Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-Memorandum ID-2002-16

Building an Intelligent Help System for AdaptIt

Distribution:

Jos Rohling (ID) (3x)
Martijn Vastenburg (TUD)
Anneke Donker (ID)
Jelke van der Pal (VE)
Harmen Abma (VE)
Henrik Schlanbusch (UiB)
Oyvind Meistad (UiB)
Nikos Vlassis (UvA)
UvA Onderwijsburo (2x)
Wouter Kalis (3x)

Division:
Information and Communication Technology

Order-/codenumber:
1506.4.3

Prepared:
Wouter Kalis /

Issued:
September 2002

Approved:
JCD / JR /

Classification title:
Unclassified

Summary

The National Aerospace Laboratory (NLR) is participating in a project called AdaptIt. The main goal of this project is to develop a marketable ICT-based tool for training designers. Future users are currently testing the tool and they have indicated that they would like the AdaptIt tool to give them more operational support.

This thesis describes the building of an intelligent help system for the AdaptIt tool called the I-Advisor. This I-Advisor will provide the users with more context-sensitive support, e.g. by giving detailed help information on the task the user is currently working on. This support is based on an internal user model, which the I-Advisor uses to model the user's interactions with AdaptIt, as well as a previously learned task model. The task model is learned from logs of experienced users' actions and describes the way a user should behave in the AdaptIt application to successfully design a training. Two different architectures for the I-Advisor are described in detail. The first architecture is based on Collagen and the second makes use of Hidden Markov Models. Prototypes of both architectures were implemented, and a summary of the performance of these prototypes is given.

Acknowledgements

My master thesis project was a very interesting and fun project to work on. The AdaptIt application turned out to be ideal for implementing and testing the intelligent help system architectures I had in mind. The fact that a prototype of the I-Advisor worked so well that it was actually added to the AdaptIt application is proof of that. In the beginning seven months seemed like more than enough time, but somehow at the end I still had thousands of ideas left I wanted to try out. All these ideas for further research can be found in section 10.3 for those of you who are interested.

All the people involved in the project were very enthusiastic and helped me out in a lot of ways. I am almost afraid to name all the people who helped me, since this might make it seem like I didn't do anything myself.

First of all I'd like to thank both my mentors at NLR, Martijn Vastenburg and Jos Rohling. Martijn for all his enthusiasm for the project and helping me out even after he left NLR. Jos for putting up with all my questions throughout the whole project. I would also like to thank Harmen Abma, Jelke van der Pal, Henrik Schlanbusch and Øyvind Meistad of the AdaptIt project. Harmen and Jelke for the opportunity to have the I-Advisor tested at LVNL and for answering all my questions about AdaptIt. Henrik and Øyvind for all their help in integrating my I-Advisor prototype into the AdaptIt application. My thanks also goes out to the people at LVNL who tested my I-Advisor, Jeano de Bock and Jolanda Groothuismink. Thanks also to my room-mate at NLR, Rudy Ujzanovitch, I must have bothered him a thousand times with ideas I wanted some feedback on and questions. And last but not least, Nikos Vlassis, my mentor from the University for all his help.

Contents

(100 pages in total)

1 Introduction

The National Aerospace Laboratory (NLR) is participating in a project called AdaptIt. The main goal of this project is to develop a marketable ICT-based tool for training designers. This tool embodies a validated training design methodology for personalised training. Future users are currently testing the program. They have indicated that they would like the AdaptIt tool to give them more operational support.

The level of support currently available in AdaptIt includes the Advisor and the Q-Advisor. The Advisor is a static JavaHelp-based help system, which includes how-to information, examples and background information. The user can navigate through the pages of the advisor using a table of contents, an index, and also via a graphical overview of the methodology.

The Q-Advisor, short for Quick Advisor, aims to aid the user in finding the appropriate help information in the Advisor. It consists of a separate panel within the AdaptIt application. This panel shows links to the pages in the Advisor corresponding to the editor that the user last selected. For instance if the user is working within the Skill Hierarchy diagram, then the Q-Advisor shows links to all the help pages on how to make a Skill Hierarchy.

Users of the AdaptIt tool still had trouble learning how to use the tool. The users felt they did not have a good overview of the whole process of designing a training. Although the Q-Advisor provided them with background information about how to use the different editors, it did not give them information about when to use the editor, or where they were located in the whole process of designing a training.

We propose a new advisor to give the users more context-sensitive support, e.g. by giving detailed help information on the task the user is currently working on. This new advisor is called the I-Advisor, short for Intelligent Advisor. My master thesis consists of a search for possible architectures for the I-Advisor, as well as a description of the design and implementation of two of these architectures. Chapter 2 starts with an analysis of the existing AdaptIt application, Advisor and Q-Advisor, followed by a description of what the role of the I-Advisor should be. Literature research to intelligent help systems in general can be found in chapter 3. Chapter 4 describes all the formal requirements that the new advisor should meet. Chapters 5 through 7 consist of a detailed description of the architecture chosen for the I-Advisor. Tests of a prototype of the chosen architecture by future users are described in chapter 8. Chapter 9 describes an alternative architecture for the I-Advisor, namely Hidden Markov Models. My thesis concludes with a summary of the lessons learned and ideas for possible future extensions.

2 Analysis of the Support in AdaptIt

The users of the AdaptIt tool, i.e. the training designers for whom the tool was developed, felt something was missing from the level of support. They wanted information about the context of the task they were working on at that moment. They also felt they did not have a good idea of how their current task related to the whole task of designing a training. First AdaptIt and the methodology the tool is based on will be described in more detail in order to explain why they had this feeling. Previous work to give the user more context-sensitive support was done by one of my mentors at NLR, Martijn Vastenburg. Section 2.2 describes the improvements he made to the Advisor and the Q-Advisor to help the user. This chapter concludes with a section about the role the I-Advisor should have in the AdaptIt tool.

2.1 Description of AdaptIt

The AdaptIt tool embodies a validated training design methodology for personalised training of complex skills (de Croock et al. 2002). This training design methodology is based on cognitive science and optimises the integrated use of advanced training technologies. Traditional training techniques focus on a single simple skill. A training consists of a series of predesigned steps which each train a single simple skill. To train this skill the trainee is given a small workload in the beginning, which increases gradually during the training. This leads to stress at the end of the training. The AdaptIt methodology aims to integrate all the different skills from the beginning of the training. This is accomplished by constantly training the whole task and not one single skill. The workload is distributed evenly over the whole training, leading to less stress at the end of the training. The objective of the training is to train skills and not knowledge. Knowledge is merely supportive in the training.

Designing a training in AdaptIt consists of two parts: building a *skill hierarchy* and making a *blueprint*. The skill hierarchy is a graphical presentation of a set of skills and sub skills that enable an expert to perform a complex skill. An example of a skill hierarchy describing the complex skill of driving a car is shown in figure 2.1. This complex skill consists of three different sub skills, namely shifting, braking and steering. These sub skills can also be divided into sub skills. Shifting gears for instance can be described with the sub skills shifting gears up and shifting gears down. This example skill hierarchy however has only two levels.

An Example Skill Hierarchy for the Complex Skill "Driving a Car"

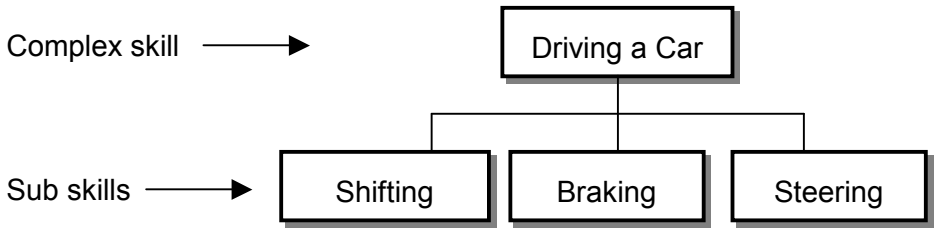

Figure 2.1

The blueprint describes a global outline of the training program in the form of a simple to complex sequence of *task classes*. A task class describes a series of *learning tasks* with the same complexity. If we take the example of training the complex skill "Driving a Car", a possible task class would be a series of learning tasks for driving in sunny weather with a small amount of other cars on the road. A more complex task class would contain learning tasks for driving in rush hour with snow on the road. The specific learning tasks would then be the different types of traffic situations the trainee is confronted with during the driving, such as a highway or a roundabout.

