown in figure 8.15 selects all plans with the matching triggering event. The guard condition of the plans is also checked at this stage. The guard conditions may contain a true value if the plan is applicable in any context, or if the guard is a more complicated clause it is passed to the Prolog interpreter where it can be checked against the current beliefs.

```
Procedure getAlternatePlans(event,plan):
   plans = knowlegeBase.get(event)
   for plan in plans:
      if unify(event, plan.event) and
         (plan.guard == true or prolog_prove(plan.guard)):
      return True
   return True
```

Figure 8.16. Algorithm for selecting a new plan

All of the other subsystems in the MRCHIPS agent are able to read and write data in the form of Prolog horn clauses to communicate with other parts of the system. The Prolog engine in the PAL interpreter began as a support to evaluate data within the agent, but was re-written and grew over time to support a large subset of the Edinburgh syntax Prolog, including arithmetic, list manipulation, the cut operator, and macro operators. The Prolog parser is implemented as a separate object from the Prolog interpreter. It is therefore available to be “borrowed” by the other reasoning modules within the agent to read their knowledge bases. Unification is also contained in a separate object for the same reason. Matching is able to work with all of the Prolog data types. Variable values are held in a table environment and their values looked up or set during the matching process. On a successful unification the new environment (possibly empty) is returned, otherwise a Boolean False value is returned. The Prolog interpreter is mainly used for the resolution loop that is used to search clauses in the knowledge base. Recursion uses the Python stack and functions in Prolog are implemented as functions of Python code that are called via a table lookup. This mechanism allows the capability of the interpreter to be easily extended by adding new entries to the table. When new functions are added to the agent to extend its perception, cognition or motor capabilities (see sections 5 and 6 below), they are implemented as extensions to the Prolog function table.


shopping_tk_new.py - D:\HOME\languages\python\research\agent\testbed_py\shopping...
File Edit Format Run Options Windows Help
In Python you must end command statement line with a colon character
MrChips
User Text
OK
Yes
No
Last Error
Cancel
Ln: 128 Col: 29

```
    Agent-types is a list of names of functions that each create an agent."
    env_gen_random_state = Random_state(True)
    for agent_type in agent_types:
        agent_trial(environment_fr                              m_state(env_gen_rand


def run_eval_environment(env)
    "Basic environment simulator; the same as   n-environment. [p 48] \
    We decided it was silly to run the envir    nt and NOT eval the agents, \
    so in this code, and in future editions    he book, we will only have \
    RUN-ENVIRONMENT, and it will evaluate         nts."
    env.run_environment()

def agent_trial(environment_fn, agent_type          en_random_state, n):
    "Run n environments with an identical agent in each, and average the score
    total = 0
    score = 0
    for i in range(1, n)
        env = RANDOM_STA
        environment_fn(s
        env.run_environment()
```

Figure 8.17. The mentor agent’s advice to a learner

8.5 The agent environment

For reasons of stability the address spaces for each process on a modern operating system, such as Microsoft Windows and Unix, are all made transparent to each other. This regime allows each process to run without interfering with the activities of other processes. Even if a process locks up or terminates abnormally it can do so without interrupting the rest of the operating system. Each process in the Windows operating system is executed in its own, private four Gigabyte memory address space and nearly all of the resources used by a process are restricted to this memory space. This makes the observation of the activities of one process from within another process extremely difficult. However, it is possible to observe the effects of other processes where computer resources are shared, such as at the filing system and on the display. The Windows desktop consists of a variable number of desktop components of icons, menus and windows that represent the interface to underlying applications, which are addressable via a pointing device or keyboard.

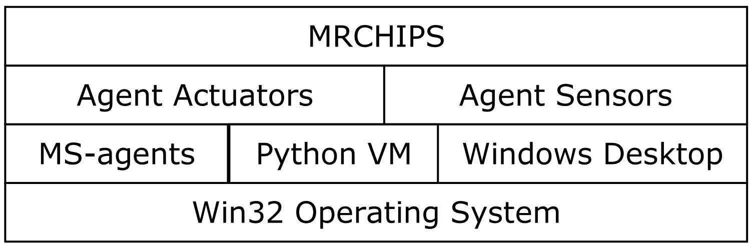


Figure 8.18. The agent's interface to the Win32 OS

Each desktop component is represented as a software structure with a number of attributes that record its appearance, size and position on the desktop, not only in terms of its position in two-dimensions but also indicating its position in front of or behind other desktop components, its Z-order. Windowing environments, such as the Microsoft desktop and Unix based systems that implement X Windows are usually event driven, that is to say in order to provide interactive processing the desktop components respond to event messages sent as a result of mouse movements and clicks or keyboard key presses. The application behind the component is usually in an idle state waiting for an event to make an appropriate response. Message passing and message handling is a major property for programs operating in an event driven windowing environment, as it is a mechanism that allows each application to share the user interface. Microsoft Windows maintains in the order of thousands of types of message that are used to perform functions from the positioning of a component on the desktop to handling communicating between windows. The Python environment is a user level application (it does not execute as part of the operating system). It is used to develop script files that are executed in the Python virtual machine, which is written in C and is executed by the computer's CPU. MRCHIPS is just an application level process that is run in the Python virtual machine (see figure 8.18), but able to access some of the underlying resources of the Windows operating system.

A default installation of Python makes use of the Tk library to provide a cross-platform for producing windowed applications for modern desktop environments. The default editor and development environment called IDLE (from Integrated DeveLopment Environment) was written making use of the Tk library. The Tk based development environment is important to MRCHIPS because it is the one on which the students are taught and so the one targeted by the agent. While the interface provides a simple to use and consistent interface into the desktop environment, it does not provide the same set of features as the underlying operating system. The most noted absent feature in terms of MRCHIPS is an interface for automation control.

8.6 Decision-making in MRCHIPS

Without its knowledge-base the MRCHIPS agent architecture provides only an empty shell incapable of any real reasoning. It is the contents of the of plans, cases, rules and other knowledge structures that are able to use the architecture and provide the agent with its diagnostic capability and behaviours. At the core of MRCHIPS decision-making are the plans that are used to coordinate various modes of the agent's behaviour that can be explained using a kind of finite state machine (FSM), as in figure 8.18 below. However the modes only approximate the FSM as the MRCHIPS architecture allows for concurrent reasoning so the various states are not mutually exclusive.

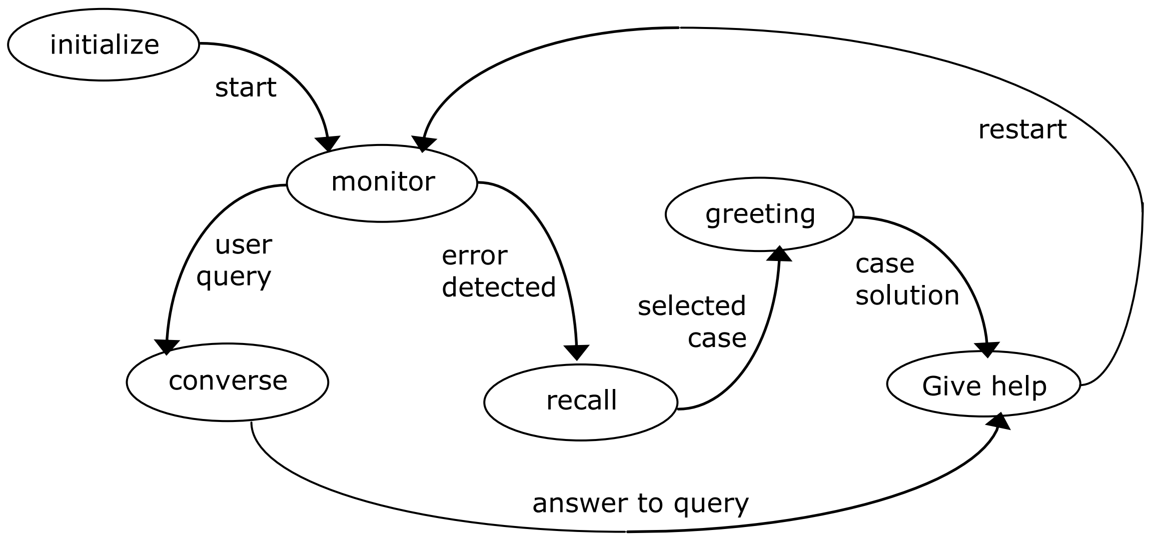


Figure 8.19. Simplified finite state machine for main MRCHIPS behaviours

The way by which each state contributes to the reasoning is as follows:

a) During the initialisation mode the agent announces its presence to the learner and is minimised to be out of the way. When the agent starts MRCHIPS announces his presence and then is minimised to the windows taskbar if the user wishes to manually launch or exit the application.
b) Control is then switched to the monitor mode. This is the main mode of the agent's operation where the desktop is first monitored for the Python development environment and the placement of editor, console and dialog windows. When the learner is not using Python the plans that control the monitor mode poll the desktop every five seconds. Once a Python window is active its contents and the desktop are polled every two seconds for the presence of an error message. The agent polls the desktop on a two second cycle for the presence of the Python IDE. When the IDE is found a record is made of the window and the file being edited.
c) The recall mode prepares and sends the symptoms of the error to the CBR and awaits the selected case solution. As MRCHIPS has no mechanism to directly detect when the user attempts to run their code the agent monitors for the error output from Python. Once an error has been detected the source code is profiled by the agent in a Python subroutine where syntax errors are first tokenised and parsed and the details added to the agent's belief

base or with semantic errors the error message read from the Python shell window is processed and again added to the agent belief base. Once the profiling is complete the profiler routine sends an event to the agent to signal this and the appropriate plans use the output from the profiler to construct the problem case, which is then sent to the CBR module for a matching case.

d) The greeting mode alerts the user that MRCHIPS will offer a solution to the error after a set delay and waits for a period before checking the error again.

e) If the user has not corrected the error by the end of the delay the agent is switched to the "give help" mode where the solution from the case is used to provide help to the user. The selected case is placed into the agent's beliefs knowledge base and activated by a call to a plan called executeCase, which selects the steps of the solution from the case and inserts them into the BDI's intention stack for execution. Control is then passed back to the monitor mode to check for future errors.

f) The converse mode is activated when the agent awaits input from the user. Its initial plan is spawned from a separate intention in the BDI and in effect operates in parallel with the monitor planning. The converse mode is used to accept text input from the user and sequence the natural language parsing and question answering operations.

There are also additional sub states that oversee the handling of other components of MRCHIPS, handshaking with system resources and overseeing input and output operations. Other plans in MRCHIPS are mainly concerned with "housekeeping" tasks such as controlling the MS agent character and coordinating communication with the user. Some functions such as monitoring which window is on top of another are coded directly in Python for reasons of speed and efficiency.

8.7 Related work

The MRCHIPS agent shares some of the features of the other pedagogical agent systems explained in the literature review as well as introducing new features to address the requirements of its domain.

Agent	Reasoning	World	Interactive	Pedagogy	Environment	Source
Steve	Rule-based	3D	Yes	Coaching	VRML	C/Soar
FatiMA	OCC	3D	Episodic	Immersive	Ogre 3D	Java
BodyChat	Procedural	3D	Yes			C++
PPP	Procedural	2D	No	Lecture	Document	
Jacob	Procedural	3D	Yes	Coaching	VRML	Java
Adele	Planner	2D	Yes		Web-applet	Java
MRCHIPS	BDI/CBR	2D	Yes	Coaching	Windows	Python

Table 8.3. Comparison of MRCHIPS with other virtual pedagogical agents

Like the Steve and FatiMA agents MRCHIPS implements a cognitive agent architecture. FatiMA agents simulate emotions as an important part of their decision-making, Steve and Adele have no facility for this and the author reported no adverse effects as a result. The Jacob and PPP agents perform their pedagogical tasks with little reasoning capacity, certainly less than available to the other systems, but as a result are less interactive than the others. Unlike most of the virtual agents MRCHIPS exists in a 2D desktop environment, because that is where the learner works but there is no reason why it cannot be adapted to work in a 3D world. Adele and PPP are based in 2D environments for the same reason as MRCHIPS in order to make use of pre-existing resources to conduct interactions. In terms of the interaction MRCHIPS has most in common with the Adele system with the main difference in the scope of the pedagogy. Adele presents teaching materials while MRCHIPS is guided only by the code produced by the learner. Most of the other systems provide additional tools to allow domain experts, who may not be programming experts, to prepare subject materials; MRCHIPS only allows this by direct alteration of its knowledge base. A summary of the features of MRCHIPS in comparison with other virtual pedagogical agents is shown in table 8.3.

8.8 Summary

Previous similar systems used CBR to either extend the reasoning capabilities of a planner, such as with Prodigy/Analogy, or to completely implement BDI reasoning, such as CBR-BDI. The MRCHIPS architecture differs from these systems in that the CBR provides its diagnostic capability and the BDI facilitates this by its interaction with the environment and the learner. The agent can pursue multiple goals while interleaving the execution of multiple plans and the diagnostic case-based reasoning. The agent makes use of domain knowledge in the form of cases that can be rapidly selected and used to initiate additional goals and plans. Additional support subsystems allow MRCHIPS to exist as an independent application on the MS Windows operating system, able to monitor and interact with the Python development and the desktop environment. Although capable, MRCHIPS still lacks some features that were designed for but not fully developed. First, a natural language interface to the case-base would allow logical errors to be analysed were MRCHIPS not able to determine the cause. The second is a mechanism in the CBR to record new cases. Encoding cases as generalised examples has reduced the effect of the absence of this feature. The third feature is a mechanism to recall and make use of events stored in the journal to inform decision-making. This would act as episodic memory and allow the agent to be situated in time. As a result the agent is unable to maintain a model of the user from which to reason and produce primarily reactive behaviours. However MRCHIPS is capable of providing sufficient analysis and mentoring of novice errors. The evaluation of MRCHIPS and a discussion of its performance are given in the following chapters.

Chapter 9:

Research methodology and experimental design

The purpose of this chapter is to examine the experimental design for the evaluation of the performance of novice programmers working with MRCHIPS. In the following sections a discussion is given for the suitability of different methods for conducting different types of research. This is followed by an explanation of the rationale to use the selected method and the strategy behind the data collection. A discussion is then given for the options influencing the choice of research method for the evaluation of the MRCHIPS agent.

9.1 Research methodology

The purpose of a research methodology is to structure the collection of data that will be used towards the testing of an academic hypothesis. There are various approaches to the collection of research data and the method of collection generally depends on some combination of the nature of the subject and the aims of the research. However, data gathering can be categorised into 3 general groups: those that are largely quantitative, those that are largely qualitative and hybrid research methods.

9.1.1 Quantitative research methods

Quantitative research is generally used to measure a collection of parameters with the aim of verifying or questioning a theory or hypothesis. According to Walliman (2011) the primary purpose of quantitative analysis is to measure, make comparisons, examine relationships, make forecasts, test hypotheses, construct concepts and theories, explore, control, and explain. Although quantitative analysis deals with data in the form of numbers and uses mathematical operators, such as statistics, to investigate their properties the measurements are guided by the kind of question asked and can be as subjective as a qualitative method. Quantitative research involves the collection of data so that information can be quantified and analysed in order to support or refute a given theory. "Quantitative research begins with a problem statement and involves the formation of a hypothesis, a literature review, and a quantitative data analysis." (Williams 2007). Quantitative research methods often involve experimentation where a series of measurements or counts may be taken, although it is also possible to use some of the methods from quantitative research such as the survey where participants are invited to rate or categorise a given experience.

9.1.2 Qualitative research methods

Qualitative research methods deal with data expressed mainly in words that offer descriptions, opinions, beliefs, accounts, experience, etc. Qualitative research is usually carried out when first exploring a domain (Wisker 2001) and is more often used where individuals or groups of people are the focus for the research. The main methods for qualitative data gathering are:

- The interview: A face-to-face discussion with human subjects. It is usual for one of the participants to posses experience or knowledge of interest and the other to make a record of the event, such as by note taking.
- Focus groups: small groups of participants brought together to focus on a given issue. The group are presented with questions and scenarios regarding issues and asked for their response or opinion.
- Participant observation: the researcher joins the group as they are going about their activity and studies their activity. This is recognised as a highly subjective data gathering method, as the observer may be too distant to have enough of a full view of the subjects or so deeply immersed that they cannot remain objective.
- Personal learning logs: the researcher maintains a log recording their observations, experiences and reactions as data is gathered.

9.1.3 Hybrid research methods

A hybrid research method (sometimes known as mixed-mode, mixed-method or fused research) is an approach that relies on a combination of quantitative and qualitative methods (Wisker 2001). Although it would not be unusual to find quantitative techniques used in a qualitative research or quantitative methods in qualitative research the hybrid methodology is more accurately used to refer to the combined analysis from different methods contributing to the testing a research hypothesis. There are various techniques for the analysis of hybrid data one common method is to count the number of times an item of qualitative data occurs. Another hybrid method might is to enumerate the frequency of qualitative themes within a sample (Driscoll *et al.* 2007). Quantitative analysis is usually used to provide detailed assessment of the magnitude of phenomena and qualitative data used to provide a deep understanding of a domain. The hybrid research methodology allows researchers to overcome the limitations of using a single method and provides advantages for exploring more complex research questions.