The two main parts in AdaptIt, building the skill hierarchy and making the blueprint, consist of several steps. Figure 2.2 shows all the constituting steps. Building a skill hierarchy is composed of five steps. These steps have to be carried out by the training designer to create a good skill hierarchy.

The Main Steps in Designing a Training

1) Build Skill Hierarchy	2) Make Blueprint
a) Analyse Complex Skill	a) Define Skill Clusters
b) Determine Skills To Be Trained	b) Sequence Task Classes
c) Specify Performance Objectives	c) Design Learning Tasks
d) Classify Skills as (Non) Recurrent	d) Design Supportive Information
e) Identify Complexity Factors	e) Design Just-In-Time Information
	f) Design Part-Task Practice

Figure 2.2

This division of the methodology into these parts and steps is meant to guide the users through the AdaptIt tool. The users of the tool are the training designers who will use the tool to help them design their trainings. As a user becomes more experienced, he will have less need of this division of the methodology into parts and steps to guide him. He will develop his own way of using the tool. All the different steps listed in figure 2.2 have to be carried out at one time or another, but they can be achieved in different orders. The steps can for instance be executed in a cyclical way. Some users might have the tendency to first achieve the first four goals of the building of a skill hierarchy, and then go back to the first goal again to refine the hierarchy. At the end they will focus on the last goal, i.e. identify the complexity factors. A user can even decide to start off with identifying the complexity factors if he chooses to, as well as jump around between the goals at any time. The AdaptIt tool is built in such a way that it places no restrictions on the execution order of the goals.

2.2 Description of the Advisor and the Q-Advisor

Both the Advisor and the Q-Advisor were built to aid the user in learning how to use the AdaptIt tool. Some parts of the Advisor were added specifically to give the user a better overview of the whole application. Figure 2.3 shows an example Advisor page. This page gives information about the design of a skill hierarchy. A graphical overview of the methodology behind AdaptIt is located at the top of the Advisor. The graphical representation of the skill hierarchy is highlighted in this overview because the user has opened the Advisor page about the skill hierarchy. On the top left a list of the different phases in AdaptIt can be found, and a table of contents of all the help pages is shown at the bottom left. All these different parts of the Advisor were added with one goal in mind: to give the user a better overview of the whole process of designing a training.

Although the user had several navigation possibilities through the Advisor pages at his disposal, he still had to find the right Advisor pages on his own. He had to know what goal he was working on. He could then look for this goal in the Advisor pages by using the graphical overview, the list of phases or the table of contents. The Q-Advisor's main objective is to make this task easier for the user. The Q-Advisor displays links to the Advisor pages, based on which editor the user has selected. Figure 2.4 contains a screenshot of the AdaptIt tool, along with the Q-Advisor in the bottom left corner. The user is currently working within the Skill Hierarchy diagram, which is the top right panel. The Q-Advisor thus shows links to all the help pages about how to make a Skill Hierarchy. These links are in this case "How to create a skill hierarchy" and "Description of a skill hierarchy". Clicking on these will take the user directly to the corresponding Advisor pages.

The Advisor

Graphical Overview of the Methodology

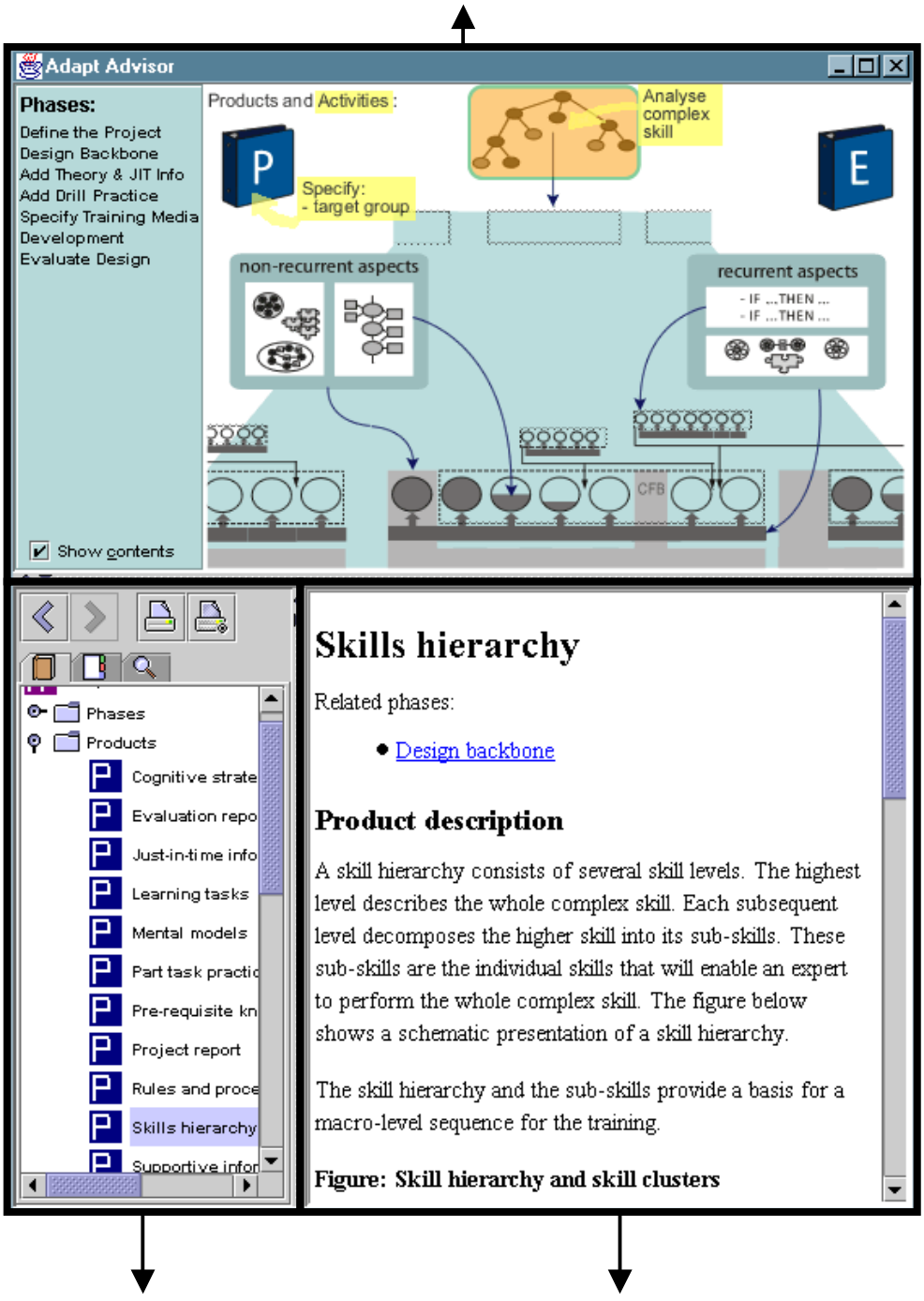

Table of Contents

Help Page in the Advisor

Figure 2.3

AdaptIt application with the Q-Advisor

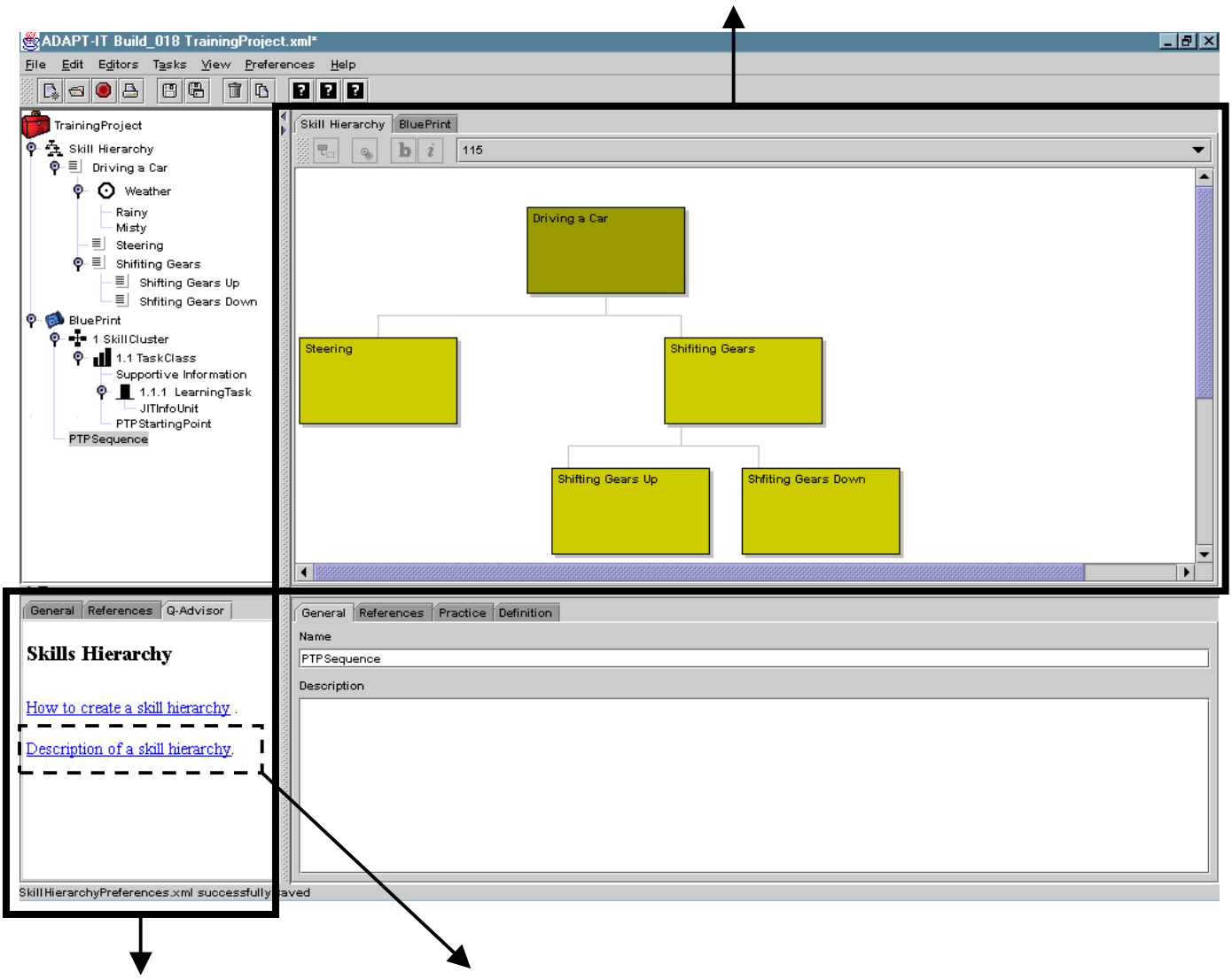

Figure 2.4

It is possible though to accomplish the different steps of designing a training in one and the same editor. In the Blue Print Diagram editor for instance the user can accomplish almost all the steps listed in figure 2.2 as being part of making a blueprint. By looking only at the editor, the Q-Advisor can say nothing about which of these steps the user is trying to accomplish at that moment. He can only list all the different possible goals the user can accomplish in that editor. The Q-Advisor does not take into account the context in which the user selected that specific editor.

2.3 Role of the I-Advisor

The I-Advisor should be seen as an improved version of the Q-Advisor, which not only looks at the user's last action, but also bases its advice on the context of the user's actions. The I-Advisor will build an internal model of the user based on all the user's actions in AdaptIt. It will need some form of a previously learned task model that describes the way an experienced user of the AdaptIt tool behaves in the application. The I-Advisor can compare this dynamic user model to the static task model, and learn where in the whole process of designing a training a certain user is to be found. Once the I-Advisor has learned the user's current task, it can show links to the corresponding Advisor pages in the I-Advisor panel. These links will be shown to the user in such a way that he can see the context of his current task, as well as where he is in the whole task of designing a training. Chapter 4 contains a formal description of the requirements the I-Advisor should meet.

The whole process of giving advice is shown in figure 2.5. The user performs an action in the AdaptIt tool. The I-Advisor listens to the AdaptIt tool and receives this action. It updates its user model with this action with the help of the previously learned task model. If the I-Advisor has successfully deduced the user's current task, it will send new advice to the I-Advisor panel in the AdaptIt tool. This panel shows links to the help pages in the Advisor corresponding to the user's current task context.

The I-Advisor's advice is given in the same panel as the Q-Advisor, without disturbing the user during his work. It is left up to the user whether or not he makes use of the advice. It was possible to give the I-Advisor a more active role than the Q-Advisor, for instance by leading the new user through the application. The next paragraph will explain why the I-Advisor will not be given this more active, tutor-like role.

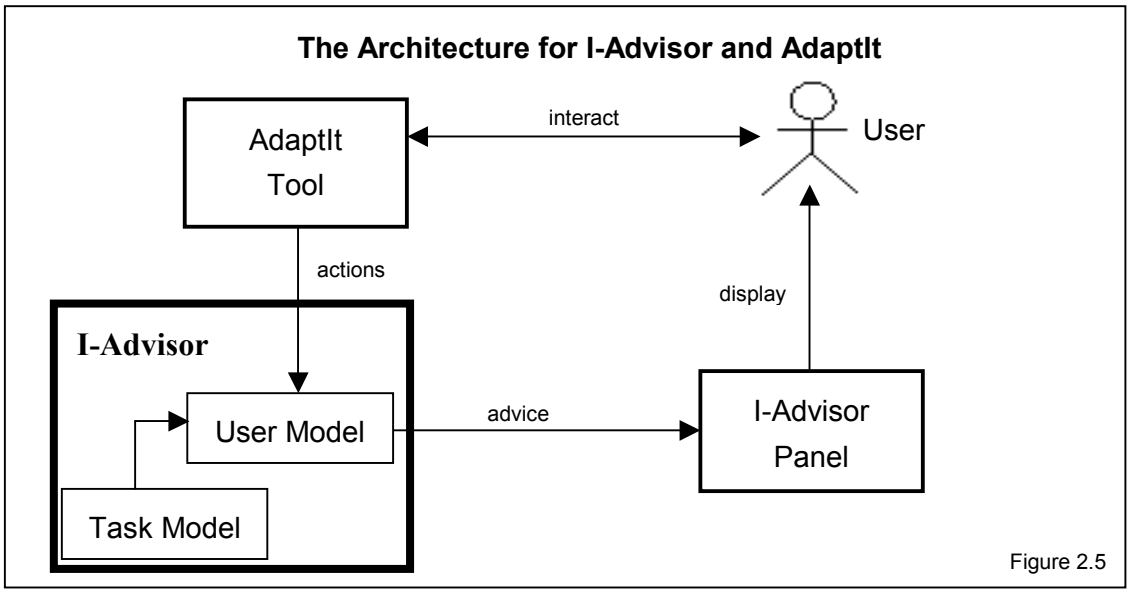

Figure 2.5

Figure 2.2 described all the different goals that are part of designing a good training according to the AdaptIt methodology. New users can use this sequence of goals to guide them through the tool. The AdaptIt tool was built in such a way that it places no restrictions on the execution order of the goals. This is to encourage experienced AdaptIt users to discover their own way of using the tool. The I-Advisor also wants to encourage users to learn to use the tool in the way that suits them most. A wizard or tutor often makes a lot of choices for the user, whereas we want the user to makes these choices himself. That way he is more conscious of the choices he has made. Therefore the I-Advisor will have the same passive role as the Q-Advisor and be more of an advisor than a tutor.

3 Literature Research

The I-Advisor's objective is to help users in learning how to interact with the AdaptIt tool. To accomplish this the I-Advisor makes use of an intelligent technique called user modeling to maintain a model of the user. User modeling is part of a research domain called *intelligent user interfaces* (IUI's). Section 3.1 explains what IUI's are, including examples of the different intelligent techniques used in IUI's and examples of the possible application areas for IUI's.

In order to be able to provide a user of the AdaptIt application with support, the I-Advisor has to discover what goal this user wants to accomplish with his actions in the AdaptIt tool. In other words, the I-Advisor has to recognise what the user's plan is. This process is called *plan recognition*. Section 3.2 looks at the different types of plan recognition, and explains the type of plan recognition chosen in the I-Advisor.

The last two sections will become more specific. First I will go over the possible architectures for intelligent help systems such as the I-Advisor. This chapter concludes with a description of the architecture I chose to base the I-Advisor on which is called Collagen (Rich, Sidner & Lesh, 2001).