9.2 Review of research objectives

For the evaluation of MRCHIPS it is worth reconsidering the main hypothesis of this research, which was to examine the best approach to data gathering to address the assertion:

The aim of this research is to investigate whether the use of an animated pedagogical agent would provide effective mentoring support to novice programmers as they learn their first programming language.

The questions of the hypothesis that can be addressed by the evaluation are:

1) To demonstrate that the presence of the agent produces a positive effect on the student's learning;
2) Within that, how much is as a result of the presence of a personality and how much is from the content of the information provided by the agent.

From the review of virtual agents, in chapter four, researchers have noted that the presence of a virtual agent tends to increase a user's performance in tasks irrespective of whether the agent provides domain information or not (Lester *et al.* 1999). It is believed that people respond to the personality of the agent as they would to the presence of a person. Research from psychology suggests the effect of people surrogates show similar increase in performance in other fields (Lester *et al.* 1999). However it is necessary to show a material improvement of the learner's ability to cope with programming errors as a result of the presence of MRCHIPS. This would indicate the need for a quantitative evaluation where the measure is of the learner's use of domain knowledge. An assessment of MRCHIPS could be carried out where students were asked their opinion of working with the agent by interview or survey. A qualitative measure might indicate a learner's preference (or not) for the presence of the agent but offer no indication of the effectiveness of MRCHIPS in helping students to learn to program. The use of an experiment with quantitative measures allows for the controlled testing of MRCHIPS where extraneous factors can be limited.

9.3 Research Design

In order to demonstrate the effectiveness of MRCHIPS it is necessary to show that novice-programming students are able to make more progress in practical exercises with the agent than they would without and that this is as a result of the agent. An ideal study would allow for two groups of students to be evaluated over the course of an academic year the time normally taken to teach Python. One group, the test students, would have access to MRCHIPS during the evaluation period the other group would not have access and would act as a control group. During the study comparisons would be made of the relative progress of one group against the other with a large enough sample for the study so that individual factors such as teaching skill, age, prior experience and motivation of the individual could be mitigated. Then any difference would be attributable to the effect of the agent. However to use MRCHIPS in such a study, where a learning tool were deliberately denied to some students, would raise issues of ethics in a university environment where what is learned by students will have a material effect on their overall progression. Another difficulty would be that test results towards the end of a long study would be expected to show a smaller difference between both groups than in the beginning as the agent supports novice level learners and both groups would continue to learn throughout the period.

During the academic year the students' progress on the "foundations of programming" module is tested in three different exercises that demonstrate different skills at various stages of the course. Towards the end of the first term students are given a comprehension exercise consisting of about ten short answer questions and small fragments of code requiring explanation. During term two they are given a complete programming project usually to provide a custom user interface to a database application. This is largely a design-based challenge allowing the students the chance to apply what they have learned. The third assessment is a practical exercise, called a Time-Constrained Assignment (TCA) and designed to be the equivalent of an end of year examination, but testing many real-world programming skills. For the TCA students are challenged to correct a faulty Python program within a fixed period of time. The students are allowed to use programming books and lecture notes, but have to correct the program individually.

Rather than devise a completely new experimental framework for the agent evaluation it was decided to base the testing around the (TCA), exercise used to assess students. The TCA provides the clearest experimental structure for testing the effectiveness of the agent and although it might appear an artificial exercise it provides a good real-world test of programming skills as professional programmers are often expected to be able to maintain and make use of code originally created by other people.

The observation of novice errors was also at the data gathering stage of the research. Although the results of the observation were presented as a trend, shown in figure 6.6 of chapter six, the data was primarily qualitative; the real value of each error was its occurrence as that was then used to populate the knowledgebase for the CBR.

9.4 Experimental overview

For the agent experiments three trial groups were run: the first group of novice students working without the MRCHIPS agent, the second group of experienced students also working without the agent and the third group of novice programmers who were mentored by the MRCHIPS agent. Throughout the rest of the text the groups will be referred to as novice, experienced and mentored respectively. The novice and experienced groups were to act as a control providing a measure of how students perform normally in the TCA. The mentored group would also be asked to complete a questionnaire to provide some qualitative information about the experience of working with the agent. The evaluation of MRCHIPS working alongside novice programmers allows evidence to be gathered to examine the first two hypotheses of this research. The first of these was:

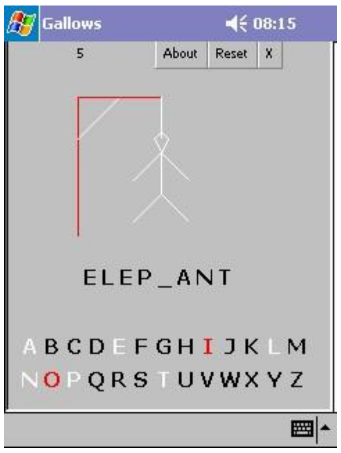
1) An intelligent agent with an anthropomorphic interface can provide effective mentoring support to novice programmers learning their first programming language.

To measure the effectiveness of the mentoring the evaluation should show that mentored students are more likely to produce work of a higher standard than would be expected of a similar novice programmer and that the mentoring aids their learning.

The second hypothesis was:

2) The use of an animated virtual character user interface increases the learner's engagement with problem solving in the programming environment.

Indicators such as positive opinions about using the agent from the learner or a willingness to explore beyond the core requirements of exercises will be assumed to be a measure of increased engagement for this evaluation.



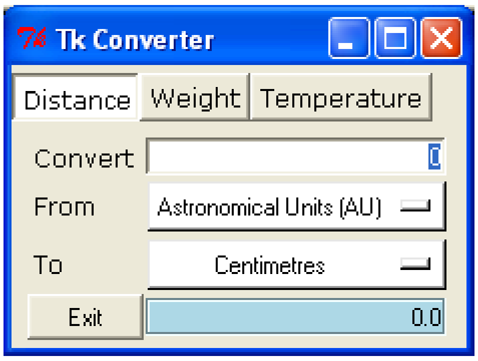


Figure 9.1. The user interfaces for the hangman and unit converter applications used by the control groups

The method of evaluation chosen was to compare the problem solving of a test group of novice learners working with the aid of MRCHIPS against those of two control groups of learners working without the agent. The control groups were novice programmers tested at the beginning of their course, after six weeks of Python study when students were familiar with the Python tools but very much at a novice level of skill and a second group of more experienced programmers tested after 24 weeks of study towards the end of their course. Three different Python applications, which made use of the Tk/Tickle library to provide a Windows interface, (see Figure 9.1) were used as programs to debug for the different evaluation groups. The level of complexity for each program was approximately the same, although the numbers of errors and their complexity was different, depending upon the curricula requirements for the control group. The program for control group one, the non-mentored novices, contained the fewest and most simple errors while the experienced programmers and mentored novices group contained more challenging errors. The code used by the mentored students contained a few duplicated errors to help examine for signs of learning.

The challenge of the exercise was for the students to find and correct some twenty syntactic, semantic and logical errors in a two-hour period. The test is run as an open book exercise, meaning students may use any printed Python or programming related material. The errors in the test program are of a similar type to those highlighted in Chapter three.

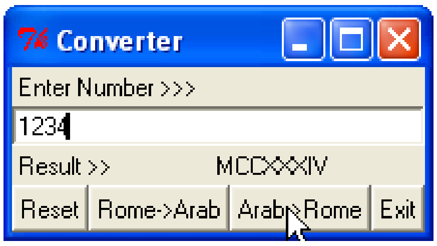


Figure 9.2. The user interface numerical converter application used by the mentored evaluation group

The test program used for the mentored evaluation group was a small Python application to convert values between Arabic and Roman numerals, see Figure 9.2 and contained eleven syntactic and semantic errors. Some of the errors were repeated, to allow testing of whether the user had learned through the guidance from the agent from the first instance of the error enough to recognise and solve the second instance of the error without guidance. The MRCHIPS agent was capable of detecting and offering assistance for all of the error types included. The errors used in the evaluation program are listed below with a brief explanation of what they were designed to elicit from the subject. Note: the errors are listed in the order the Python compiler detected them.

```
def mainform(root)
```

1. The first error was the missing colon at the end of a function definition statement. This produces a syntax error that is simple for the agent to determine and provide direct help to solve and designed to allow the subject to make a start. This is a compile-time error.

```
  m_frame = Frame(root)
 m_frame.pack(fill=BOTH)
```

2. The second problem is an un-indentation error this again produces a simple error for which the agent is able to provide direct help.

```
def arab 2rome():
```

3. The next problem was a split in the name for a function, in Python a function or variable name must be a single word.

```
    If not isinstance(arabic, type(0)):
```

4. The case sensitivity of Python was used for the next problem; the uppercase 'I' in the 'if' invalidated the keyword.

```
def roman _to_int(roman):
```

5. The space in the name definition of the function is the error for this problem – the same as the error in the third problem. Again this is to test if the subjects were learning and if they were able to solve the problem without the agent.

```
    if int_to_roman(total) = roman:
```

6. This syntax error has the assignment operator in the place of the equality operator in the if statement.

```
def reset(root)
```

7. The problem in the reset function definition is a repeat of the first problem; this was to see if the subjects were able to provide a correction without the aid of the agent.

```
    root = Tk()
     initialise(root)
```

8. This is another indentation error. This time the line is indented one space too many. If the subject corrects the error without MRCHIPS it would indicate learning.

```
    process(roo)
```

9. This is the first of the run-time errors. It is a spelling mistake with the last letter omitted from the variable name root.

```
    if not 0 < Arabic < 4000:
```

10. This is a case-sensitivity error with the Arabic variable name; as all other instances of the variable are in lower case.

```
Roman = roman.upper
```

11. This error contains two logical errors. The first is the absence of parenthesises (or brackets) to indicate a function call. The other logical error is that the function name should be lower to change all of the characters in the roman string to lower-case.

The mentored volunteers were given forty minutes to complete as much of the program as they could manage and then asked to complete a questionnaire about the experience (see Appendix E). The activity of MRCHIPS during the session was logged by the agent's journaling system and at the end of the exercise the log file, program source code and questionnaires were collected for analysis.

9.4.1 Experimental setting

The material from a total of thirty-three people was used in this study. There were ten students in control group one, novice programmers who worked without the agent. Fourteen more experienced student programmers also worked without the agent in the second control group. Both groups were from a cohort of year one university undergraduate students. The tests they carried out were also as a part of their normal curriculum activity.

Group	**Experience (wks)**	**Participants**	**Agent present**	**Total Errors**	**Duration (mins)**
Novice	6	10	No	10	60
Experienced	24	14	No	18	120
Mentored	0	9	Yes	11	40

Table 9.1. Details for the experimental setting

The experimental agent mentored group consisted of nine volunteer novice programmers who worked with the agent. The arrangement for each test group is shown in table 9.1. Due to scheduling issues the volunteers for the mentored group were not from the initially identified student body; suitable novice programming students would usually be available at the start of an academic year but the agent software was not stable enough for testing at this time. Instead volunteers were gathered with suitable computing experience but with limited experience of programming, or of Python. The exercises were run as individual sessions, six of the nine were run in the presence of the researcher and three were carried out remotely with the results emailed back to the researcher.

9.4.2 Experimental limitations

There are three main limitations with the method of experimentation; the number of participants in the test group is very small which could lead to inaccurate findings as unusual results may have larger influence than normal. However, the t-test analysis, discussed in the next chapter, can provide a measure of the confidence for the accuracy of the findings. Second, no account is made for any prior programming abilities for the participants of the mentored group the only test taken was for any knowledge of Python programming. Ideally pre-testing of the individuals could have been performed to assess their base-line ability however, students in the control groups also had different prior programming experience so these conditions for all groups would be the same. Third, using the TCA as the basis for the experiment provides quantitative data on syntax and semantic errors but does not allow testing for problem solving with logical errors. Logical errors start to affect students later in the learning process as the programs become more sophisticated, see figure 6.6 for a measure of this trend, as this experiment is concerned with testing novices the TCA was considered to be a sufficient challenge.

9.5 Ethical considerations

As the TCAs were part of the curriculum of the student participants and would contribute to their academic progress it was decided to test the control groups before the completion of the working agent, in order to avoid any potential ethical problems arising from withholding a learning tool from some or all of the students. Volunteers from the subsequent cohort of students would then form the mentored group. Another consideration was the requirement for the novice and experienced programmer groups to be given different challenges for their TCA exercises although an identical exercise would have been more convenient and the different TCA exercises can be accommodated by correlation of the individual problems across each.

9.6 Summary

The decision was taken to use a quantitative data gathering approach to evaluate the effectiveness of the MRCHIPS agent. The experimentation would be based around supporting students to complete the TCA practical examination. Three test groups would be used in the evaluation: novice programmers, experienced programmers to provide control data and mentored programmers to provide data of working with the MRCHIPS agent. This approach allows the experimentation to be based around a pre-existing evaluation infrastructure and tests the agent in a real-world application.

Chapter 10:

Evaluation of MRCHIPS

This chapter evaluates the effectiveness of a mentoring agent, MRCHIPS, in providing mentoring support to novice programmers and helping novice Python programmers overcome the common Python syntactical, semantic and logical errors. First findings from the evaluation, using the framework described in the previous chapter, are presented. A brief description is then given of the reasoning behind the choice of the t-test and correlation coefficient statistical methods used for the analysis. The findings are analysed in order to determine how well the evaluation is able to test the hypothesis. Finally, the limitations of the approach taken with this study are examined.

10.1 Findings and analysis

A summary of each error and the numbers of learners in each group able to correct them is shown in table 10.1 below.

Error	Novice (group size: 10)	Experienced (group size: 14)	Mentored (group size: 9)
Missing colon 1	9	14	9
Indentation 1	10	14	9
Split name 1	7	14	8
Incorrect operator	9	14	8
Missing colon 2	5	14	4
Indentation 2	5	13	5
Split name 2	NA	NA	6
Spelling 1	4	7	5
Case sensitivity	1	13	6
Missing bracket	0	4	1
Spelling 2	NA	6	3

Table 10.1. Results for number of errors corrected by each group

The experienced coders group were able to correct most of errors, but the results were more varied for the other groups. In all groups the majority of participants were able to correct the earlier occurrence of errors. Almost every participant corrected the first missing colon and indentation errors. The split variable name and incorrect operator errors were also corrected by most. The case sensitivity error was uncorrected by all but one in the novice group, while all but one of the participants in the experienced group and the majority of the mentored group were able to correct the same error. The errors that were the least well addressed by all groups were the errors in spelling and missing parenthesis. The spelling error would be highlighted only at runtime and reported as a missing variable while the missing parenthesis is a logical error that could not be directly detected by the language compiler/interpreter, but might produce an error at a later stage or merely an incorrect answer.

Subject A was a computer user with no programming experience and managed to introduce new errors in attempting to fix the code.

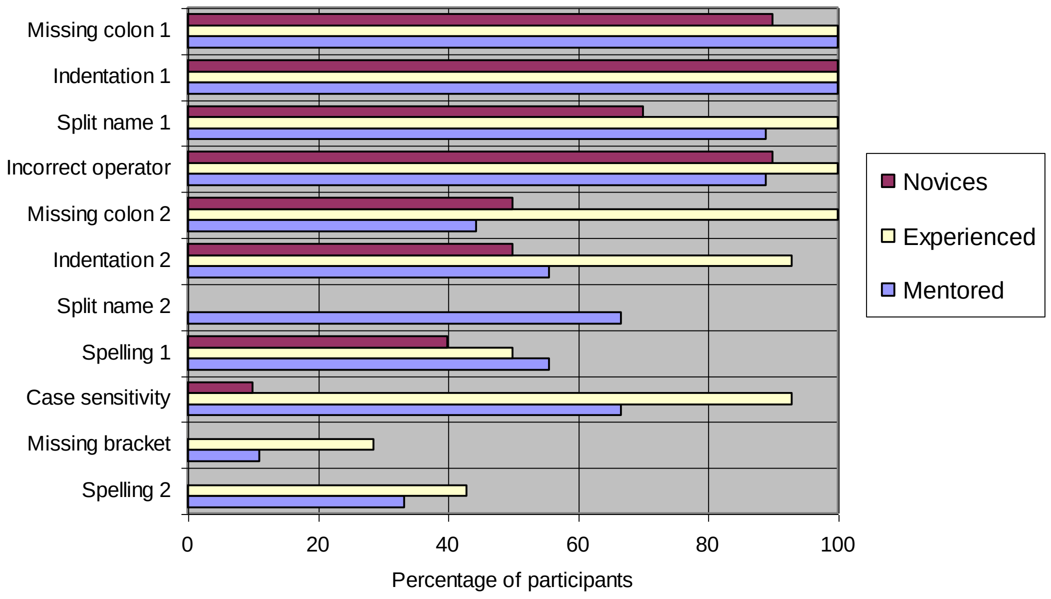


Figure 10.1. Proportion of errors corrected by each group

The marks and percentage grades for individual students in group one, the un-mentored novices, are shown below in table 10.2.