3.1 Intelligent User Interfaces

Users today have access to more information than ever before. Interfaces through which users interact with all this information tend to become very complex. It can also happen that there is a limited amount of time in which a task needs to be achieved, and it is nearly impossible to achieve tasks quickly with certain interfaces. Or you can have a certain application, which is used by completely different users with different needs. All these users however have to use exactly the same interface. Another possibility is that the domain in which the application is used changes. All of the above situations are examples of when regular interfaces can become too complex or too inflexible. To help users in these kind of situations intelligent user interfaces were born (Lieberman 1997).

The research area of intelligent user interfaces is at the boundary of a large amount of different research areas. The two most important of these research areas are called *artificial intelligence* (AI) and *human-computer interaction* (HCI). AI tries to model the way a human thinks in order to create a computer system that can do intelligent actions. In HCI computer interfaces are designed that leverage off a human user to aid the user in the execution of intelligent actions. An IUI should both be able to do intelligent actions and to leverage off the human user's intelligence. In other words, IUI's are human-machine interfaces that aim to improve the

efficiency, effectiveness, and naturalness of human-machine interaction by representing, reasoning and acting on models of the user, domain, task, discourse, or media (Maybury & Wahlster 1998). In the following paragraphs a few of the "intelligent" techniques that enable an intelligent user interface to improve human-machine interaction are described. Also some examples of possible application areas for IUI's will be given.

3.1.1 Properties of Intelligent User Interfaces

One of the key intelligent techniques used in intelligent user interfaces is *user modeling*. In order to be able to adapt the interface to a user, a model is built that contains the unique properties of that specific user. Properties of a user can be for example the user's knowledge or experience, interests, intentions and cognitive properties of the user. An example of such a cognitive property is reaction speed in case the interface has to operate in a time-critical environment. A property of a user is stored in the user model if it is useful for that specific application.

An intelligent interface that maintains a user model can either be *adaptable* or *self-adaptive*. An adaptable interface lets the user choose how the system should adapt, whereas a self-adaptive interface adapts to the users autonomously. It learns the user's needs from his interactions with the system and adapts to them. The distinction between adaptable and self-adaptive programs can be made even more precise, with several levels of adaptivity in between the two. These levels can depend on who takes initiative to an adaptation, who proposes the adaptation, who decides upon it and who finally carries it out. An example can be that a program decides that a user is better off with a different format of menus. He proposes it to the user, thus taking initiative himself. The user can accept or reject the idea. If he accepts it the program will carry it out. In this case the program has the initiative, does the proposal and carries it out, whereas the control remains with the user because he decides whether to carry out the proposal or not.

The main intelligent technique the I-Advisor will use is user modeling. The I-Advisor maintains an internal model of the user that stores certain properties of that specific user. Based on the information in this user model, the I-Advisor's panel is adapted to the user. The I-Advisor is self-adaptive, it adapts to the user autonomously. However it only adapts the interface in its own panel. The rest of the interface of the AdaptIt tool is never changed, since we do not want to disturb the user in his work.

Two other intelligent techniques used in IUI's are called *natural language processing* and *speech techniques*. With this an interface can either interpret or generate natural language utterances, either in text or in speech. This makes it easier for users to communicate with an interface, or easier for the interface to make something clear to the user.

Some intelligent user interfaces use a technique called *multimodal interaction*. This means that different modalities or ways of communicating between the user and the interface are made available. The idea behind this is that the human-computer interaction is made easier due to the richer set of channels through which the program and the user can now communicate. One can think of adding such ways of communicating such as natural language, video or three-dimensional graphics to a simple graphical and textual interface and combining all these different inputs to get a better idea of what the user wants.

3.1.2 Applications of Intelligent User Interfaces

The main application area for intelligent interfaces is in those situations where knowledge about how to solve a task partially resides in the computer system. The computer system can then help the user by doing certain actions himself or suggesting certain actions to the user. An inexperienced user's commands to the system will often be vague and maybe even incorrect given the user's real needs (Wærn 1999). A couple of applications areas that involve a lot of situations like the one mentioned above are *information filtering, intelligent tutoring* and *intelligent help*.

People today are flooded with massive amounts of information. They find it hard to extract the information which is relevant to them. *Information filtering* aims to find a structure in the available information. This structure can then be used to aid users in finding the information that is useful to them. In this case the intelligent interface models the user's reading patterns to determine what kind of information the user finds interesting. For instance a user might have the tendency to only read the sports section of the paper. An intelligent interface can filter out all the other parts of the paper, and point the user at other interesting sports information.

Tutors in general are programs that teach a user how a certain computer program works or how a certain real-life task has to be accomplished. *Intelligent tutors* can infer the user's understanding of the program from his performance on specific tasks. Based on this model of the user's understanding, the tutor can give the user advice on how to improve his performance. This can be done actively by suggesting other actions, or more passively by only answering the user's questions.

An *intelligent help system*, sometimes also referred to as an intelligent assistant, is very similar to an intelligent tutor. Only a help system is not there to teach the user something, but to help the user get a certain task done. Whereas a tutor gives the user specific tasks to accomplish in order to diagnose his understanding of the program, a help system lets the user plot his own course through the program. From the user's interactions with the program the help system

gathers information to aid the user. Just like intelligent tutors, help systems can also be either active or passive.

The I-Advisor will be an example of a passive intelligent help system. This decision was made to encourage users to discover their own way of using the AdaptIt tool. A tutor would not leave the user any freedom to discover the tool on his own.

3.2 Plan Recognition

When a customer in a CD store takes a CD out of a rack and walks to the cash register, we all automatically assume that the customer intends to buy the CD. The customer doesn't have to explicitly say that to the cashier. This process of inferring an intention from the actions of a person is called *plan recognition*.

Plan recognition can come in three different forms (Wærn 1996). When a person is aware of it and actively co-operates to make the plan recognition easier, this is called *intended plan recognition*. An example of intended plan recognition is when a user gives a command to a computer. The user explicitly states his intentions, making it easy for the computer to discover what he is supposed to do. The second form of plan recognition occurs when a person is unaware of or indifferent to the plan recognition. This is called *keyhole plan recognition*. An example of this is a cashier robot that has to learn how to behave as a cashier from interactions with customers such as the one described in the first paragraph of this section. By combining previous encounters with customers and extra world knowledge, such as how the cash register works, it can learn how to recognise the customer's plans. When the person is actively aware of the plan recognition and does his best to obstruct it, then you have *obstructed plan recognition*. In card games such as poker players will do their best to stop their opponents from recognising their plans.

In order to provide the AdaptIt user with the most appropriate help information, the I-Advisor will have to infer the user's intentions from his actions within AdaptIt. It is not hard to infer the intention behind low-level commands such as for example "Open Edit Menu" within a text editor. The tricky part, though, is to infer the higher goal behind that action. The user's higher-level goal might be to copy a certain piece of a text or he may want to undo his most recent action. Both of these intentions can be accomplished with commands that are located in that Edit menu. However if that user did not select a piece of text before issuing the "Open Edit Menu" command, you can safely assume that he did not open the edit menu to copy a piece of text.

If it had been the case that the user's interactions with AdaptIt were all the information the I-Advisor had, this would have been pure keyhole plan recognition. The I-Advisor would then have to compare the user's actions to all his previous interactions with AdaptIt, in the hope of finding typical patterns of actions he usually does. The problem with pure keyhole recognition is that most users do not follow optimal or even pre-planned paths through an application. Users will suddenly change their plans along the way, and decide to do something completely different. Relying solely on keyhole plan recognition would make recognising the user's intentions a difficult task for the I-Advisor.

The I-Advisor's situation is a combination of keyhole plan recognition and intended plan recognition. The situation of the plan recognition within the I-Advisor is shown in figure 3.1. The user does an action in the AdaptIt tool. This action is observed by the I-Advisor, and using plan recognition the I-Advisor will attempt to discover the user's current plan. So far this is a classic example of keyhole plan recognition.

The task model is what facilitates plan recognition considerably. This is because the task model is learned with pure intended plan recognition from the actions of experienced users of the AdaptIt tool. These previous users will explicitly state their intentions while performing their actions in the AdaptIt tool. For example the customer who takes a CD out of a rack will add to that action that his intention during this action was to buy the CD. The task model is then learned using the logs of the users' actions, as well as their intentions during these actions. The I-Advisor will compare the current user's actions to the examples of previous user's actions in the task model. If these actions are the same or almost the same, it can retrieve the user's intentions from the task model.