Mark	7	6	6	6	6	4	4	4	4	3
%-age	78	67	67	67	67	44	44	44	44	33

Table 10.2. Results for control group 1, novice programmers

The mean number of errors corrected was five with a standard deviation of 3.54 and an average grade of 55.5 percent. The grades for individual students in group 2, the experienced programmers, are shown in table 10.3. These students produced a mean number of eight errors corrected with a standard deviation of 3.97 and an average grade of 65.7 percent.

Mark	10	10	10	9	9	8	8	8	8	7	7	7	7	5
%-age	100	100	100	90	90	80	80	80	80	70	70	70	70	50

Table 10.3. Results for control group 2, experienced programmers

The grades for individual participants in mentored group of novice programmers are shown in table 10.4. These students produced a mean of seven errors corrected with a standard deviation of 2.58 and an average grade of 65 percent.

Participant	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**M8**	**M9**
Mark	11	10	9	8	7	6	5	5	4
%-age	100	91	82	73	64	55	45	45	36

Table 10.4. Results for group 3, mentored novice programmers

Of the nine mentored participants in the evaluation group only one, M1, was able to correct all of the errors. However participant M2 was also able to correct enough of the errors to produce a running version of the program, although the application would not produce a correct result.

In a comparison of results for each experimental group the grades for the experienced coders clustered towards the higher grades, producing higher average grades than the other groups, while grades for less experienced learners were distinctly lower, figure 10.2. The results for the mentored group were fairly evenly distributed across grades. A larger sample might cause a more conventional distribution, however some of the participants were able to perform better and produce a higher average grade than non-mentored novices.

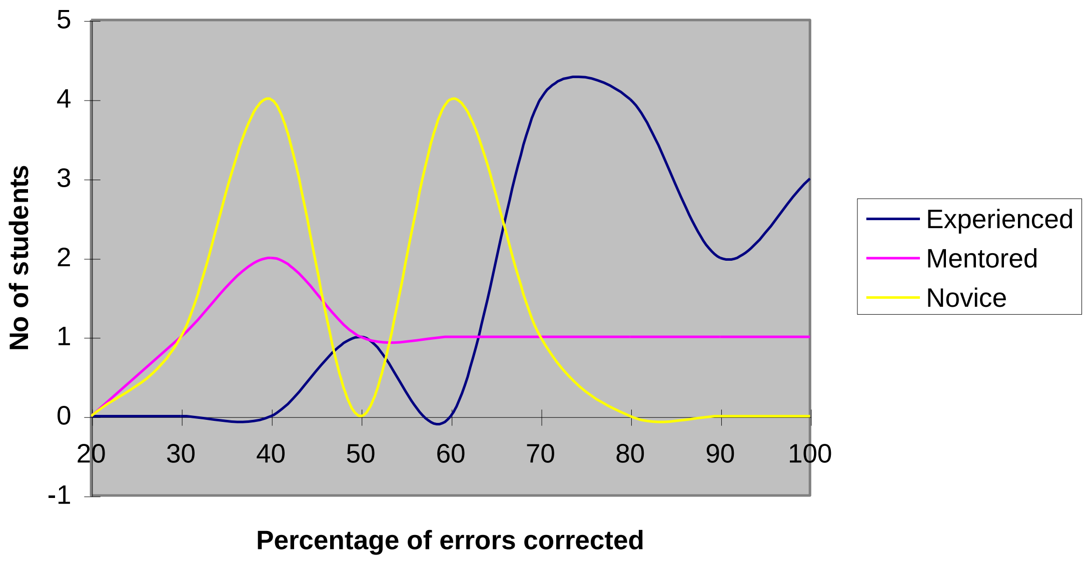


Figure 10.2. Comparison of grade distribution for each experimental group

Participants M1, M2, M4 and M5 were able to solve one or more of the repeated errors without the aid of MRCHIPS, the agent's journal recorded the offer of help as cancelled but the errors were still corrected, see table 10.5. These patterns were interpreted as indications of learning as the subjects were able to recognise and solve problems on their own.

Participant	**Errors solved with agent**	**Errors solved by self**	**Total**	**Tutor present**	**Time (mins)**
M1	7	4	11	Yes	38
M2	6	4	10	No	41
M3	9	0	9	No	67
M4	6	2	8	Yes	40
M5	6	1	7	No	35
M6	6	0	6	No	60
M7	5	0	5	Yes	40
M8	5	0	5	No	46
M9	4	0	4	No	27

Table 10.5. Results for evaluation group, subjects and MRCHIPS

Four of the participants (M1, M4, M5 and M7) chose independently to keep the MRCHIPS character on the desktop as they worked, even though the instructions indicated the MRCHIPS character be minimised when not in use. Student M4 reported that the text-to-speech feature did not work on their computer but s/he was still able to proceed. Student M8 reported that MRCHIPS shutdown during the processing of the fifth error and was unable to progress beyond that point even after a system reset. Attempts by the researcher to determine the cause of the error or to reproduce the problem were unsuccessful.

10.1.1 The t-test analysis

The t-test is carried out to test the hypothesis that the presence of the agent, MRCHIPS, is responsible for the difference in performance between the two groups: novice control group and mentored group. The t-test is used to estimate the mean population distribution in data when the sample size is small. It is based on the assumption that random data samples should exist on a normal distribution curve. The t-test relies on the t-distribution, which is a family of continuous probability distributions that are used for estimating the mean population distribution, see figure 10.3. By analysis of values from a sample, such as the mean and the standard deviation, and a t-distribution, the t-test calculation is able to provide a comparison of the performance between two independent (or unpaired) samples (Madsen 2011). The t-test also allows for a measure of confidence for results when the sample sizes are statistically small (Freund & Simon 1996).

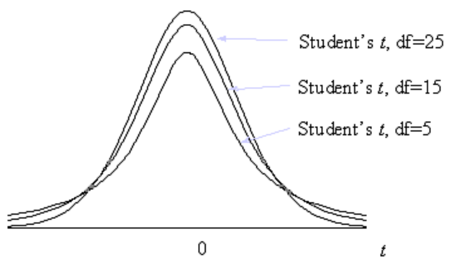


Figure 10.3. T-distributions with different degrees of freedom (courtesy of StatsDirect Limited)

The two-sample t-test compares the mean values between two sets of data. The analysis tests a null hypothesis that proposes the population means related to two random samples, from an approximately normal distribution, to be equal, i.e. u1 – u2 = 0 and an alternate hypothesis where the means are the inverse of the null hypothesis, i.e. u1 – u2 ≠ 0. A probability is calculated as a measure of the chances of observing a random value when the null hypothesis is true. If the probability value is below a given threshold then the null hypothesis can be ruled out and the alternate hypothesis shown to be valid.

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

Figure 10.4. The t-test expression (courtesy of J. P. Key. Oklahoma State University)

However the t-test expression in figure 10.4 is not always accurate when the sample size is less than 30. The t-test expression for statistically small sample groups with a different variance is given in figure 10.5 below. Where the symbols have the same meaning as for expression 10.4 and the terms $\Sigma(x_1 - \ddot{x}_1)^2$ and $\Sigma(x_2 - \ddot{x}_2)^2$ are the sum of the squared deviations for sample 1 and sample 2 respectively.

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\left[\frac{\Sigma(x_1 - \bar{x}_1)^2 + \Sigma(x_2 - \bar{x}_2)^2}{n_1 + n_2 - 2}\right]\left[\frac{1}{n_1} + \frac{1}{n_2}\right]}}$$

Figure 10.5. The t-test expression for small samples (courtesy of J. P. Key. Oklahoma State University)

The sampling distribution is the t-distribution with $n_1 + n_2 - 2$ degrees of freedom. Once the t-value has been calculated it can be compared against the standard t-distribution table for the corresponding critical value for the measure at which the value is said to be significant. A more detailed treatment of the reasoning behind the t-test is given in Coolidge (2000) and can be found in the literature.

The p value is a calculation of the probability of producing a rare value that is outside of the t-distribution (Madsen 2011). The conventional level of significance for a statistical measure is tested at the $p = 0.05$ value, that is to say when the probability of rejecting a correct hypothesis is less than 5% (Coolidge 2000).

10.1.2 The t-test calculation

The preliminary analysis for the data collected from the non-mentored novice group (table 10.2) and the mentored novice group (table 10.4) of programming students are shown in table 10.6 below. The results from the experienced programmers group is not needed to test the experimental hypothesis and is not considered for this analysis as the hypothesis is concerned with a comparison of the relative performance of the novice programmers working with or without the MRCHIPS agent.

Novice	**Mentored**
7	11
6	10
6	9
6	8
6	7
4	6
4	5
4	5
4	4
3	

Table 10.6. Empirical data from the novice and mentored groups

The null hypothesis is for the inverse of the experimental hypothesis, that the presence of MRCHIPS has no effect on the performance of novice students, that is to say $u_1 - u_2 = 0$, the mean difference between the performances of the groups of novice students irrespective of any assistance will be or close to zero.

Calculation of $\ddot{x}_1$

$$\ddot{x}_1 = \frac{7+6+6+6+6+4+4+4+4+3}{10}$$

$$\ddot{x}_1 = 5.0$$

Calculation of $\ddot{x}_2$

$$\ddot{x}_2 = \frac{11+10+9+8+7+6+5+5+4}{9}$$

$$\ddot{x}_2 = 7.2$$

Calculation of the sum of the squared deviation for the novice group

$$\begin{aligned}\Sigma(x_1 - \ddot{x}_1)^2 &= (7 - 5.0)^2 + \dots + (3 - 5.0)^2 \\ &= 16\end{aligned}$$

Calculation of the sum of the squared deviation for the mentored group

$$\begin{aligned}\Sigma(x_2 - \ddot{x}_2)^2 &= (11 - 7.2)^2 + \dots + (4 - 7.2)^2 \\ &= 47.6\end{aligned}$$

	Novice	**Mentored**
Mean (x)	5.0	7.2
Std dev (s)	1.3	2.4
Number (n)	10	9
Squared deviation $\Sigma(x - \ddot{x})^2$	16	47.6
Degree of Freedom (df)	9	8

Table 10.7. Preliminary analysis of the sample research data

Substitution of the values from table 10.7 and the sums of the squared deviations into expression from figure 10.5 gives the following formula

$$t = \frac{5 - 7.22}{\sqrt{\frac{16 + 47.6}{10 + 9 - 2}} * \frac{1}{10} + \frac{1}{9}}$$

$$t = \frac{-2.22}{\sqrt{\dfrac{63.6}{17} * \dfrac{19}{90}}}$$

$$t = \frac{-2.22}{\sqrt{0.7898}}$$

$$t = -2.505$$

The value for t was calculated to be -2.505. The sign of the t value indicates that it is the value for the mentored group that produces the larger mean values. The directional component of the research hypothesis is that mentored novices, the x_2 group, perform better than non-mentored novices, group x_1 therefore the negative value produced for t is consistent with the hypothesis.

The calculated value t = -2.505 exceeds the critical value of t = $\pm$ 2.110 at p = 0.05 with a df = 17. The calculated value t = -2.505 does not exceed the critical value of t at p = 0.1 (for df = 17) which is t = $\pm$ 2.898. This would indicate a p value between 0.05 and 0.01. A precise value for p can be calculated directly from a spreadsheet program using the TTEST function. The p value from the TTEST calculation was found be 0.016, which corresponds to a 1.6% chance of rejecting a correct hypothesis and is consistent with the t value calculated above. The mean difference between the data is therefore higher than would be expected from random chance alone with a very small probability of producing a rare value. As the t value does exceed the critical value the null hypothesis is rejected and the difference between the means of the two groups is significant. It can be concluded that on average novice students working with the MRCHIPS agent produce higher marks than those working without.

10.1.3 Pearson's correlation coefficient

Pearson's correlation coefficient allows a measure of the relationship between the activity of MRCHPS and the performance of the mentored students. The correlation coefficient is a statistical measure of the strength of linear dependence between two variables. It is expressed in values from +1.0, indicating a direct relationship between the variables to −1.0, indicating an inverse relationship. A value of 0.0 indicates no relationship between the variables.

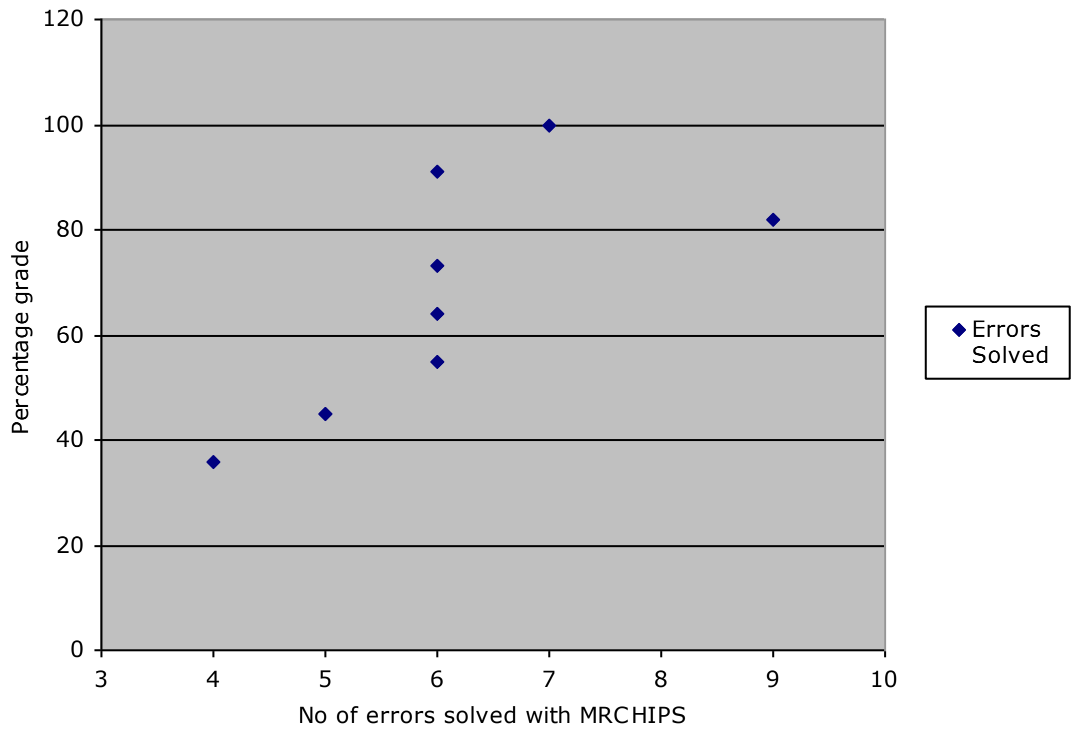


Figure 10.6. Number of errors solved with MRCHIPS and student grade

A comparison of the number of errors solved with help from MRCHIPS and the final grade attained is shown in the scatter chart in figure 10.6. Analysis of the data shows a positive correlation coefficient of 0.73, which would indicate MRCHIPS to have a significant influence on a student's success. Further analysis of the results for the help from MRCHIPS and errors that students were then able to correct without the agent's help, shown in figure 10.7, which would indicate student learning gives a correlation coefficient of only 0.21. This is an indication of some correlation, but is not clear enough to be significant.