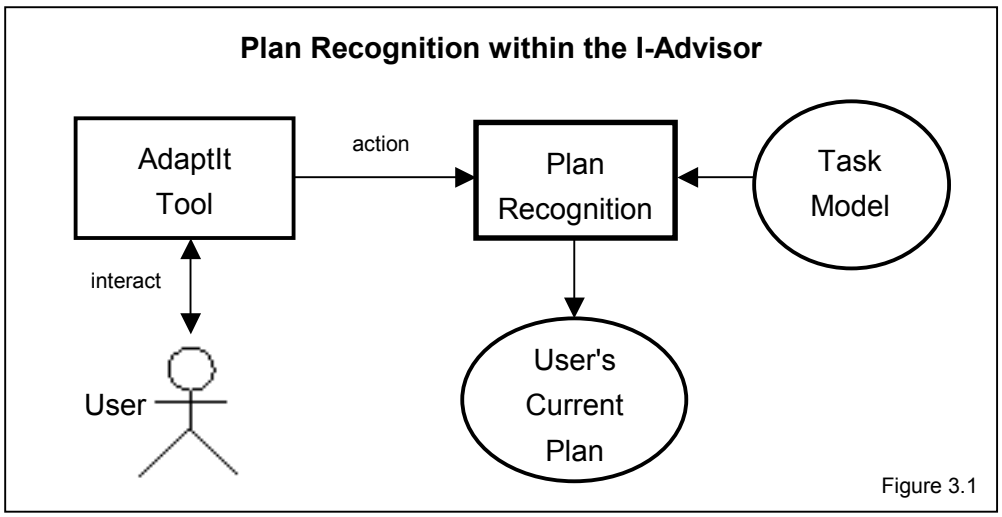

Figure 3.1

3.3 Possible Architectures for Intelligent Help Systems

During my search for possible architectures which were used for intelligent help systems two architectures were mentioned often. The first one was the one used in the most well known example of an intelligent assistant, the Office Assistant. This intelligent assistant in the form of a paperclip is located in most Microsoft Office applications. The Office Assistant uses Bayesian networks to infer the user's needs (Horvitz, 1998).

The other possibility was an object-oriented middleware for building collaborative interface agents called Collagen (Rich, Sidner & Lesh, 2001). Collagen is an abbreviation that stands for COLLaborative AGENt. Collagen assumes that a user and an agent collaborate and thus co-ordinate their actions in order to achieve shared goals. In our case the I-Advisor and the user of the AdaptIt application collaborate to successfully design a training.

Both Collagen and Bayesian networks seemed appropriate for the I-Advisor. However the Collagen approach had a slight advantage. Statistical approaches such as Bayesian networks need a lot of logs from example users to be effective. There were not that many logs of experienced test users available. This is caused primarily by the fact that the AdaptIt tool is still being developed and there are thus not that many training designers experienced at using it. Furthermore Collagen is less of a black box than the Bayesian network. It is easier to determine why it gives a certain advice, whereas in a Bayesian network it is often hard to discover why a probability has a certain value. This is caused by the large amount of computations involved in calculating a certain probability due to the way the probabilities in a Bayesian network influence each other. Last but not least, Collagen was very well documented (see http://www.merl.com/projects/collagen/), making it the most logical choice.

3.4 Collagen

Collagen is based on a theory called the collaborative discourse theory. When two or more people (or agents) collaborate, they need to communicate to co-ordinate their actions to achieve the shared goals. This communication is called the *discourse*. The collaborative discourse theory is based on research about how people collaborate, and in Collagen this theory is applied to human-computer interaction. The collaborative interface agent mimics the relationships that typically hold when two humans collaborate on a task involving a shared artefact.

3.4.1 Discourse State

Collagen makes use of a *discourse state* to follow and store the state of the discourse during a collaboration (Lesh, Rich & Sidner, 1999). This state is a model of the status of the collaborative tasks and the conversation about them. The discourse state is made up of two parts, a *focus stack* and a *plan tree*.

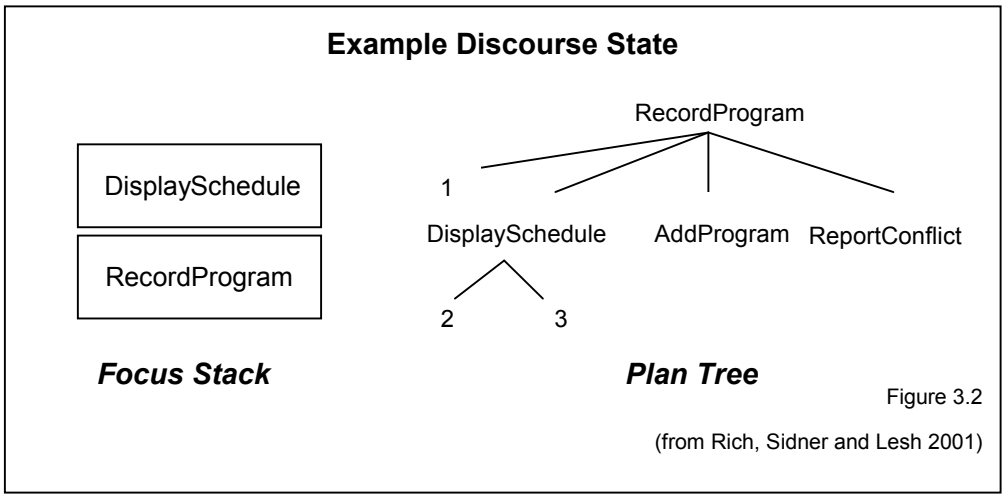

Figure 3.2

(from Rich, Sidner and Lesh 2001)

The focus stack consists of a stack of goals and for each goal on the stack there is an element in the plan tree. The goal on top of the focus stack is the **current purpose** of the discourse. Figure 3.2 shows an example discourse state, with a focus stack and a plan tree. In this case the current purpose on top of the focus stack is called Display Schedule. A plan tree in Collagen is an encoding of a partial SharedPlan between the user and the agent. A SharedPlan is a formal representation of the mutual beliefs about the goals and actions to be performed in a collaboration (Grosz & Sidner, 1986).

After every action performed by the user in the AdaptIt tool the discourse state is updated, as shown in figure 3.3. Suppose I-Advisor observes the user while performing an action A. The I-Advisor will use the task model to update the discourse state so that it explains action A. If the I-Advisor successfully recognises the user's plan, a new slide is built. This slide contains the current purpose the I-Advisor has learned that the user is working on. Chapter 6 will describe in detail the discourse state and how it is updated with the help of a simple example.