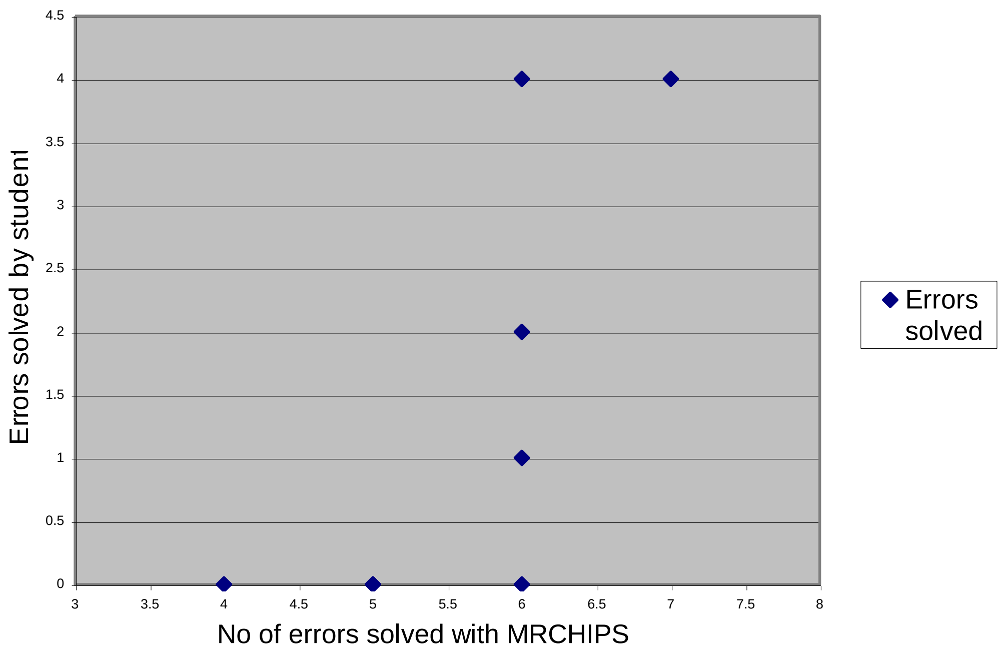


Figure 10.7. Errors solved with MRCHIPS verses errors solved alone

Anecdotal feedback from the mentored participants indicated that they found they were able to follow the help offered by MRCHIPS and that some felt they were not having to correct errors on their own. Although not part of the experiment those from the mentored group commented that they preferred to have the agent speak to them as they read the text of help messages from MRCHIPS.

10.2 Discussion

MRCHIPS is an agent-based solution to the problem of mentoring novice-programming students. The MRCHIPS architecture allows for the reactive and deliberative reasoning required for the agent to operate within a dynamic desktop environment while making diagnostic decisions about programming errors. The Beliefs Desires and Intentions (BDI) based planning system is used to coordinate operations within the agent from responding to inputs, controlling outputs and scheduling the other reasoning resources in the agent. Reactive reasoning is supported in the BDI by maintaining an agenda of goals and selecting an appropriate plan to solve the goal. Deliberative reasoning in the BDI is supported using a series of plans refine the steps of a goal before ultimate solution is found. Most of the agent's deliberative reasoning involved diagnosis of novice errors. Categorising errors into different types lead to their consideration as individual diagnostic situations, which provided a strong correlation to their representation as cases and indicated the use of case-based reasoning (CBR) in the agent. However not all of the deliberative reasoning is processed by the CBR; the diagnosis of syntax errors are processed using a rule-based parser. The syntax for programming languages are usually defined as a series of rules, such as in the BNF notation used in chapter eight in table 8.1. The rule representation therefore naturally lends itself to efficient processing in a rule-based parser. The agent-based solution allows different reasoning methods to be applied to perform different types of problem solving. The MRCHIPS agent met its initial design requirements. It was able to monitor the learner's activity, accurately diagnose the errors and respond to the learner in a timely manner. This was possible even though the architecture is run from an interpreter, PAL, within another interpreter, Python. Much of the speed and accuracy in reasoning is gained via use of the lookup tables, built from the Python hash table data structure, which allow for the fast indexing of data and reduces the need to search. The BDI uses a hash table to reference to the underlying Python functions that implement the PAL interpreter. The CBR also makes use of hash tables to form the discrimination network so indexing is performed via a single lookup for each argument of the case. Operations that are time consuming such as unification and the BNF parser, both of which involve a systematic search through data structures, are used sparingly.

However there are a few limitations in the operation of MRCHIPS, features that it is not able to carry out or that would require some redesign to implement. The limitations examined detail below and are related to:

a) Strategic – how the agent informed the learner, only monitored the learner in some modes and no direct modelling of the learner;
b) Technical – no capability to adapt to unforeseen situations;

c) Social – the limited capacity for natural language interaction;
d) Portability – only available on Windows platforms.

Two flaws discovered during evaluation had to be corrected to before further experimentation could take place. First was the strategy used by MRCHIPS to inform the user that it had a solution to an error. On detecting an error MRCHIPS waits for a period to allow the learner to self correct if possible. In its original configuration MRCHIPS provided no feedback that it had detected the error and appeared only after the delay. Feedback from the learners indicated that this was disconcerting so a semi-transparent pop-up window was introduced to alert the learner that MRCHIPS would provide help after a delay. The second was that the agent only monitored the environment when the MRCHIPS character was minimised. As some learners preferred to work while MRCHIPS was on the desktop they were unable to receive further assistance. It was incorrectly thought that whenever the agent character was on the desktop the learner would be in dialog with it, so they were unable to proceed with writing code. Fortunately the solution required only changes in the plans within the knowledge base to allow scanning of the environment to be performed as a separate intention, effectively running as a thread and irrespective of the state of the agent character.

The MRCHIPS strategy only allows for indirect modelling of the user by modelling the types of programming errors. No attempt is made to directly model the user in the way that a system such as the Genie intelligent assistant, reviewed in chapter 3, is able to do. Modelling the user would involve making an assessment of the user's level of expertise and adjusting the behaviour of MRCHIPS to suit the user's preferences. For example a novice user might prefer help only in the form of the solution to an error, but once more accomplished he or she might prefer a longer explanation to the cause of the error. Modelling the user via programming errors was adequate for experimentation with MRCHIPS but for longer term use direct modelling of the user would allow the agent to track the user's progress, present information in a format that is tuned to the user's ability and allow for the more complex social interactions.

The CBR in MRCHIPS has no capacity to automatically acquire new cases this would have provided MRCHIPS with a form of learning and the capability to adapt to novel or unforeseen situations. There are two areas of the agent architecture able to support learning but they were not required for the evaluation. First the function of the CBR could be extended to implement the adaptation and storage operations for new cases. The BDI plans could be used to guide the adaptation process, which would require the manipulation of the Prolog data structure used to represent the case. The second learning capability is an autobiographical memory, which would allow the agent to consult the record of its experiences for decision-making and reflection. The journaling system already records the decisions of MRCHIPS but the agent makes no further use of the information. Autobiographical memory would allow MRCHIPS to model the record of individual learners and adjust decisions to meet their needs.

MRCHIPS has limited capacity for complex social interactions with the user, which could be used with the diagnosis of logical errors and to offer messages of support and encouragement. No method could be determined to allow MRCHIPS to diagnose the cause of logical errors because the program code would be legal and so the agent would need to understand the programmer's intentions for the code. A solution was designed to have the agent guide the learner through a question and answer process and offer suggestions to allow them to determine the cause but this was not implemented. There are a few plans and cases in the agent's knowledge base that offer messages of encouragement, but these are presented at random. The use of autobiographic memory would allow messages to be tracked and encouragement could then be offered within a strategy.

MRCHIPS is only currently able to run on Microsoft Windows based platforms. The reasoning subsystems the BDI planner, CBR and BNF parser are platform independent but the agent interface subsystem is specific to the operation of the WIN32 programming interface and the animated character relies on the Microsoft Agents engine, which is only available for Windows. Converting the agent interface to work with other GUI systems should be possible if the appropriate operating systems resources, such as system events and messages, are accessible. An alternative to the Microsoft Agent character interface would also be required such as Double Agent or that used in the Adele system reviewed in chapter four.

10.3 Summary

A quantitative evaluation for the effect of MRCHIPS on the work of novice programmers has been given. The data presented in this chapter provides evidence for the effective support of a pedagogical agent for assisting novice programming students as they learn Python as a first language. The results of the experimentation were able to demonstrate that the presence of the agent was able to assist participants to make progress with developing a Python program, not least because MRCHIPS was able to provide answers. Comparing the results from the groups of novice programmers, those working with MRCHIPS were 10% more productive than those working with no agent. From the t-test the calculated value t = -2.505 was found to exceed the critical value of t = $\pm$ 2.110 at p = 0.05 with a df = 17. Therefore the null hypothesis is rejected and it is concluded the mean score for the mentored novice students (65.5%) was significantly higher than for the un-mentored novice students (55.5%). Analysis of the experimental findings show there to be a significant correlation between the presence of MRCHIPS and the improvement in performance of the novice programmers. There was a positive correlation coefficient of 0.73 between the support offered by MRCHIPS and the grade achieved by the mentored students. There were also indications of learning where subjects were able to recognise and solve problems without the guidance of the agent, although the correlation coefficient of 0.21 was less significant. The evaluation was also able to show some support for learning in that four of the mentored students were able to recognise and solve one or more of the repeated errors without the aid of the agent. The mentored student who was able to solve the logical error even though MRCHIPS had no support suggested he recalled some knowledge from an earlier programming experience and replied, "It just seemed to be the way it worked." The major caveat with the results is that the size of the study group was small and the study was short in duration. Therefore the effect of an individual's performance on the reading would have a disproportionate effect on the findings. It had originally been planned to then run a larger study over a longer learning period. Unfortunately due to a change in employment that required a move away from the university contact with the student study group was lost.

Chapter 11:

Conclusions and future work

This chapter summarises the aim of this research, its findings and proposes future work. In the following sections a discussion is given on the extent to which the research and objectives were achieved, a critical reflection on the research conducted, followed by a summary of the original contributions of the research, and finally ideas are presented for future work.

11.1 Review of research objectives

In this research it was proposed that a cognitive agent powering an animated virtual character could provide effective support for novice programmers as they learnt their first programming language in a desktop environment. To investigate the hypothesis the framework of March & Smith, and Järvinen was used to research four complementary questions:

Hypothesis 1) An intelligent agent with an anthropomorphic interface can provide effective mentoring support to novice programmers learning their first programming language.

This hypothesis can be answered with a measured degree of certainty. There was a strong correlation found between the mentoring presence of the MRCHIPS agent and the higher performance for the novice programming students. From the t-test the calculated value t = -2.505 was found to exceed the critical value of t = + 2.110 at p = 0.05 with a df = 17. The p value was found be 0.016, which corresponds to a 1.6% chance of rejecting a correct hypothesis. The mean score for the mentored novice students of 65.5% was higher than for the un-mentored novice students of 55.5%. Learners that worked with MRCHIPS scored on average 10% higher than beginner programmers without the agent. Results from the evaluation study therefore show that the presence of MRCHIPS made a positive improvement in the performance of novice programmers. This difference is more significant as the non-mentored beginner programmers had had about 6 weeks of Python study at the time of their test where the mentored group had no Python exposure before the test. The mentored students who followed the advice given by MRCHIPS were able to correct more of the errors; there was a positive correlation coefficient of 0.73 between the support offered by MRCHIPS and the grade achieved by the mentored students. Of the mentored group four of the nine subjects were able to solve one or more of the repeated errors without the aid of MRCHIPS. These were interpreted as indications of learning, with a correlation coefficient of 0.21.

The MRCHIPS cognitive architecture was able to provide positive answers for a reasoning solution for the domain. Although this research was able to show the increase in productivity, some learning of syntax and signs for an increase of engagement from the learner, it was not able to show a similar effectiveness for logical errors. However the size of the study was small and of a short duration, so even with the use of the control groups the findings should be read as an indication of the agent's possibility. Researchers using other teaching virtual agents such as Steve (Rickel & Johnson 1998) and FatiMA (Aylett et al. 2007) reported comparable improvements in the performance of learners as found with MRCHIPS. The literature also reported that programmers improved their performance with intelligent tutoring systems such as UNCLE (Wang & Bonk 2001) and CABLE (Chen et al. 2006) although the systems would not be suitable for novice learners.

Hypothesis 2) The use of an animated virtual character user interface increases the learner's engagement with problem solving in the programming environment.

The engagement of the user is probably the least evaluated part of the hypothesis due to the choice to bias data gathering to a more quantitative method. However feedback from the subjects was positive about the agent with the learners reporting that they found MRCHIPS helpful even for those who were unable to substantially complete the exercise. There was a strong positive correlation coefficient between the activity of MRCHIPS and the progress of the mentored learners. Although no tests were made of the mentored learners preference for the degree of embodiment MRCHIPS has the capability of using different anthropomorphic characters to produce this effect. Feedback from users expressed a preference for more natural forms of communication such as having MRCHIPS speak the help messages. A positive response to the agent is consistent with the *persona effect* (Lester *et al.* 1999) reviewed in chapter 4 where participants reported a preference for the presence of an anthropomorphic character and demonstrated improvements in cognitive tests when working with an animated agent interface (Krämer 2005). There is the caveat that it may be the novelty of an intelligent virtual agent. It remains unclear whether the positive response was as a result of the help provided by MRCHIPS or the novelty of the animated character. It is possible that long-term use of MRCHIPS could elicit similar levels of irritation by its sister product the Microsoft office paper clip. However as the MRCHIPS reasoning is context sensitive and attempts to fade support with the level of user competence the chance of alienating the user may be reduced.

Hypothesis 3) The processing capabilities of a procedural BDI agent can be extended to provide the more knowledge based reasoning capabilities of a cognitive agent architecture.

This question was answered by the construction of the MRCHIPS agent. The MRCHIPS architecture follows Langley's four commitments for the development of cognitive agents architecture (1991) explained in section 5.4 and an explanation of how MRCHIPS implements the commitments is given in section 8.2. At the core of MRCHIPS is the BDI planner, the CBR for diagnosis, the BNF parser and the agent interface subsystem. Both the BDI and CBR provide methods for providing different kinds of reasoning based on theoretical models of cognition. Sharing reasoning across the different subsystems in the agent architecture allows each to contribute by providing reasoning for where it is best suited. So the BDI planner provides goal seeking and procedural control and the CBR provides domain specific diagnostics. The BNF parser became a necessary addition when it was found the CBR would be inefficient for reasoning about syntax errors. The design of the agent architecture allows the activity of all subsystems to be coordinated by the BDI including the CBR. The reason for this is to allow the agent architecture to be adaptable in its operation allowing the plans in the knowledge base to determine the reasoning resources in use.

Hypothesis 4) Agent based reasoning provides a framework to extend knowledge-based systems into existing computing desktop environments and to avoid the need to build a specialised learning application environment.

The domain knowledge of learner errors is contained in the CBR and BNF knowledge bases. For conventional knowledge based systems the user would consult the application presenting the properties of the problem and await diagnosis. For the novice programmer to have to consult the knowledge base involves increasing his or her cognitive load, as they would have to learn how to use the application and decide when to use it. Using an intelligent virtual agent to monitor the learner in the environment and decide when to consult diagnostic resources allows the knowledge-based reasoning to be available to the learner. In order for the agent to operate within the Windows environment required the application of various programming techniques to allow the agent to monitor the learner by assembling information from different parts of the operating system and the Python development environment. The automation routines of the Win32 API allow MRCHIPS to access information about the activity on the Windows desktop. Unfortunately the Python development environment is built on top of the Tkinter library, which has limited support for the automation facilities, preventing MCHIPS from cleanly performing a copy of the content of the Python editor window. This made it necessary for one change to the environment as MRCHIPS adds a menu item called *clear* to the Python editor window upon installation. This is the only change that MRCHIPS requires to the environment.

11.2 Critical reflection

There were various challenges faced in undertaking this research, the discussion below outlines some of the factors that influenced the options and the decisions taken.

- The MRCHIPS agent was devised to provide mentoring support for novice programmers within the framework of the cognitive apprenticeship pedagogy. Cognitive apprenticeship has a number of features that made it an attractive choice for use in this research. First the pedagogy correlates to the practice used in mentoring, most notably the coaching and scaffolding methods. The exploration method would also be provided by the availability of a mentoring resource to support the learner when experimenting with the programming language. Second the pedagogy provides a structured framework with separate methods, where the aims and outcomes of each method may be considered in isolation and easily measured if required. Third, the methods of the pedagogy may be implemented in different ways, such as by exercise, reading material, discussion, etc. This flexibility allows the possible use of a technological solution where the details of activity may be different, but aims and outcomes are used to determine how the activity contributes.

- The development of the architecture went through many iterations of design, mainly due to attempts to integrate a CBR engine based on the MOPS data structure (Riesbeck & Schank 1989) with the BDI engine based on the Prolog Horn clause. During the development of the agent no method could be found for integrating the Horn clause with the MOPS data structure that would not cause a loss of data or become time consuming when converting of data was to be passed back and forth between subsystems. Once it was decided to base the cases on the same Horn clause data structure and use a discrimination network to control storage and retrieval the development progressed quickly. Using a single knowledge representation scheme the different reasoning subsystems simplified communication. Concepts that mean the same thing have the same representation in the knowledge base even though they are processed in different ways by different subsystems. The single representation also allows for some agent resources to be shared such as the Prolog language parser, which is used by all subsystems to read the agent knowledge base and the unifier used for matching data.

- Although the MRCHIPS was designed to provide mentoring in a desktop environment the architecture was designed to follow the principles of a cognitive architecture. The reason for this was to allow for the likely range of reasoning requirements within the desktop environment. The MRCHIPS architecture satisfies nearly all of the commitments for a cognitive architecture as described by Langley (2006) and discussed in chapter five; the commitment to long-term memories is currently underdeveloped; it would be addressed by the ability to retain new cases in the CBR or the inclusion of an autobiographical memory similar to that used in agents like FatiMA (Aylett *et al.* 2007). It is likely that other cognitive agent architectures such as Soar (Laird *et al.* 1987), Icarus (Langley *et al.* 1991), or ACT-R (Anderson 1993) would also be suitable frameworks on which to build MRCHIPS. The decision was taken to build MRCHIPS in Python for two reasons. First to gain an insight into how to implement a cognitive architecture. Secondly in addition to its suitability for teaching the properties of Python make it an attractive choice for prototype application development as would be required for this research. In addition implementing the agent in the same language as would be used by the learner would simplify its installation process. As MRCHIPS is simply a Python application all the resources required for its execution would be available once Python was installed. It was imagined that student volunteers would install MRCHIPS on their own computers without supervision so the installation process was made as simple as possible.

- Although the Microsoft agent character interface is integral to the way MRCHIPS operates no experimentation was attempted on changes to the interface. Work had been carried out to provide a dialog text box to handle inputs to the agent. The Microsoft agent engine only allows speech input and as the presence or quality of a speech input engine was unknown for the computer on which students might use MRCHIPS a dialog box was added. Consideration was given to assessing the effect of the degree of embodiment and animation on learning but this was not pursued as research elsewhere had been carried out to investigate this (Lusk & Atkinson 2007). It is also worth noting that Microsoft has withdrawn support for MS-agents on operating system versions after Windows XP; an open-source alternative application called Double Agent from Cinnamon Software Inc. is free to download from the Internet, it is designed to be fully compatible with MS-agents and available for more recent versions of Windows but at this time no evaluation has carried out to its use with MRCHIPS. A significant effort had been made to supply MRCHIPS with a natural language parser but no solution could be developed that supported a large enough vocabulary, that could process statements rapidly enough, and would remain stable enough to be used for the experimentation. What had not been anticipated was how important the text-to-speech feature was to engagement with the agent, with learners commenting that they preferred to have the agent speak to them as they read the text of the help message from MRCHIPS.

- The amount of experimentation with the agent was only enough to establish that MRCHIPS had a positive effect on the outcome for learners in a task requiring coaching support. There were also good indications of scaffolding, as some learners did not use

MRCHIPS once they had recognised the reoccurrence of errors and applied a remembered correction. Due to a change of employment there was no opportunity to test whether MRCHIPS had an effect on learner exploration. Ideally a larger evaluation would be carried out taking place over several months, involving numbers of students comparable to the cohort size and including a similar sized randomly selected control group with access to similar resources working to a similar lesson plan, but in the absence of the agent. At the end of the trial students of both groups would be tested on what they had learned. Given that both sets of students had access to similar resources any difference in the outcome of their results could be then attributed to the presence of the agent. However, even under ideal experimental conditions other factors would still be present that would influence or cause to question the outcome. For instance as people partake in any process their experience tends to grow. It would not be unreasonable to expect learners to become more proficient programmers with or without an agent assistant leading to the conclusion that there is no significant measurable difference after a sufficient period of time. Therefore, in order to demonstrate the usefulness of MRCHIPS, it was necessary to show that novice programmers' were able to make more progress in practical exercises as a result of the agent than they would without it. However a larger evaluation of the agent is still required.

11.3 Research contributions

The principle contribution of this research is in demonstrating how an agent system may be used to provide mentoring support to learners working with conventional development tools and in a conventional desktop environment. This approach allows learner practice to occur within the same environment as used by experienced programmers, a strategy that adheres to one of the major principles of the cognitive apprenticeship pedagogy, that of using knowledge in a real world context (or as close to as possible). It differs from other intelligent tutoring systems that use specialized learning environments. Using an agent-based approach allows the expertise in the knowledge base to be brought to where the learner has to work and avoids increasing the student's cognitive load of having to learn how to use the learning environment in order to use the working environment. The second contribution is the development of a novel agent architecture that is able to utilise different reasoning capabilities to provide the mentoring support. This is achieved by combining a BDI planner with a CBR reasoning engine in a unique architecture to address the processing requirements to monitor the environment, control a user interface via an interactive anthropomorphic animated character and to make the knowledge base available to diagnose errors within the learner's program code.

11.4 Future work

There are a number of ways in which the MRCHIPS architecture may be improved. The completion of the natural language parser for a question answer system would allow MRCHIPS to be consulted to help solve logical errors. The simplest method to add this to the architecture would be to have questions to the agent form some intermediate data structure that could be used as a problem to the CBR. The selected solution case would then contain the response or activity required to provide an answer.

A more interesting challenge would be to redesign the journaling system to provide autobiographic episodic memory for the agent. This would involve implementing journaling memory as a consultable knowledge structure and allow the agent to be able to recall events from interaction with the learner and possibly provide a richer set of interactions with the learner "This problem is similar to …" or "Do you remember the …". The use of autobiographical memory would be one way to provide the commitment to a long-term memory system, required by cognitive agent architecture, for MRCHIPS. Two methods would be available to allow the agent to analyse and reflect on events. First, in the selection of BDI plans the process may be refined by specifying the past events that need to have occurred in conditions of plans. Secondly, sequences of episodic memory could be used to index the CBR and the resultant case used to specify what activity should then be performed by the agent.

The only development environment currently supported by MRCHIPS is the Tkinter based environment that is shipped with the Python installation. However because it is based on the TCL/TK toolset it works differently from applications developed using the Win32 environment such as the development environment provided by the PyWin32 library. The MRCHIPS agent could be extended to work with different development environments such as the Win32 based IDE that are is installed with PyWin32 or applications like Notepad++. The MRCHIPS knowledge base could be extended to recognise which development environment the learner was using and adjust its operation to cope with the configuration of the tools.

The MRCHIPS architecture was designed to allow adaptation for the mentoring of learners in other programming domains as diverse as Java, CLIPS, Prolog or SQL. MRCHIPS was originally planned with a programming domain for Visual Basic 6 but this was redesigned when curriculum for the learners was changed to use Python. Support for Java might provide a better illustration of the effectiveness of MRCHIPS as the Java syntax makes fewer, if any concessions to learners but for some may still be the first programming language that they will be taught. The adaptation would require analysis of the errors in the language and the development environment in question. Then changes would be required to the monitor module, the BNF parser for the language and the case base in order to provide support. The CLIPS and Prolog languages provide alternate programming paradigms and related syntax differences to those of conventional procedural languages as a challenge for the agent to provide help.

References

Aamodt, A. & Plaza, E. (1994) Case-Based Reasoning: Foundational Issues, Methodological Variations and System Approaches. *AI Communications* 7(1): 39-59.

Ancona, D., Demergasso, D. & Mascardi, V. (2005). A Survey on Languages for Programming BDI-style Agents. [Online.] Available from: http://www.cs.uu.nl/~mehdi/al3files/VivianaMascardi.ps [Accessed: 21st January 2010].

Anderson, J.R. (1989) Practice, Working Memory and the ACT Theory of Skill Acquisition: A Comment on Carlson, Sullivan and Schneider. *Journal of Experimental Psychology, Learning, Memory and Cognition* 15: 527-530.

Andre, E. (1999) Believable Agent Deixis. *Proceedings of the Workshop on Deixis, Demonstration and Deictic Belief*. Eleventh European Summer School in Logic, Language and Information ESSLLI XI. pp. 30-42.

Arbor, A. (1999) *How to mentor graduate students: A guide faculty at a diverse university.* University of Michigan

Aylett, R.S., Louchart, S., Dias, J., Paiva, A. & Vala, M. (2005) Fear Not! An Experiment in Emergent Narrative. In: Panayiotopoulos, T., Gratch, J. Aylett, R. Ballin, D., Olivier, P. & Rist, T. (eds.) *Intelligent Virtual Agents: Proceedings of the 5th International Working Conference, IVA 2005*, Kos, Greece, September. Berlin: Springer. pp. 305-316.

Aylett, R.S., Vala, M., Sequeira, P. & Paiva, A. (2007) FearNot! – An Emergent Narrative Approach to Virtual Dramas for Anti-bullying Education. *Proceedings of the International Conference on Virtual Storytelling (ICVS), 5-7 December, St Malo, France.* Berlin: Springer-Verlag. pp. 199-202.

Baillie De-Byl, P. (2004) *Programming Believable Characters for Computer Games*. Higham, MA: Charles River Media.

Bajo, J. & Corchado, J.M. (2005) Evaluation and Monitoring of Air-Sea Interaction Using a CBR-Agents Approach. In: Muñoz-Avila, H. & Ricci. F. (Eds.) *Case-based reasoning research and development: 6th International Conference on Case-Based Reasoning,* ICCBR 2005. Chicago: Springer. pp 50-62

Booch, G. (1993) *Object oriented analysis and design with applications*. 2nd Ed. Boston: Addison Wesley.

Boyle, T. (2001) Constructivism in Computer Science Education. Paper presented at Middlesex University, 2 July 2001.

Bratman, M. (1987) *Intention, Plans and Practical Reason*. Cambridge, MA: Harvard University Press.

Brooks, R.A. (1991) Intelligence without Representation. *Artificial Intelligence* 47: 139-159.

Brunning, S. & Couper, T. (2003) WinGuiAuto Source Code. [Online]. Available:
http://www.brunningonline.net/simon/blog/archives/winGuiAuto.py.html
[Accessed: 2005].

Case, D. (2000) Is Python a suitable tool for developing Artificial Intelligence Applications? MSc thesis. Milton Keynes: The Open University.

Cassell, J., Bickmore, T., Billinghurst, M., Campbell, L., Chang, K., Vilhjalmsson, H., & Yan, H. (1999) Embodiment in Conversational Interface: Rea. *The CHI is the Limit: Proceedings of the CHI '99*

Conference on Human Factors in Computing Systems, 15-20 May, Pittsburgh, PA. New York: ACM Press. pp. 520-527.

Chang, K.E., Sung, Y.T. & Chen, S.F. (2001) Learning through computer-based concept mapping with scaffolding aid, *Journal of Computer Assisted Learning* 17: 21-33.

Chee, Y.S. (1994) SMALLTALKER: A Cognitive Apprenticeship Multimedia Environment for Learning Smalltalk Programming. *Proceedings of the ED-MEDIA 94 World conference on Educational Multimedia and Hypermedia*. Charolottesville, PA: Association for the Advancement of Computing in Education. pp. 492-497.

Chee, Y.S. & Xu, S. (1997) SIPLeS: Supporting Intermediate Smalltalk Programming through Goal-based Learning Scenarios. *Proceedings of the AI-ED 8th World Conference on Artificial Intelligence in Education,* Kobe, Japan. Amsterdam: IOS Press. pp. 95-102.

Chan Mow, I, Au, W. & Yates, GCR (2006). The impact of CABLE on teaching computer programming. *Fourteenth International Conference on Computers in Education (ICCE).* Beijing, China, November.

Charniak, E., Riesbeck, C. K., McDermott, D. V. & Meehan, J. R. (1987) *Artificial Intelligence Programming*, 2nd Ed. Hillsdale, New Jersey: Lawrence Erlbaum Association.

Chen, F. (2003) *Agent-oriented Fault Detection, Isolation and Recovery And Aspect-Oriented Plug and Play Tracking Mechanism*. Masters Dissertation. Texas A & M University.

Chi, R.T.H. & Kiang, M.Y. (1991) An Integrated Approach of Rule-Based and Case-Based Reasoning for Decision Support. *Association for Computing Machinery* 3(1): 255-267.

Clancey, W.J. (1992) Representations of Knowing: In Defence of Cognitive apprenticeship. *Journal of Artificial Intelligence in Education* 3(2): 139-168.

Collins, A., Brown, J.S. & Newman, S.E. (1989) Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In: Resnick, L.B. (ed.) *Knowing, learning and instruction: Essays in honor of Robert Glaser*. Hillsdale, NJ: Erlbaum. pp. 453-494.

Collins, A., Brown, J.S. & Holum, A. (1991) Cognitive apprenticeship: Making Thinking Visible. *American Educator* (Winter).

The Collins Dictionary and Thesaurus (1989) *William Collins Sons & Co Ltd.*

Conway, M.J. (1997) Alice: Easy-to-Learn 3D Scripting for Novices. PhD Thesis. Charlottesville, VA: University of Virginia.

Coolidge, F.L. (2000). *Statistics A Gentle Introduction*. SAGE Publications.

Corchado, J. M. & Pellicer, M. A. (2005) Development of CBR-BDI Agents. *International Journal of Computer Science & Applications* 2(1): 25-32.

Davies, S.P. (1991) The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behaviour. *Cognitive Science, 15*: 547–572.

d'Inverno, M. & Luck, M. (1998) Engineering Agentspeak(L): A formal computational model. *Journal of Logic and Computation* 8(3).

Driscoll, D. L., Appiah-Yeboah, A., Salib, P. & Rupert, D. J. (2007) Merging Qualitative and Quantitative Data in Mixed Methods Research:

How To and Why Not. *Ecological and Environmental Anthropology.* Vol. 3, No. 1. pp. 19-28

Evers, M. & Nijholt, A. (2000) Jacob: An Animated Instruction Agent. In: *Advances on Multimodal Interfaces ICMI 2000*. Berlin: Springer. pp. 526-533.

Ferguson, I.A. (1992) TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents. *Ph.D. thesis, Computer Laboratory, University of Cambridge*, Cambridge UK.

Florian, R.V. (2003) Biologically inspired neural networks for the control of embodied agents. *Technical Report Coneural-03-03.* Available from: http://www.coneural.org/reports/Coneural-03-03.pdf.

Freund, J.E. & Simon, G.A. (1996) *Modern Elementary Statistics.* 9th Ed. Prentice Hall

Ghefaili A. (2003) Cognitive Apprenticeship, Technology, and the Contextualization of Learning Environments. *Journal of Educational Computing, Design and Online Learning* 4 (Fall).

Gilmore, D.J. (1990) Expert Programming Knowledge: A Strategic Approach. In: Hoc, J.M., Green, T.R.G., Samurcay, R. & Gilmore, D.J. (eds.) *Psychology of Programming (Computers and People)*. Academic Press. pp. 223-234.

Glezou, K. & Grigoriadou, M. (2007) A novel didactical approach of the decision structure for novice programmers. *Proceedings of Eurologo 2007 Conference,* Bratislava

Gobil, A.R.M., Shukor, Z. & Mohtarl, I.A. (2009) Novice Difficulties in Selection Structure. *2009 International Conference on Electrical Engineering and Informatics.* Selangor, Malaysia: 351-356

Gonzalez, A.J. & Dankel, D.D. (1993) *The Engineering of Knowledge-Based Systems: Theory and Practice.* Englewood Cliffs, NJ: Prentice Hall.

Gray, W.D., Corbett, A.T., & VanLehn, K. (1988) Planning and Implementation Errors in Algorithm Design. *Eleventh Annual Conference of the Cognitive Science Society*. Montreal, Canada, pp. 594-600

Green, T.R.G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge, UK: Cambridge University Press, pp 443-460.

Gulz, A. (2004) Benefits of virtual characters in computer based learning environments: claims and evidence. *International Journal of Artificial Intelligence in Education* 14: 313-334.

Harel, I. & Papert, S. (1991) Constructivism. Norwood, NJ: Ablex Publishing.

Hoc, J.M., Green, T.R.G., Samurcay, R. & Gilmore, D.J. (eds.) *Psychology of Programming (Computers and People)*. Academic Press.

Hopgood, A.A. (2001) *Intelligent Systems for Engineers*. 2nd Ed. Baco Raton, FL: CRC Press.

Jackson, P. (1999) *Introduction to Expert Systems*. 3rd Ed. Reading, MA: Addison-Wesley.

Järvinen, P. (2004) *Research Questions Guiding Selection of an Appropriate Research Method*. Tampere: University of Tempere.

Johnson, W.L., Shaw, E. & Ganashan, R. (1999) Pedagogical Agents on the Web. In: Etzioni, O., Müller, J.P. & Bradshaw, J.M. (eds.) *Proceedings of the Third Annual Conference on Autonomous Agents*, Seattle, WA. New York: ACM Press. pp. 283-290.

Johnson W.B. (2007) *On Being A Mentor: A Guide for Higher Education Faculty.* Lawrence Erlbaum Associates, inc.

Jadud, M.C. (2004) A first look at novice compilation behavior using BlueJ. *Sixteenth Workshop of the Psychology of Programming Interest Group*

Kieras, D. & Mayer, D.E. (1997) An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction* 12: 391-438.

Koedinger K.R. (2001) Cognitive Tutors as Modelling Tools and Instructional Models. In: Forbus, K.D. & Feltovich, P.J. (eds.) *Smart Machines in Education.* Cambridge, MA: MIT Press. pp. 145-167.