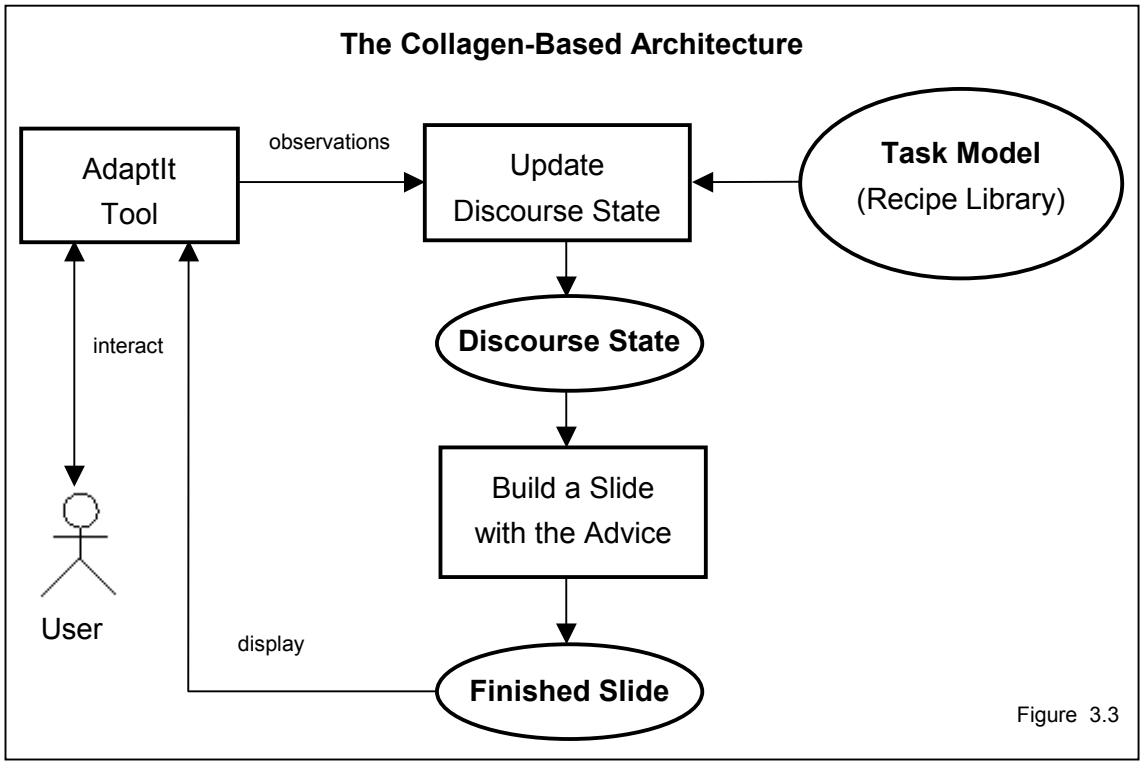

Figure 3.3

3.4.2 Task Model

The task model plays a very important role in plan recognition. This section will give an overview of what a task model actually consists of. All the terms which are introduced here will be explained in more detail in chapter 5.

The task model in Collagen consists of *actions* and *recipes* (Garland & Lesh, 2001). Sometimes the task model is referred to as the recipe library. Actions can be either primitive actions or non-primitive actions, also known as intermediate goals. *Primitive* actions can be executed directly within an application, such as for example "Click On Save Button". *Non-primitive* actions can only be achieved indirectly by achieving other actions. Examples of non-primitive actions are "Buy a CD" and "Do Christmas Shopping".

Recipes describe a set of *steps* that can be performed to achieve a non-primitive action. It is possible to have several different recipes to achieve a single non-primitive action. The steps in the recipe can be both primitive actions and non-primitive actions. Recipes can also contain constraints on the temporal ordering of the steps, as well as logical relations between the parameters, such as equality relationships. Figure 3.4 shows an example recipe for recording a program on a VCR. The steps in this recipe are DisplaySchedule and AddProgram, along with an optional step ReportConflict.

An Example Recipe

```
public recipe RecordRecipe achieve RecordProgram {
    step DisplaySchedule display ;
    step AddProgram add ;
    optional step ReportConflict report ;
    constraints {
        display precedes add ;
        add precedes report ;
        add.program == achieves.program ;
        report.program == achieves.program ;
        report.conflict == add.conflict ;
    }
}
```

Figure 3.4

(from Rich, Sidner and Lesh 2001)

3.4.3 Why Collagen Was Not Used

The Mitsubishi Electric Research Laboratories (MERL) who created Collagen were so kind as to provide us with the Collagen code. Although this was appreciated, it was decided to only use their code as an example for the I-Advisor. The main reason for this was that the I-Advisor was in a quite different situation than the one Collagen was originally intended for. Collagen assumes that both the agent and the user can and will do actions within the application. The I-Advisor will however not perform any actions in the AdaptIt application. This would require quite a lot of changes in the Collagen code. Therefore we decided to build a simplified version of Collagen based on their articles. This simplified version of Collagen could easily be modified to better fit the I-Advisor's situation.

4 Requirements for the I-Advisor

The main requirement for the I-Advisor is that it must be able to give a user of the AdaptIt application context-sensitive support. This support is meant to aid him in learning how to use the AdaptIt application.

4.1 Functional Requirements

(1) The support given by the I-Advisor shall consist of links to the pages in the help system corresponding to the user's current task context.

(2) This support given by the I-Advisor shall be based on:
- the user's actions within AdaptIt.
- a task model.

(3) The I-Advisor shall contain a task model, which describes a part of the AdaptIt application.

(4) A domain expert shall be able to view and annotate the logs of previous users, in order to aid the learning of the task model.

4.2 Technical Requirements

(5) The I-Advisor shall be implemented in Java just like the AdaptIt tool.

(6) The I-Advisor shall operate within a small part of the AdaptIt application.

(7) No changes within the original AdaptIt application shall be necessary to link the I-Advisor to the AdaptIt application.

(8) The I-Advisor shall interact with the AdaptIt application in the same way as the Q-Advisor and the Advisor.

4.3 Possible Future Expansions

(1) The support given by the I-Advisor should consist of a list of the advised next steps.

(2) The I-Advisor should be able to operate within other parts of the AdaptIt application.

(3) The I-Advisor should give an adjustable amount of assistance, such as for example making the I-Advisor more active / passive.

(4) The I-Advisor should be able to cope with and adapt to changes in the Help System.

5 Task Model

With the help of some examples this chapter will explain the basics of how a task model is learned by the I-Advisor. Section 5.1 describes how the user's actions are logged and explains the structure of the actual task model. How a domain expert adds certain knowledge to the task model by *annotating* the logs of the user's actions is explained in section 5.2. Sections 5.3 and 5.4 explain the two phases of the learning of the task model, namely *alignment* and *induction*. The last section concludes with an example of a finished task model. The way a task model is learned is based on Collagen (Garland & Lesh, 2001). Whenever the I-Advisor differs from Collagen, this will be stated explicitly.

The task model is learned using previous users' interactions and added domain expert's knowledge. At the moment the previous users are domain experts. This decision was made to facilitate the learning of the task model, since a domain expert has more knowledge of the application and thus has more structure in his actions. New users will not have enough structure in their actions to learn a useful task model from. Once a user becomes more experienced at using the application, it is possible to also base the task model on his actions.

An Example Log Consisting of 13 Actions:

1) Action **a** (1)	8) Action **b** (6)
2) Action **b** (2)	9) Action **b** (5)
3) Action **c** (1)	10) Action **g** (3)
4) Action **d** (1)	11) Action **e** (4)
5) Action **e** (1)	12) Action **a** (5)
6) Action **f** (2)	13) Action **f** (4)
7) Action **a** (3)	

Figure 5.1

5.1 Task Model Basics

Figure 5.1 contains an example log of 13 different actions performed by a user. The actions are all represented by two variables, namely the *action type* and the *parameter*. The first action in our example log is of action type **a** and has as parameter **1**. This signifies that the user performed an action of type **a** upon the object **1**.

The learning of the task model takes place in two phases. In order to explain these phases, it first has to be explained what a task model consists of. A task model is made up of *actions* and *recipes*. Actions can be either primitive actions or non-primitive actions. *Primitive actions* are

actions that can be executed directly within an application, such as clicking on a button. The actions shown in figure 5.1 are all primitive actions. *Non-primitive actions* can only be achieved indirectly by achieving other actions. An example of a non-primitive action can be writing a letter to someone, an action which can only be accomplished by performing a lot of different primitive actions, such as typing a letter or hitting the space bar. All the examples in this thesis will use capital letters to signify non-primitive actions and lower-case letters for primitive actions.

Recipes describe all the different ways a non-primitive action can be achieved. They decompose the non-primitive action into sub goals, which are called *steps*. These steps can be either non-primitive or primitive actions. Figure 5.2 shows an example recipe for the non-primitive action **A**, consisting of 4 steps. To achieve **A** one must first perform the non-primitive actions **B** and **D**, followed twice by the primitive action **b**. Steps of a recipe can be optional, like the first and third step in our example recipe. It is thus also possible to achieve **A** by performing first **D** and then **b**. There can also be equality relations among the parameters. The equality relation in our example is for the third and fourth step. This means that the parameters for both of these steps have to be equal for the action **A** to be achieved.

An Example Recipe for A:

(1) Step **B** Optional
(2) Step **D**
(3) Step **b** Optional
(4) Step **b**
Equality: step **3** and **4**

Figure 5.2

The learning of the task model consists of two phases. The first phase is called *alignment* and the second phase *induction*. In the alignment phase simplified recipes are created without optional steps and without equality relations among the steps. In this phase all the user's actions are mapped to steps in a recipe. The optional steps and equality relations are both learned in the induction phase with the help of the mappings from the alignment phase. Section 5.4 will explain this in more detail.

5.2 Annotating Logs

Before we can start with the alignment phase on the example log from figure 5.1 we need to know which actions are steps in achieving the same non-primitive action. We call a sequence of actions which achieve the same non-primitive action a *segment*. The process of indicating which segments of actions contribute to the same non-primitive action is called *annotating* the logs of actions. It is possible to learn which actions belong to a non-primitive action by using statistical learning methods. In this case all the actions which occur together regularly are put in the same non-primitive actions. We however would like the structure of the non-primitive actions to be similar to the hierarchy in the pages of the Advisor. This is to make it easier to link the I-Advisor's advice to the corresponding pages in the Advisor. Furthermore statistical methods typically require lots of test data, which we do not have at our disposal. They also tend to ignore prior domain knowledge. Therefore a domain expert annotates the logs of the user's actions.

In figure 5.3 a domain expert has annotated the user's actions from figure 5.1. All the actions together form a segment contributing to the non-primitive action **G**. **G** is made up of two sub goals of type **H**. At the lowest level there are two segments of actions which can be performed to achieve the non-primitive action **J**. The first segment of actions which achieves **J** consists of **e** (1) and **f** (2). The second segment consists of the actions **e** (4), **a** (5) and **f** (4). There are also two ways to achieve the non-primitive action **I**. The first segment of actions that achieves **I** is made up of **a** (1), **b** (2) and **c** (1). The second segment consists of **a** (3), **b** (6) and **b** (5).

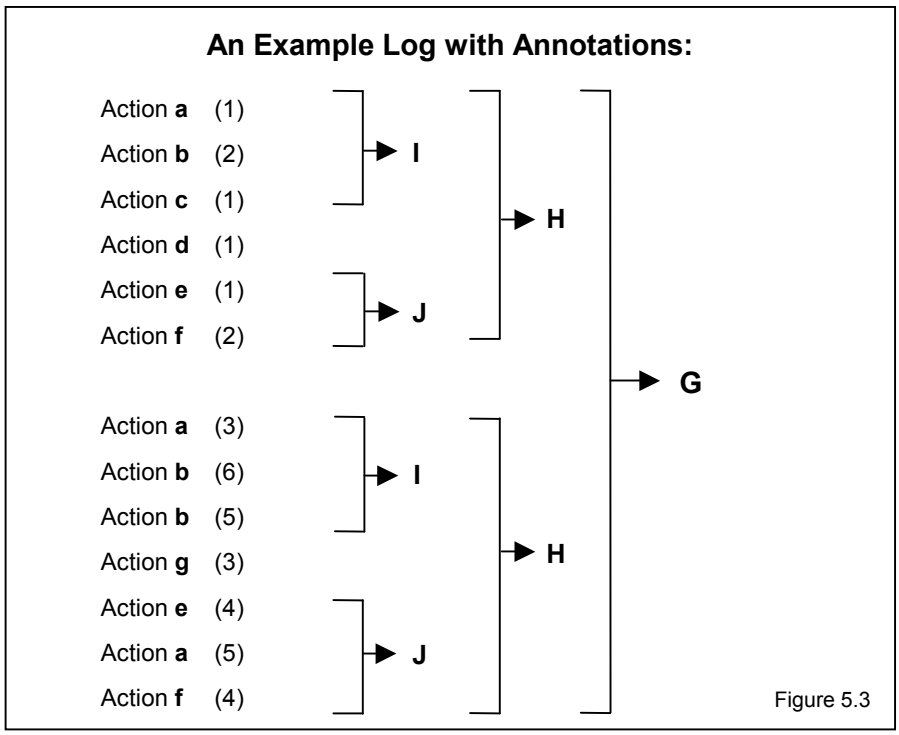

Figure 5.3

The fact that the domain expert adds the hierarchy to the actions has the disadvantage that the task model cannot be learned automatically anymore. For every log of a certain user's actions added to the task model, a domain expert has to annotate those user's actions. This makes refining the task model by basing it on more logs more difficult. It might be possible to later on implement a tool to make it easier to add a hierarchy to the actions. This tool can for instance show a user the existing hierarchy of the task model up to now, as well as all the possible non-primitive actions a primitive action can be a part of. Possibly this tool can make annotating the actions so simple that users can do it themselves without the help of a domain expert.

5.3 Alignment

The first step of the alignment phase consists of sorting all the annotated segments of the same type into sets of segments. As shown in figure 5.4, both of the segments of type **I** are placed in one segment set. The method also checks if each new segment which is to be added to an existing segment set is already a subset of the segments in the set. This is to ensure that segments which are similar are mapped to the same recipe.

The segments are sorted in the set by their diversity, or how many different types of actions they consist of. The lowest diversity for a segment is 1, meaning it contains actions of only one type. The highest possible diversity is equal to the total number of actions in a segment. This signifies that there are no actions in this segment of the same type. In figure 5.4 the first segment in the segment set of type **I** has the highest diversity, i.e. a diversity of 3. The second segment's diversity is only 2, since there are two primitive actions of type **b** in this segment. A new segment which is to be added to the set is only checked against the first and most diverse segment. Due to the sorting of the segments in the set by diversity, all other segments in the segment set are subsets of the first. A new segment thus does not have to be checked against all the different segments in the set.

The subset relation is used differently in the I-Advisor than it usually is. The I-Advisor only looks at the different types contained in a segment, it does not check how many times the different types appear. The second segment in figure 5.4 of type **I** is thus a subset of the first using the I-Advisor’s subset relation, because both the actions **a** and **b** appear in the first segment. It does not matter that the action **b** appears twice in the second segment and only once in the first.

The diversity of a new segment can be higher or lower than or equal to the diversity of the first segment of a certain segment set. If the diversity of a new segment is lower or equal, then it is

The Segments Sets After Alignment Step 1

Segment Set 1 of **Type G:**
Segment 1:
Segment **H** Segment **H**
diversity = 1

Segment Set 2 of **Type H:**
Segment 1:
Segment **I** Action **d** (1) Segment **J**
diversity = 3

Segment Set 3 of **Type H:**
Segment 1:
Segment **I** Action **g** (3) Segment **J**
diversity = 3

Segment Set 4 of **Type I:**	
Segment 1:	*Segment 2:*
Action **a** (1) Action **b** (2) Action **c** (1)	Action **a** (3) Action **b** (6) Action **b** (5)
diversity = 3	*diversity = 2*

Segment Set 5 of **Type J:**	
Segment 1:	*Segment 2:*
Action **e** (4) Action **a** (5) Action **f** (4)	Action **e** (1) Action **f** (2)
diversity = 3	*diversity = 2*

Figure 5.4

checked whether this new segment is a subset of the first segment. When this check succeeds, the new segment is added to the set. Otherwise a new segment set is created. In the case that a new segment has a higher diversity than the first segment, then it is checked if the first segment is a subset of the new segment. If this is true, the new segment becomes the first segment of the set. A new segment set is once again created if this check fails.

Figure 5.4 shows the annotated example log from figure 5.3 after the first step of the alignment phase. The first segments of all the segment sets have the highest diversity and all the other segments are subsets of the first. In the case of the non-primitive action of type **H**, there are two separate segment sets. When it was checked whether the segment from segment set 3 was a subset of the segment from segment set 2, the subset relation failed. This was caused by the fact that the action **g** is not located in the other segment. Thus a new segment set is created for this segment.

The diversity is not based on Collagen. It is something which was introduced to simplify the use of the subset test. By using the diversity to sort the segments in a set, a new segment does not have to be tested against all the different segments in a set.

The use of the subset relation to put the segments together into sets stems from Collagen (Garland & Lesh, 2001). It helps us gather the segments together with the same action types. If you look at the example from figure 5.4, you might say that it is possible to make one segment set for the non-primitive action **H**. It is however possible to capture all combinations of action types in one recipe. In that case though you will learn large recipes with a lot of optional steps. We would like to learn short, simple recipes which capture the similarities between the segments.

The second and final step of the alignment phase is the creation of the actual simplified recipes. These simplified recipes have only required steps and no equality relations. For each set of segments a recipe is created which is equal to the union of the actions in all the segments of the set. All the actions in the different segments are mapped to the steps of this recipe.

This is explained using the segment sets of **I** and **J** from figure 5.4. Figure 5.5 consists of the recipes created from these segment sets. The recipe is initialised to be equal to the actions in the segment with the highest diversity of the segment set. The recipe for the segment set of type **I** in our example is initialised to be equal to the first segment. It will thus contain three steps, one for each action in the segment.