Kopp, K., Gesellensetter, L., Krämer, N.C. & Wachsmuth. I. (2005) A Conversational Agent as Museum Guide – Design and Evaluation of a Real-World Application. In: Panayiotopoulos, T., Gratch, J. Aylett, R. Ballin, D., Olivier, P. & Rist, T. (eds.) *Intelligent Virtual Agents: Proceedings of the 5th International Working Conference, IVA 2005*, Kos, Greece, September. Berlin: Springer. pp. 329-343.

Krämer, N.C. (2005) Social Communicative Effects of a Virtual Program Guide. In: Panayiotopoulos, T., Gratch, J. Aylett, R. Ballin, D., Olivier, P. & Rist, T. (eds.) *Intelligent Virtual Agents: Proceedings of the 5th International Working Conference, IVA 2005*, Kos, Greece, September. Berlin: Springer. pp. 442-453.

Kummerfeld, S.K. & Kay, J (2006) The neglected battle fields of Syntax Errors. *Proceedings of the fifth Australasian conference on Computing education.* Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 105–111

Laffey, J., Tupper, T., Musser, D., & Wedman, J. (1998) A Computer-Mediated Support System for Project-Based Learning. *Education Technology Research and Development* 46(1): 73-86.

Lahtinen, E., Ala-Mutka, K., & Jarvinen, H.M. (2005) A Study of the Difficulties of Novice Programmers. *Innovation and Technology in Computer Science Education '05*: June 27-29.

Laird, J., Newell, A. Rosenbloom, P. (1987) SOAR: An architecture for general intelligence. *Artificial Intelligence* 33(1 September): 1-64.

Landsberg, M. (1996) *The Tao of Coaching.* London, Harper Collins

Langley, P. (2006) Cognitive Architectures and General Intelligent Systems. *AI Magazine* 27(2): 33-44.

Lee, J., Huber, M.J., Durfee, E.H. & Kenny, P.G. (1994) UMPRS: An implementation of the procedural reasoning system for multirobotic applications. In: *Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, Service and Space*. pp. 842-849.

Lesgold, A., Lajoie, S., Bunzo, M., & Eggan, G. (1992) SHERLOCK: A coached practice environment for an electronics troubleshooting job. *Computer-assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches*. Hillsdale. NJ: Erlbaum. pp. 201-238.

Lester, J., Voerman, J., Towns, S. & Callaway, C. (1999) Deictic Believability: Coodinating Gesture, Locomotion, and Speech in Lifelike Pedagogical Agents. *Applied Artificial Intelligence* 13(4-5): 383-414.

Leutner, D. (2000) Double-fading support – a training approach to complex software systems, *Journal of Computer Assisted Learning* 16: 347-357.

Lui, K.M. & Chan, K.C.C. (2006) Pair programming productivity: novice-novice vs expert-expert. *International Journal of Human-Computer Studies* 64(9): 915-925.

Lusk, M.M. & Atkinson, R.K. (2007) Animated pedagogical agents: Does their degree of embodiment impact learning from static and animated worked examples? *Applied Cognitive Psychology*. 21: 1-18.

Lutz, M. (2001). *Programming Python*, 2nd Ed. Sebastopol, CA: O'Reilly.

March, S.T. & Smith, G.F. (1995) Designs and Natural science Research on Information Technology. *Decision Support Systems* 15(4): 251-266.

Masden, B. (2011). *Statistics for Non-Statisticians*. 1st Ed. Berlin: Springer.

Mateas, M. (1997) *An Oz-Centric Review of Interactive Drama and Believable Agents*. Working Paper CMU-CS-97-156. Pittsburgh, PA: Carnegie Mellon University.

Mayer, R.E. (1981) The Psychology of How Novices Learn Computer Programming. *Association of Computing Machinery, Computing Surveys* 13(1): 121-141.

McBreen, H., Anderson, J. & Jack, M. (2001) Evaluating 3D Embodied Conversational Agents in Contrasting VRML Retail Applications. In: *Proceedings of the International Conference on Autonomous Agents, Workshop on Multimodal Communication and Context in Embodied Agents,* Montreal, Canada. pp. 83-87.

McIver, L. & Conway, D. (1996) Seven Deadly Sins of Introductory Programming Language Design. *Proceedings, Software Engineering: Education & Practice 1993* (SE:E&P'96),309-316.

Millington, I. (2006) *Artificial Intelligence for Games*. San Fransisco: Morgan Kaufmann.

Morley, D. & Myers, K. (2004) The SPARK Agent Framework. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems Volume 2*. 19-23 July. New York: ACM Press. pp. 714-721.

Moskal, B., Lurie, D., & Cooper, S. (2004) Evaluating the Effectiveness of a New Instructional Approach. *Proceeding of SIGCS04, Norfolk, Virginia.* New York: ACM. pp. 75-79.

Müller, J.P. (1991) *The Design of Intelligent Agents: A Layered Approach*. Berlin: Springer-Verlag.

Nilsson, N. (1994) Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research* 1:139-158

Nuxoll, A. & Laird, J.E. (2004). A Cognitive Model of Episodic Memory Integrated with a General Cognitive Architecture. *Proceedings of the International Conference On Cognitive Modelling (ICCM2004),* 30 July –1 August, Pittsburgh, Pennsylvania, USA. pp. 220-225. [Online] Available from:http://www.informatik.unitrier.de/~ley/db/conf/iccm/iccm2004.html. [Accessed 21st January 2010].

Padgham, L. & Winikoff, M. (2004) *Developing Intelligent Agent Systems: A Practical Guide*. Chichester, West Sussex: John Wiley & Sons Ltd.

Pane, J.F. & Myers, B.A. (1996) Usability Issues in the Design of Novice Programming Systems. *Human-Computer Interaction.* Institute Technical Report CMU-HCII-96-101.

Pea, R.D. (1986) Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2(1): 25-36.

Pennington, N. & Grabowski, B. (1990) The Tasks of Programming. Hoc, J.M., Green, T.R.G., Samurcay, R. & Gilmore, D.J. (eds.) *Psychology of Programming (Computers and People)*. Academic Press.

Perkins, D.N. & Martin, F. (1986) Fragile Knowledge and Neglected Strategies in Novice Programmers. In: Soloway, E. & Iyengar, S. (eds.) *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing. pp. 213-229.

Rao, A.S. & Georgeff, M. P. (1995) BDI Agents: From Theory to Practice. In: Lesser, V. and Gasser, L. (eds.) *Proceedings of the first international conference on multi-agent systems (icmas-95),* 12-14 June, San Fransisco, CA, USA. Cambridge, MA: MIT Press, pp. 312-319.

Rickel J. & Johnson, W.L. (1998) STEVE: A pedagogical agent for virtual reality. In: Sycara, K.P. & Woolridge, M. (eds.) *Proceedings of the 2nd International Conference on Autonomous Agents*, Minneapolis, 10-13 May. New York: ACM Press. pp. 332-333.

Rickel, J. Johnson, W.L. (1999) Animated Agents for Procedural Training in Virtual Reality: Perceptions, Cognition and Motor Control. *Applied Artificial Intelligence* 13(4): 343-382.

Rickenberg, R. & Reeves, B. (2000) The effect of animated characters on anxiety, task performance and evaluations of user interfaces. *Proceedings of CHI 2000 Conference on Human Factors in Computing Systems*, New York. pp. 49-56.

Riesbeck, C.K. & Schank, R.C. (1989) *Inside Case-Based Reasoning*. Mahwah, NJ: Lawrence Erlbaum.

Rogalsk, J. & Samurcay, R. (1990) Acquisition of Programming Knowledge and Skills. In: Hoc, J.M., Green, T.R.G., Samurcay, R. &

Gilmore, D.J. (eds.) *Psychology of Programming (Computers and People)*. Academic Press.

Russell, S. & Norvig, P. (1995) *Artificial Intelligence: A Modern Approach*. London: Prentice Hall.

Schalkoff, R.J. (2011) *Intelligent Systems: Principles, Paradigms, and Pragmatics*. Jones and Bartlett Publishers.

Shaw, E., Johnson, W.L., & Ganeshan, R. (1999) Pedagogical agents on the Web. In: Etzioni, O., Müller, J.P. & Bradshaw, J.M. (eds.) *Proceedings of the Third Annual Conference on Autonomous Agents*, Seattle, WA. New York: ACM Press. pp. 283-290.

Sproull, L., Subramani, M., Kiesler, S., Walker, J.H., Waters, K. When the interface is a face. *Human-Computer Interaction* 11(2 June): 97-124.

Stanley, K.O., Bryant, B.D. & Miikkulainen, R. (2005) Evolving Neural Network Agents in the NERO Video Game. In: *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*. Piscataway, NJ: IEEE. pp. 298-303.

Stewart, T.C. & West, R.L. (2006) Deconstructing ACT-R. In: *Proceedings of the Seventh International Conference on Cognitive Modelling*. Trieste, Italy. pp. 298-303.

Stewart, T.C & West, R.L. (2007) Cognitive Redeployment in ACT-R: Salience, Vision and Memory. In: *Proceedings of the ICCM Eighth International Conference on Cognitive Modelling*. Oxford: Taylor and Francis. pp. 313-318.

Sun, R., Merrill, E. & Peterson, T. (2001) From implicit skills to explicit knowledge: a bottom up model of skill learning. *Cognitive Science* 25(2): 203-244.

Tanimoto, S. (1990) *The Elements of Artificial Intelligence Using Common LISP*. 2nd. Ed. New York: Computer Science Press.

Tholander J. & Karlgren K. (2002) Support for Cognitive Apprenticeship in Object-Oriented Model Construction. *Computer-Supported Collaborative Learning,* Boulder, Colorado.

Thompson, S.M. (2006) An Exploratory Study of Novice Programming Experiences and Errors. Masters Thesis University of Victoria

Traynor, D. & Gibson, J. P. (2004) Towards the development of a cognitive model of programming; A software engineering proposal. *Sixteenth Workshop of the Psychology of Programming Interest Group*

Tulving, E. (2002) Episodic Memory: From Mind to Brain. *Annual Review of Psychology* 53: 1-25.

Ueno, H. (1998) A Generalized Knowledge-Based Approach to Comprehend Pascal and C Programs. *Frontiers in AI and Applications,* Vol.48, Ios Press

Veloso, M.M. (1994) Prodigy/Analogy: Analogical reasoning in general problem solving. In: Wess, S., Althoff, K.D. & Richter, M. (eds.) *Topics in Case Based Reasoning*. Berlin: Springer-Verlag. pp. 33-50.

Veloso, M.M, & Rizzo, P. (1998) Mapping planning actions and partially-ordered plans into execution knowledge. In: Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments, R. Bergmann and A. Kott, (eds.) *AAAI Press,* Menlo Park, CA, pp. 94-97.

Vere, S. & Bickmore, T. (1990) A Basic Agent. *Computational Intelligence* 6(1): 41-60.

Vilhjalmsson, H. & Cassell, J. (1998) BodyChat: Autonomous Communicative Behaviours in Avatars. In: Sycara, K.P. & Woolridge, M. (eds.) *Proceedings of the Second International Conference on Autonomous Agents,* 10-13 May, Minneapolis. New York: ACM Press. pp. 269-276.

Wang F.K., Bonk C.J. (2001). A Design Framework For Electronic Cognitive Apprenticeship. *Journal of Asynchronous Learning Networks (JALN)* 5 (2 September). pp 131-151

Wiemer-Hastings, P., Graesser, A.C., Harter, D., & the Tutoring Research Group (1998) The foundations and architecture of AutoTutor. *Proceedings of the 4th International Conference on Intelligent Tutoring Systems*. San Antonio, TX. Berlin: Springer-Verlag. pp. 334-343.

Williams, C. (2007) *Research Methods*. Journal of Business & Economic Research. Volume 5, Number 3.

Winikoff, M. (2006) *An AgentSpeak Meta-Interpreter and its Applications*. Berlin: Springer.

Wolz, U. (1988) Automated Consulting for Extending User Expertise in Interactive Environments: A Task Centred Approach. Columbia Univ. Department of Computer Science, Tech. Rep. CUCS-393-88.

Wolz, U. & Kaiser, G.E. (1988) A Discourse-Based Consultant for Interactive Environments. *4th IEEE Conf. On Artificial Intelligence Applications,* pp. 28-33.

Wooldridge, M. (2002) *An Introduction to MultiAgent Systems*. Chichester, West Sussex: John Wiley & Sons Ltd.

Wooldridge, M. & Jennings, N.R. (1995) Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review* 10(2): 115-152.

Woolf, B.P., Beck, J., Eliot, C., Stern, M. (2001) Growth and Maturity of Intelligent Tutoring Systems: A Status Report. In: Forbus, K.D. & Feltovich, P.J. (eds.) *Smart Machines in Education.* Cambridge, MA: MIT Press. pp. 99-144.

Woolley, N.N., Jarvis, Y. (2007) Situated Cognition and Cognitive Apprenticeship: A model for teaching and learning clinical skills in a technologically rich and authentic learning environment. *Nurse Education Today,* 27: 73-79.

Xu, S. & Chee, Y. S. (1998) Transformation-based Diagnosis of Student Programming Errors. *Sixth International Conference on Computers in Education.* Beijing, China

Bibliography

Andre, E., Muller, J. & Rist, T. (1996) The PPP Persona: A Multipurpose Animated Presentation Agent. *Proceedings of the Advanced Visual Interfaces.* New York: ACM Press. pp. 245-247.

Bratko, I. (1986) *Prolog Programming for Artificial Intelligence.* Wokingham: Addison-Wesley Publishing.

Case, D. (2005) A Synthetic Agent For Mentoring Novice Programmers Within A Desktop Computer Environment. In: Panayiotopoulos, T., Gratch, J. Aylett, R. Ballin, D., Olivier, P. & Rist, T. (eds.) *Intelligent Virtual Agents: Proceedings of the 5th International Working Conference, IVA 2005*, Kos, Greece, September. Berlin: Springer. p. 502.

Chen, F. (2003) Agent Oriented Fault Detection, Isolation and Recovery and Aspect-Oriented Plug-and-Play Tracking Mechanism. Thesis. Texas A&M University.

Christiaen, H. (1988) Novice Programming Errors: Misconceptions or Mispresentations? *SIGCSE BULLETIN* 20(3): 5-7.

Coelho, H. & Cotta, J.C. (1998) *Prolog by Example*. Berlin: Springer.

Cooper, S., Dann, W. and Pausch, R. (2003) Using animated 3D graphics to prepare novices for CS1. *Computer Science Education* 13(1): 3-30.

Cooper, S., Moskal, B. and Lurie, D. (2004) Evaluating the Effectiveness of a New Instructional Approach. In: Joyce, D., Knox, D., Dann, W. & Naps, T.L. *Proceedings of the 35th SIGCSE Technical Symposium on computer Science Education*, 3-7 March, Norfolk, VA, 3-7. New York: ACM Press. pp.75-79.

Corchado, J. M. & Pellicer, M. A. (2005) Development of CBR-BDI Agents. *International Journal of Computer Science & Applications* 2(1): 25-32.

Covington, M.A., Nute, D. & Vellino, A. (1996) *Prolog Programming in Depth*. Englewood Cliffs, NJ: Prentice Hall.

Dean, T., Allen, J. & Aldimonds, Y. (1995) *Artificial Intelligence: Theory and Practice*. Menlo Park, CA: Addison-Wesley Publishing.

d'Inverno, M., Kinny, D., Luck M. & Wooldridge, M. (1998). A Formal Specification of dMARS. In: Singh, Rao and Woolridge (eds.), *Intelligent Agents IV in Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*. Lecture Notes in Artificial Intelligence, 1365, Berlin: Springer-Verlag.

Flake, S. & Geiger, C. (2000) Agents with Complex Plans: Design and Implementation of CASA. *From Agent Theory to Agent Implementation II, Proceedings of the 15th European Meeting on Cybernetics and Systems Research*, Vienna, Austria, April. [Online.] Available from: http://citeseer.ist.psu.edu/flake00agents.html. [Accessed 21 January 2010].

Flake, S. & Geiger, C. (2000) Structured Design of a Specification Language for Intelligent Agents. In: Tiagarajan, P.S. & Yap, R. (eds.) *Proceedings of the Fifth Asian Computing Science Conference on Advances in Computer Science*. 10-12 December, Phuket, Thailand. Berlin: Springer. pp. 373-374.

Flenov, M. (2005) *Hackish C++ Pranks & Tricks*. A-LIST LLC

Gutschmidt, T. (2004) *Game Programming with Python, Lua, and Ruby*. Boston, MA: Premier Press.

Hagner, N. & Tunevi, A. (2000) Implementing Case-Based reasoning in SICSus Prolog. SICS technical report, T91: 16.

Jarvela, S. (1995) The Cognitive Apprenticeship Model in a Technologically Rich Learning Environment: Interpreting the Learning Interaction. *Learning and Interaction* 5: 237-259.

Kaiser, G. E, (1990) AI Techniques in Software Engineering. In: Adeli, H. (ed.) *Knowledge Engineering: Vol.II Applications*, McGraw-Hill, Inc.

Karlgren, K., Tholander, J., Dahlqvist, P., & Ramberg, R., (1998) Authenticity in Training Systems for Conceptual Modelers. In: Brookman, A.S., Guzdial, M., Kolodner, J.L. & Ram, A. (eds.) *Proceedings of the International Conference on the Learning Sciences (ICLS-98)*, Atlanta, Georgia, 16-19 December.

Kolodner, J.L. (1983) Indexing and Retrieval Strategies for Natural Language Fact Retrieval. *Transactions on Database Systems* 8(2): 434-464.

Kolodner, J.L. (2000) *Case-Based Reasoning*. San Francisco, CA: Morgan Kaufmann.

Kumar, S. & Cohen, P.R. (2004) STAPLE: An Agent Programming Language Based on the Joint Intention Theory. *Proceeding of AAMAS04, New York City.* New York: ACM. pp. 1390-1391.

Liu, T.C. (2005) Web-based Cognitive Apprenticeship Model for Improving Pre-service Teachers' Performances and Attitudes towards Instructional Planning: Design and Field Experiment. *Educational Technology & Society* 8(2): 136-149.

Luck, M. & d'Inverno, M. (2003) Unifying Agent Systems. *Annals of Mathematics and Artificial Intelligence, Special Issue on Computational Logic in Multi-Agent Systems* 37(1-2): 131-167.

Lutz, M. & Ascher, D. (2003) *Learning Python*, 2nd Ed. Sebastopol, CA: O'Reilly.

Lutz, M. (2004). When Pythons Attack: Common Mistakes of Python Programmers. [Online]. Available from: http://onlamp.com/pub/a/python/2004/02/05/learn_python.html [Accessed 21 January 2010].

Mizoguchi, F. (ed.) (1991) *Prolog and its Applications*. Cook, N. (trans.) London: Chapman and Hall.

McIver, L. (2000). The Effect of Programming Language on Error Rates of Novice Programmers. In: Blackwell, A.F. & Bilotta, E. (eds.) *Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group*, Corigliano Calabro, Italy, April. pp. 181-192.

Mount, S., Shuttleworth, J. & Winder, R. (2008) *Python for Rookies: A First Course in Programming.* Thomson Learning (EMEA) Ltd.

Mueller, E.T. (1990) *Daydreaming in Humans and Machines: A Computer Model of the Stream of Thought*. Norwood, NJ: Ablex Publishing.

Norvig, P. (1992) *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, Inc.

Olivia, C., Chang, C.F., Enguix, C.F. & Ghose, A.K. (1999) Case-Based BDI Agents: An Effective Approach For Intelligent Search on the World Wide Web. *Proceedings of AAAI Spring Symposium on Internet Agents*. pp. 1-8.

Pal, S.K. & Shiu, S.C.K. (2004) *Foundations of Soft Case-Based Reasoning*. Hoboken, NJ: John Wiley & Sons.

Rao, A.S. & Georgeff, M. P. (1991) Modelling Rational Agents within a BDI-Architecture. Online available from: http://eva.cic.ipn.mx/~sher/SID /BDI.pdf. [Accessed: 21/01/10].

Schank, R. C., Kass, A. & Riesbeck, C. K. (1994) *Inside Case-Based Explanation*. Hove, East Sussex: Psychology Press.

Schank, R. C. & Riesbeck, C. K. (1981) *Inside Computer Understanding: Five Programs Plus Miniatures*. Hove, East Sussex: Psychology Press.

Schank, R.C. & Reisbeck, C.K. (1982) *Inside Computer Understanding*. Hillside, NJ: Lawrence Erlbaum Associates.

Shabo, A., M. Guzdial, and J. Stasko. (1996) Computer Science Apprenticeship: Creating Support for Intermediate Computer Science Students. In: Edelson, D.C. & Domeshek, E. A. (eds.) *Proceedings of the International Conference of the Learning Sciences*. Evanston, IL, 25-27 July. International Society of the Learning Sciences. pp. 308-315.

Sterling, L. (1994) *The Art of PROLOG: Advanced Programming Techniques (Logic Programming)*. Cambridge, MA: MIT Press.

Stobo, J. (1989) *Problem Solving with Prolog*. London: Pitman Publishing.

Tholander J. (2001) Students interacting through a Cognitive Apprenticeship Learning Environment. *European Conference on Computer-Supported Collaborative Learning*, Maastricht, The Netherlands.

Tholander, J., Karlgren, K., Rutz, F., & Ramberg, R. (1999) Design and Evaluation of an Apprenticeship Setting for Learning Object-Oriented Modeling. *Paper presented at the ICCE99,* Chiba, Japan. November. [Online] Available from: http://people.dsv.su.se/~klas/Publications/ Design_and_Evaluation_of_an_Apprenticeship_submi.pdf [Accessed 21 January 2010].

Turner, R.M. (1994) *Adaptive Reasoning for Real World Problems: A Schema-Based Approach*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Van Le, T. (1993) *Techniques of PROLOG Programming: with Implementation of Logical Negation and Quantified Goals*. Hoboken, NJ: John Wiley & Sons.

Winograd, T. (1972) *Understanding Natural Language*. Academic Press Inc.

Appendix A:

Brief Overview of Python

A.1 The Python language:

The Python language is the main development tool used to teach programming to the students in the "Foundations of programming" module for the Information Sciences course at the University of Northampton. For a more complete explanation of Python books such as "Programming Python" (Lutz 2001), "Learning Python" (Lutz & Ascher 1999), "Python for rookies" (Mount, Shuttleworth & Winder 2008), and "Game Programming with Python, Lua, and Ruby" (Gutschmidt 2004) are recommended.

$$\begin{tikzcd}
\text{Snobol} \arrow[d] & \text{SED} \arrow[dr] & \text{C} \arrow[d] & \text{AWK} \arrow[dl] & \text{Pascal} \arrow[d] & \text{Basic} \arrow[d] \\
\text{Icon} \arrow[drr] & & \text{Perl} \arrow[d] & & \text{Modula} \arrow[dll] & \text{ABC} \arrow[dlll] \\
& & \text{Python} & & &
\end{tikzcd}$$

Figure A.1. The Python language family tree

Python is a general purpose programming language, it is interpreted therefore supports interactive development, although some features of its syntax are unusual among programming languages it is simple and promotes uncluttered code, it supports a range of high-level abstract data types that are easy to manipulate and has a large range of third party development tools and libraries of code for different applications. The language was first developed, in the 1980s, at the National Research

Institute for Mathematics and Computer Science in the Netherlands by Guido van Rossum. A number of features from older programming language influenced the design of Python as illustrated in Figure 3.1. Python was originally designed as a configuration language for the Amoeba distributed operating system but the design proved to be general enough to allow for application in other domains. Guido has stated that Python was named after a favourite television series, "Monty Pythons Flying Circus" and that the language is greatly influenced by his experience from the development of an earlier programming language designed for teaching called ABC (Lutz 2001).

A.1.1.1.1 Data types:

Python programs support a number of built-in data types such as numbers, strings, lists and dictionaries. Numbers are, quiet conventionally, used for arithmetic and are available as integers and floating point values. Strings are immutable collections characters that can be broken apart and joined together in various ways. In many other programming language strings are mutable (characters may be altered in place) however Python has a large and easy to use set of operators to split and join strings that this limitation is seldom an issue for the programmer. Lists are collections of items of any data type such as numbers, strings or even other lists to model different types of data requirements; the members of a list may also be of mixed types and adding or removing members allows the dynamic alteration of the length of lists.

```
name = [“Michael”, ”Palin”]
nest = [[“a”, ”list”, ”of”], [”lists”, ”containing”], [”some”, ”strings”]]
```

The values within list may be accessed for either retrieval or assignment using the name of the list and a numerical index value e.g.

```
print “the first item is”, nest[0]
name[0] = “Jackson”
```

Items on a list are considered ordered and may be indexed by an integer indicating their position, dictionaries are unordered collections of data items but their position may be indexed by additional data types most often strings, providing an association table of values. Dictionaries are associative memory structures that also hold multiple items of data but this time values are indexed via other data types such as strings e.g.

```
team ={ “idle” : ”eric”,   ”cleese” : ”john”,
          “chapman” : ”graham”,   ”palin” : ”micheal” }
```

Each item in the dictionary forms a key-value pair. To access a value the name of the dictionary and with the key name must be specified e.g.

```
print “the first name is”, team[“chapman”]
name[“palin”] = “sarah”
```

Indexing data items by strings allows for the modelling of data at a higher level of abstraction than the use of simple arrays, for instance representing a database of geographic information attributes such as capitol city, population, etc may be catalogued by the attribute name, even thought it would be possible to duplicate the data handling features of dictionaries by the use of arrays.

A.1.1.1.2 Syntax:

The language was designed to fulfil a number of considerations in mind for the code writer among them, to be easy to learn, easy to use and to support rapid prototyping and turnaround. For these reasons it has a relatively simple syntax and with a small set of keywords built into the language. Python is a weakly typed language in that variables do not hold type information but are merely references to data structures. Variables do not require declaration but are created at instantiation. The simplest statement in Python is assignment that loads a value to a variable e.g.

```
answer = 42
eric = 0.5
parrot = “dead”
```

several values may be loaded at once in a statement e.g.

```
first, second, third = 1, 2, 3
```

Values may be retrieved from variables by using the variable name e.g.

```
series = second * third
print “Life the universe and everything”, answer
```

Python supports a conventional set of arithmetic and logic operators. Expressions containing only integer values produce an integer result; if a floating-point value is present integers are automatically promoted. Python does not support any syntax words to indicate the beginning and end of blocks such as the begin/end in Pascal or braces in C++ and Java, instead Python uses indentation to indicate this e.g.

```
if x > 5:
   print “x is greater than five”

for index in [1,2,3,4,5]:
   print “currently in loop number”, index
```

The plus operator can be used on strings to concatenate them together, in fact a space between strings performs the same operation but the plus operator is required to concatenate lists, so its use on strings produces more consistent code.

Functions in Python are also blocks of indented code with a name and the option of parameters to hold values passed into the function e.g.

```
def square(x):
   return x*x
```

Functions are called by use of their name followed by parenthesis, which may contain values to be passed to the function or remain empty when no value is to be passed. Functions are first class data items meaning function values may be assigned to variables or passed parameters by use of the function name without the parentheses. All functions return a value, even if they do not contain an explicit return value in which case a None object is returned, the Python value for no data.

A.1.1.1.3 Object orientation

Python is an object-oriented language, the built-in data types are implemented as objects and the syntax supports a set of object-oriented programming features to extent the language. However the use of the object-oriented features is entirely optional, it is possible and not unusual to produce substantial programs using only procedural code. For small or experimental programs or those who lack the experience programs can be develop using purely procedural code or object-based applications using object-oriented libraries. As designs grow to require a more structured solution object-oriented programming techniques are available where the programmer can define their own classes and objects. Python objects are created from class prototypes that are used to define the data and methods of the object.

```
class Person:
   def __init__(self, name):
      self.name = name
   def say_hi(self):
      print 'Hello,’, self.name,  ‘how are you?'
```

In order to use a class an object needs to be created and initialised from the class, calling the class by its name performs the instantiation running any code in the __init__ (initialisation) method.

```
p = Person (“Brian”)
```

As with other object-oriented languages the variable, called p, becomes a reference to an object of the type Person. To send a message to the object it’s method may be invoked using a dot notation.

```
p.say_hi()
```

This will cause the code in say_hi to be run, printing the hello message to be printed out. Objects are implemented internally as dictionaries, and message passing may also be performed by conventional dictionary access. Even if the novice programmer does no object-oriented programming they are likely to encounter classes and objects when they access system resources such as file handling objects and graphical libraries like the Tk library called TKinter.

Python is also equipped with a large set of libraries from the developer and third-parties. The most prominent library is the TKinter. TKinter is a cross-platform tool that allows developers to write portable windowed applications that make use of the desktop environment available on operating systems. TKinter is distributed as an integral part of distribution not least because the Python native development environment called IDLE is written in Python using TKinter. The IDLE development environment provides an integrated set of tools that are useful for the production of code such as editor with colour syntax highlighting, a virtual console for interactive code execution and a debugging environment. In addition the source code for TKinter, IDLE and a number of other libraries are all provided in the Python distribution.

It is the availability of the language features like the brevity of the notation, high-level data-types, scalability of the language and large library third party code that makes Python a popular programming language. Additional features such as the interactive development environment, optional object-orientation, etc that makes Python a popular choice as a learning tool for an inexperienced programmer. It is these reasons and also for its availability on machines that the students learn to write code on why Python as also been used to implementation large parts of the agent solution.

Appendix B:

The MRCHIPS User Guide

User guide

A Brief introduction to MRCHIPS

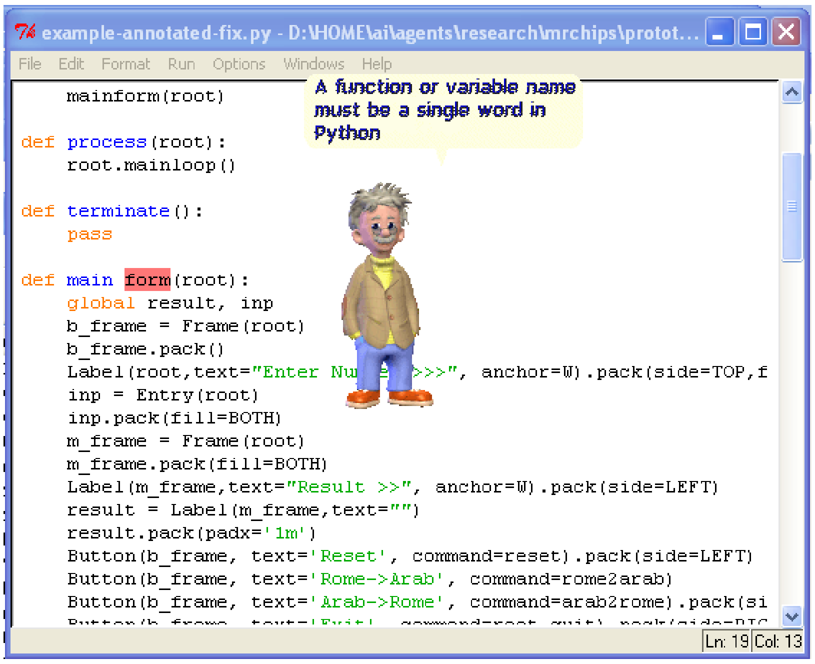
Installation:

- To run MRCHIPS you need to have the following programs installed:
    - Python 2.4 or Greater
    - PyWin32

- If you are using Python 2.4 you will also need to install ctypes library.

- MRCHIPS also requires MS-agents for its user interface.

- If you are using MRCHIPS on Windows 7 you will have download and install MS-agents from the Microsoft web site.
- To install MRCHIPS copy the files onto your computer.


example-annotated-fix.py - D:\HOME\ai\agents\research\mrchips\protot...
File Edit Format Run Options Windows Help
Ln: 19 Col: 13

A function or variable name must be a single word in Python

```
    mainform(root)

def process(root):
    root.mainloop()

def terminate():
    pass

def main form(root):
    global result, inp
    b_frame = Frame(root)
    b_frame.pack()
    Label(root,text="Enter Nu    >>>", anchor=W).pack(side=TOP,f
    inp = Entry(root)
    inp.pack(fill=BOTH)
    m_frame = Frame(root)
    m_frame.pack(fill=BOTH)
    Label(m_frame,text="Result >>", anchor=W).pack(side=LEFT)
    result = Label(m_frame,text="")
    result.pack(padx='1m')
    Button(b_frame, text='Reset', command=reset).pack(side=LEFT)
    Button(b_frame, text='Rome->Arab', command=rome2arab)
    Button(b_frame, text='Arab->Rome', command=arab2rome).pack(si
```

Figure 1. The MRCHIPS agent offering advice to the programmer

Running the MRCHIPS:

- To start MRCHIPS locate the main.pyw file and double click
- The MRCHIPS agent will appear, announce its presence and then hide the Windows toolbar.
- MRCHIPS will monitor the desktop from the toolbar but can be launched manually

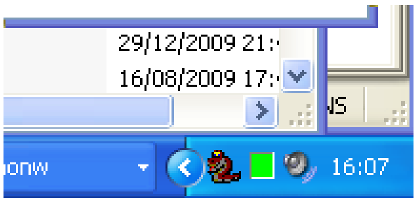


Figure 2. The MRCHIPS toolbar control

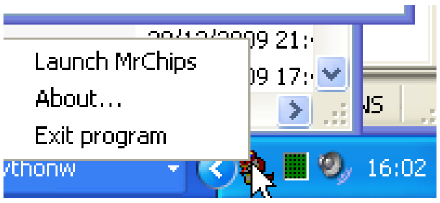


Figure 3. The MRCHIPS toolbar control menu

- When MRCHIPS is on the desktop an accompanying dialog box is often present, which can be used to responses to questions from MRCHIPS.

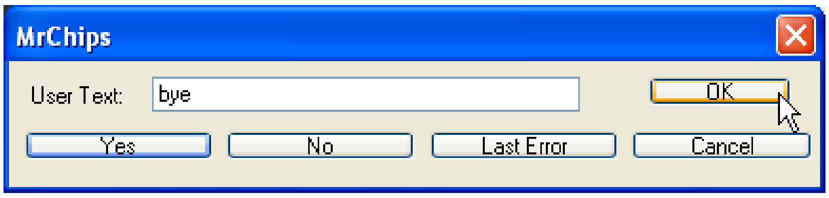


Figure 4. The MRCHIPS input dialog box

- "yes" or "no" answers may be entered into the user text field, or the by pressing the buttons in response to questions
- MRCHIPS can be made to hide by typing "hide" or "bye" into the user text field
- To shut down MRCHIPS the exit option may be chosen from the menu in the toolbar icon or by typing an "exit" command into the user text field

Appendix C:

The Evaluation Brief

MRCHIPS The Python Programmers Assistant

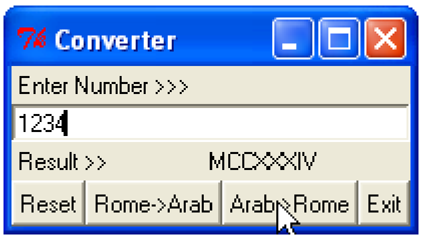
Roman-Arabic numerals converter

Introduction

This is the preparation for the testing of a Python desktop assistant to help programmers as they find their way around a programming language for the first time. Please read through the following information carefully, so that you come to the test, with everything you need to know to do your best. This exercise is run in the format of a Time-Constrained Assignment (TCA) but all results are for the purpose of testing the agent and any results gathered will be made anonymous for use.

Background

The Roman-Arabic numerals converter is a small educational application designed to make demonstrate number theory in a fun and easy way. Based on an idea from an application originally developed in a different language a number of errors were introduced when implementing the Python version of the code.



To aid you in correcting the program you have the assistance of MRCHIPS a desktop agent that is able to provide mentoring support as you work your way through the problems. MRCHIPS will sits out of site for most of the time as you work your way through your program but if you should encounter any errors that you are unable to solve by yourself will appear to offer assistance. Please note: this is an early test of the MRCHIPS agent so it may not always precise with its help.

You are required to:

1) Run, test and debug the program until it works as designed.
2) Indicate on the hard copy of the program were you have fixed bugs or altered the program.
3) Add additional comments to the program (to help illustrate your understanding).

Deliverables:

1. A soft copy of your corrected program code worked on.
2. The log file from MRCHIPS, called journal.txt
3. A completed copy of the questionnaire.

Guidance:

1. Use the information presented to you by the Python environment, line numbers, highlighted areas etc.
2. Use the example program to ensure you understand how the program should behave. Any differences (behaviour, colour, position, etc) should be treated as bugs to be fixed.
3. Deal with one error at a time, one code change at a time run and test your program frequently.
4. There are about a dozen errors that need to be corrected.

Appendix D:

The Evaluation Test Source Code

```
from Tkinter import *

##----------------------------------
##   Arabic/Roman numerals
##    converter in python
##    by D.Case  20/04/08
##----------------------------------

def initialise(root):
    root.title('Converter')
    mainform(root)

def process(root):
    root.mainloop()

def terminate():
    pass

def mainform(root)
    global result, inp
    b_frame = Frame(root)
    b_frame.pack(side=BOTTOM)
    Label(root,text="Enter Number >>>", anchor=W).pack(side=TOP,fill=BOTH)
    inp = Entry(root)
   inp.pack(fill=BOTH)
    m_frame = Frame(root)
    m_frame.pack(fill=BOTH)
    Label(m_frame,text="Result >>", anchor=W).pack(side=LEFT)
    result = Label(m_frame,text="")
    result.pack(padx='1m')
    Button(b_frame, text='Reset', command=reset).pack(side=LEFT)
    Button(b_frame, text='Rome->Arab', command=rome2arab).pack(side=LEFT)
    Button(b_frame, text='Arab->Rome', command=arab2rome).pack(side=LEFT)
    Button(b_frame, text='Exit', command=root.quit).pack(side=RIGHT)


def arab 2rome():
    val = inp.get()
    try:
        num = int(val)
    except ValueError:
        num = val
    result['text'] = int_to_roman(num)


def rome2arab():
    val = inp.get()
    result['text'] = roman_to_int(val)
```

```
def int_to_roman(arabic):
    """ Convert an integer to a Roman numeral. """
    If not isinstance(arabic, type(0)):
        return "expected integer, got %s" % type(arabic)
    if not 0 < Arabic < 4000:
        return "Argument must be between 1 and 3999"
    ints = (1000, 900,  500, 400, 100,  90, 50,  40, 10,  9,   5,  4,   1)
    nums = ('M',  'CM', 'D', 'CD','C', 'XC','L','XL','X','IX','V','IV','I')
    result = []
    for i in range(len(ints)):
        count = int(arabic / ints[i])
        result.append(nums[i] * count)
        arabic -= ints[i] * count
    return ''.join(result)


def roman _to_int(roman):
    """ Convert a Roman numeral to an integer. """
    if not isinstance(roman, type("")):
        return "expected string, got %s" % type(roman)
    roman = roman.upper    # upper case letters for conversion
    nums = {'M':1000, 'D':500, 'C':100, 'L':50, 'X':10, 'V':5, 'I':1}
    total = 0
    for i in range(len(roman)):
        try:
            value = nums[roman[i]]
            # If the next place holds a larger number, this value is negative
            if i+1 < len(roman) and nums[roman[i+1]] > value:
                total -= value
            else:
                total += Value
        except KeyError:
            return 'roman is not a valid Roman numeral: %s' % roman
    # easiest test for validity...
    if int_to_roman(total) = roman:
        return total
    else:
        return 'roman is not a valid Roman numeral: %s' % roman

def reset(root)
    result['text'] = ""
    inp.delete(0, len( inp.get() ))

def main():
    root = Tk()
     initialise(root)
    process(roo)
    terminate()

main()
```

Appendix E:

The MRCHIPS Evaluation Questionnaire

Please answer the following questions as clearly as possible:

1. What gender are you? ☐ Female ☐ Male

2. What is your age group?
☐ < 18 ☐ 18-25 ☐ 26-35 ☐ 36-45 ☐ 46-55 ☐ 56-65 ☐ >=66

3. Length of prior programming experience?
☐ None ☐ < 1 Year ☐ 1-2 Years ☐ 2-3 Years ☐ > 3 Years

4. Type of programming experience?
☐ None
☐ Hobby/self taught
☐ Part of a course
☐ Other, please specify

5. Have you ever written a program other than for your studies?
☐ No
☐ Yes, please specify

6. Did MRCHIPS appear during your programming session? ☐ Yes ☐ No

7. How many times did MRCHIPS offer help you to solve?
☐ 0 ☐ 1-2 ☐ 3-4 ☐ 5-6 ☐ 7-8 ☐ > 9

8. Did you find the help offered accurate?
☐ 0 ☐ 1%-25% ☐ 25%-50% ☐ 50%75% ☐ 75%-100%

9. How responsive was MRCHIPS when you found an error?
☐ too slow ☐ about right ☐ too quick

10. Did you find MRCHIPS more of a help or hindrance to your working?
☐ Help
☐ Hindrance

11. Did you have to ask the tutor for additional help during the session?
☐ No
☐ Yes, please specify

12. During the session did consult any other sources of programming sources of help?
☐ No
☐ Yes, please specify

A Pedagogical Agent

An Animated Pedagogical Agent For Assisting Novice Programmers Within A Desktop Computer Environment

Desmond Case, Bernadette Sharp, Len Noriega - University of Staffordshire

Abstract

This research proposes that an intelligent animated agent is able to provide learning support, in the form of mentoring, to novice programmers within the Cognitive Apprenticeship pedagogy. This small paper outlines the nature of learning to program, how an intelligent agent may be used to support the learner and the design of a new architecture, called MRCHIPS, to control reasoning and behaviour for such an agent.

1. Introduction

The question addressed by this research is whether an animated pedagogical agent can provide effective mentoring support for the novice when learning a programming language for the very first time. The original contribution of this approach is the use of an intelligent agent for mentoring programming students (rather than tutoring) within the Cognitive Apprenticeship pedagogy.

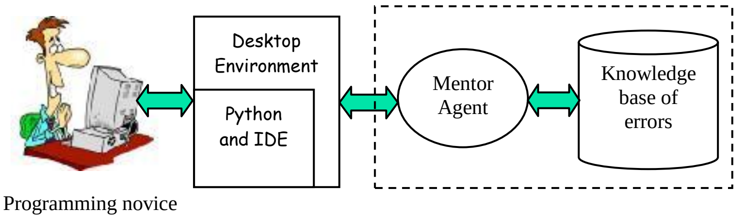


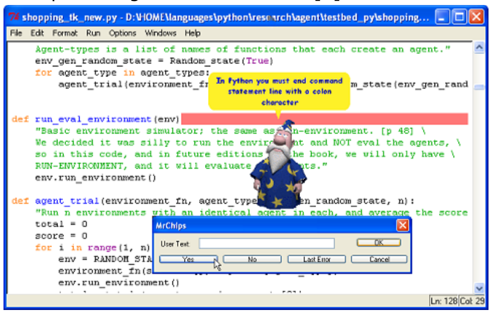
Figure 1. Outline of the novice and mentor agent interaction

2. The Problem

Educational researchers [5] have observed that novice programmers make the same mistakes and encounter the same problems when first learning a programming language. The learner errors are usually from a fixed set of misconceptions that are easily corrected by experience and with simple guidance. Despite rich interactive development environments, learners continue to generate errors as they experiment with the language structures and find debug messages unhelpful because of their lack of experience of the significance of error information. During practical sessions a supervisors task is often to simply call on prior experience to offer guidance and offer reassurance that errors are all part of the development process.

3. Background Theory

The behaviour of a tutor during practice based sessions is to provide coaching in that the learner is encouraged to develop code by themselves and the tutor offers support as they require it. The support is then gradually reduced as the learner becomes more skilled. This approach is closest to the methods of the Cognitive Apprenticeship pedagogy [2], the tutors support can take a number of forms such as explanations, examples or specific direction depending on the nature of the problem, the learners preferences etc. but interactions require the learner be an active participant in producing work of their own [4].


shopping_tk_new.py - D:\HOME\languages\python\research\agent\testbed_py\shopping...
File Edit Format Run Options Windows Help
In Python you must end command statement line with a colon character
MrChips
User Text
OK
Yes
No
Last Error
Cancel
Ln: 128 Col: 29

```
    Agent-types is a list of names of functions that each create an agent."
    env_gen_random_state = Random_state(True)
    for agent_type in agent_types:
        agent_trial(environment_fr                    m_state(env_gen_rand


def run_eval_environment(env)
    "Basic environment simulator; the same as  n-environment. [p 48] \
    We decided it was silly to run the envir   nt and NOT eval the agents, \
    so in this code, and in future editions   he book, we will only have \
    RUN-ENVIRONMENT, and it will evaluate        nts."
    env.run_environment()

def agent_trial(environment_fn, agent_type        en_random_state, n):
    "Run n environments with an identical agent in each, and average the score
    total = 0
    score = 0
    for i in range(1, n)
        env = RANDOM_STA
        environment_fn(s
        env.run_environment()
```

Figure 2. MRCHIPS offering advice to a learner

4. Proposed Solution

An agent system could be made to sit alongside the learners development environment to monitor activity as they write code and alert or advise them of errors and problems in a format suited to the requirements of a novice as illustrated in figure 1. The use of an animated agent character offers the advantage of modes of communication that are more intuitive to the learner and avoids the cognitive load of learning an additional application interface. A number of projects have investigated the effectiveness of animated characters for imparting information to the user [5]. A responsive agent system would help to maintain the effect of a knowledgeable character [9]. The mentor would also need to be able to monitor the users activity, analyse the nature of a users' problem and provide an effective response. For these reasons the following architecture innovation is proposed. An illustration of the MRCHIPS agent using a character from the Microsoft's Agent interface and working in the Python environment is shown in figure 2.

5. Agent Architecture

The MRCHIPS architecture consists of a hybrid of two reasoning systems based on Beliefs-Desires-Intentions (BDI) and Case-Based Reasoning (CBR) (see figure 3) and other support systems. The two reasoning systems coordinate the different levels of analysis required to provide the capabilities of the agent. The BDI system provides the processing required to interface to the environment, monitor the user and control the activity of the agent character. The reactive and deliberative capabilities of the BDI [1] allow the agent to track low-level user tasks such as window position and mouse clicks. By tracking the user's activity this layer will also be able to make inferences about user activities and select suitable responses for the agent. The CBR system maintains specific domain knowledge about analysis of programming errors and strategies for communicating solutions to the learner.

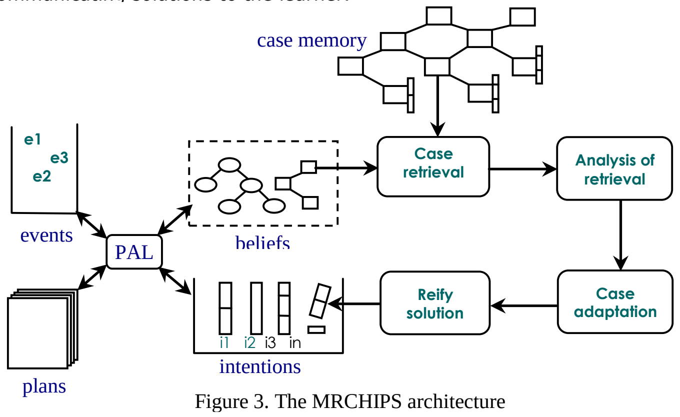


Figure 3. The MRCHIPS architecture

The CBR subsystem makes use of information from the BDI data structure to form the problem when a matching case is selected the agent is committed to performing the solution by its inclusion on the list of agent intentions.

6. Related Work

Other research has also proposed combining of BDI-CBR agent systems [3, 8] for intelligent web searching and a tourist guide agent. These systems have primarily been concerned with adding learning capabilities to BDI and have in different ways used CBR to implement BDI agents. The innovation with the proposed agent architecture is that the BDI-CBR subsystems are structured to reason in parallel to provide the spectrum of agent behaviours, in a similar way to hybrid agent systems such as INTERRAP [7].

7. Current Progress

A prototype of the MRCHIPS agent was completed in autumn 2009 equipped with a knowledge base for Python programming students. Testing was carried out on a group of novice Python programmers results demonstrated a mean grade improvement of 40% when compared to novice students who worked without the aid of the agent.

8. References

[1] Ancona, D., Demergasso, D. & Mascardi, V. (2005). A Survey on Languages for Programming BDI-style Agents. [Online.] Available from: http://www.cs.uu.nl/~mehdi/al3files/VivianaMascardi.ps

[2] Collins, A., Brown, J.S. & Newman, S.E. (1989) Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In: Resnick, L.B. (ed.) Knowing, learning and instruction: Essays in honor of Robert Glaser. Hillsdale, NJ: Erlbaum. pp. 453-494.

[3] Corchado, J. M. & Pellicer, M. A. (2005); Development of CBR-BDI Agents. International Journal of Computer Science & Applications. Vol. 2: 1, pg, 25-32

[4] Ghefaili A. (2003) Cognitive Apprenticeship, Technology, and the Contextualization of Learning Environments. Journal of Educational Computing, Design and Online Learning 4 (Fall).

[5] Gulz, A. (2004); Benefits of Virtual Characters in Computer Based Learning Environments: Claims and Evidence. International Journal of Artificial Intelligence in Education 14. pg. 313-334

[6] Luck, M., Ashri, R. & d'Inverno. (2004); Agent-Based Software Development. Artech House Computing Library.

[7] Muller, J. (1996); The Design of Intelligent Agents, A Layered Approach. Springer-Verlag.

[8] Olivia, C., Chang, C.F., Enguix, C.F. & Ghose, A.K. (1999); Case-Based BDI Agents: An Effective Approach For Intelligent Search on the World Wide Web. Procedings of AAAI Spring Symposium on Internet Agents.

[9] Rist, T., Andre, E., Baldes, S., Gebhard, P., Klesen, M., Kipp, M., Rist, P.,Schmitt, M. (2003); A review of the development of embodied presentation agents and their application fields. Life-like Characters: Tools, Affective Functions and Applications.