The Recipes After Alignment Step 2

Type I

Segment 1	**Mapped to:**	*Segment 2*	**Mapped to:**	**The Recipe:**
Action **a** (1) →	Step 0	Action **a** (3) →	Step 0	Step 0 → Action **a**
Action **b** (2) →	Step 1	Action **b** (6) →	Step 1	Step 1 → Action **b**
Action **c** (1) →	Step 3	Action **b** (5) →	Step 2	Step 2 → Action **b**
				Step 3 → Action **c**

Type J

Segment 1	**Mapped to:**	*Segment 2*	**Mapped to:**	**The Recipe:**
Action **e** (4) →	Step 0	Action **e** (1) →	Step 0	Step 0 → Action **e**
Action **a** (5) →	Step 1	Action **f** (2) →	Step 2	Step 1 → Action **a**
Action **f** (4) →	Step 2			Step 2 → Action **f**

Figure 5.5

Then we go through all the other segments of the set and check if there is a matching step in the recipe for each of the actions in the segment. In the case of segment 2 in the segment set for **I**, there is no step in the recipe for the third action, namely **b** (5). Therefore a step is inserted after the step the preceding action was mapped to. The preceding action, in this case **b** (6), was mapped to step 1. Thus the new step for the action **b** (5) is inserted after step 1.

The resulting recipes are passed on to the induction phase, along with the segment sets and all the mappings between the actions in the segments and the steps of the recipes.

5.4 Induction

The induction phase consists of learning the optional steps and the equality relations between the parameters in the recipes. I will go over both steps briefly, since the I-Advisor does not use the optional steps and the equality relations. It can learn them however, in the same way as Collagen does. Chapter 7 explains why the I-Advisor does not use them.

To discover which steps are optional, the mappings learned earlier in the alignment phase are used. If a segment in a certain segment set exists which does not have an action mapped to a step of the corresponding recipe, then that step is marked as optional. In figure 5.6 step 1 is marked as optional, since the second segment has no action mapped to that step.

Induction of the Optional Steps

Type J

Segment 1	**Mapped to:**	*Segment 2*	**Mapped to:**	**The Recipe:**
Action **e** (4) →	Step **0**	Action **e** (1) →	Step **0**	Step **0** → Action **e**
Action **a** (5) →	Step **1**			Step **1** → Action **a**
Action **f** (4) →	Step **2**	Action **f** (2) →	Step **2**	Step **2** → Action **f**
				Optional: Step 1

Figure 5.6

The induction of the equality relations between the parameters is shown in figure 5.7. We look at each action of a certain segment, and remember which parameters are equal to each other. Each discovered equality is then tested against all the other segments before it is added to the recipe. In figure 5.7 an equality is found in the first segment between step 0 and step 2. The parameters of the actions mapped to step 0 and step 2 in the second segment are not equal, however. The equality relation between step 0 and step 2 is thus not added to the recipe.

Induction of the Equality Relations

Type J

Segment 1	**Mapped to:**	*Segment 2*	**Mapped to:**	**The Recipe:**
Action **e** (4) →	Step 0	Action **e** (1) →	Step 0	Step 0 → Action **e**
Action **a** (5) →	Step 1	Action **f** (2) →	Step 2	Step 1 → Action **a**
Action **f** (4) →	Step 2			Step 2 → Action **f**
Equalities:		**Equalities:**		**Equalities:**
(Step 0, Step 2)		**none**		**none**

Figure 5.7

5.5 Finished Task Model

The complete task model is displayed in figure 5.8. For each non-primitive action there is now a recipe which successfully describes all the segments in the set of segments matched to that recipe. For the non-primitive action **H** two recipes were necessary, all the other non-primitive actions have been described using one recipe. Only the recipe for **I** has an equality relation, namely for step 0 and step 3. This means that the parameters of step 0 and 3 have to be equal for the user to successfully achieve **I**.

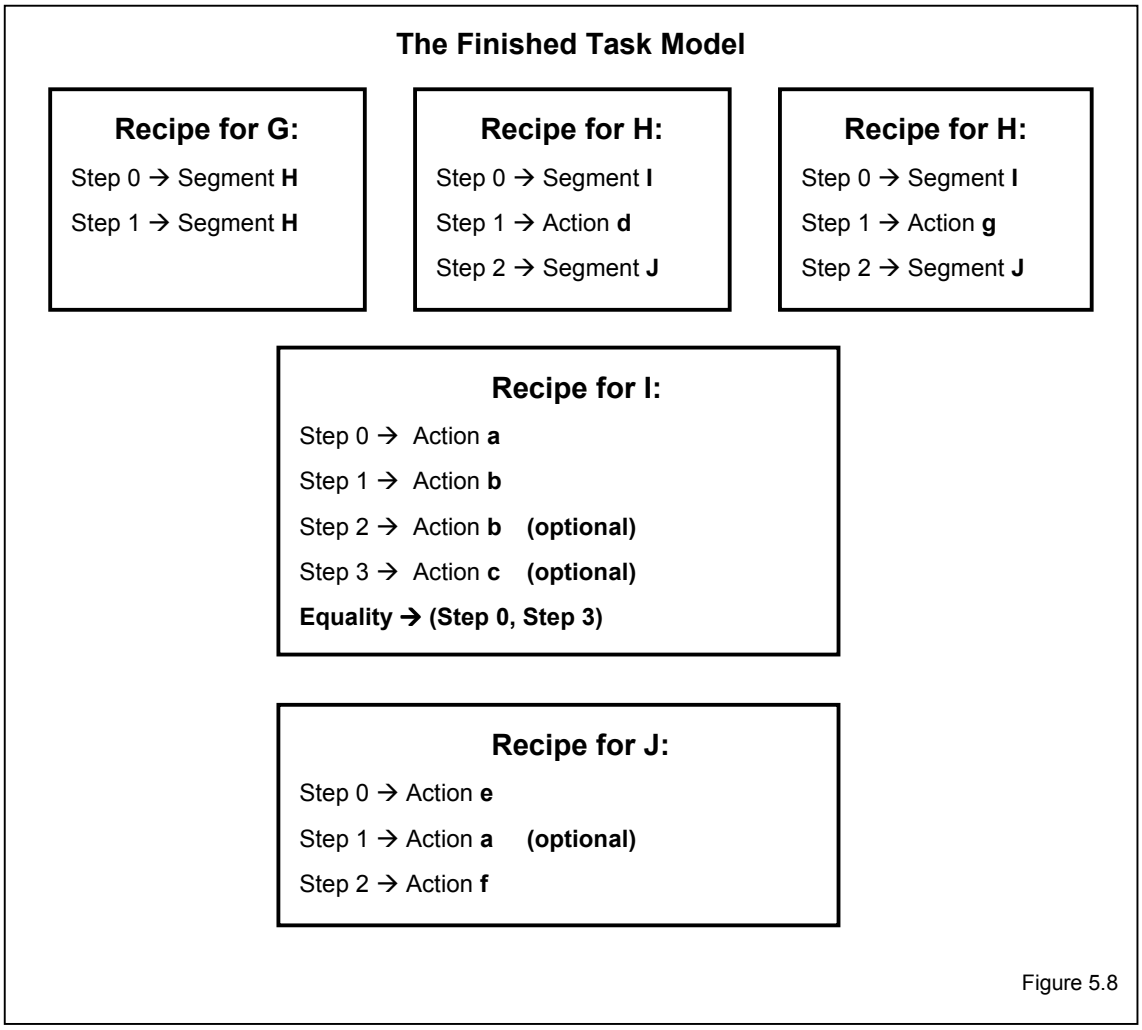

Figure 5.8

6 User Model

The user model that the I-Advisor maintains is equal to the *discourse state* used in Collagen. The discourse state can used as a user model in AdaptIt, because only the user will perform actions in the AdaptIt tool. The I-Advisor will only advise possible actions, it will never perform them on its own. This section will discuss the discourse state again, since it is used differently in the I-Advisor.

The focus stack is a stack of goals. The goal on top of the focus stack is the user's *current purpose* or *current focus*. The focus stack is used to store the non-primitive actions the user wants to accomplish with his primitive actions. On the bottom of the stack is the most general goal. The current focus on top of the stack is the most specific goal the user is working on at that moment. In the example discourse state in figure 6.1 the current purpose is **D**. The most general goal on the focus stack is **A**.

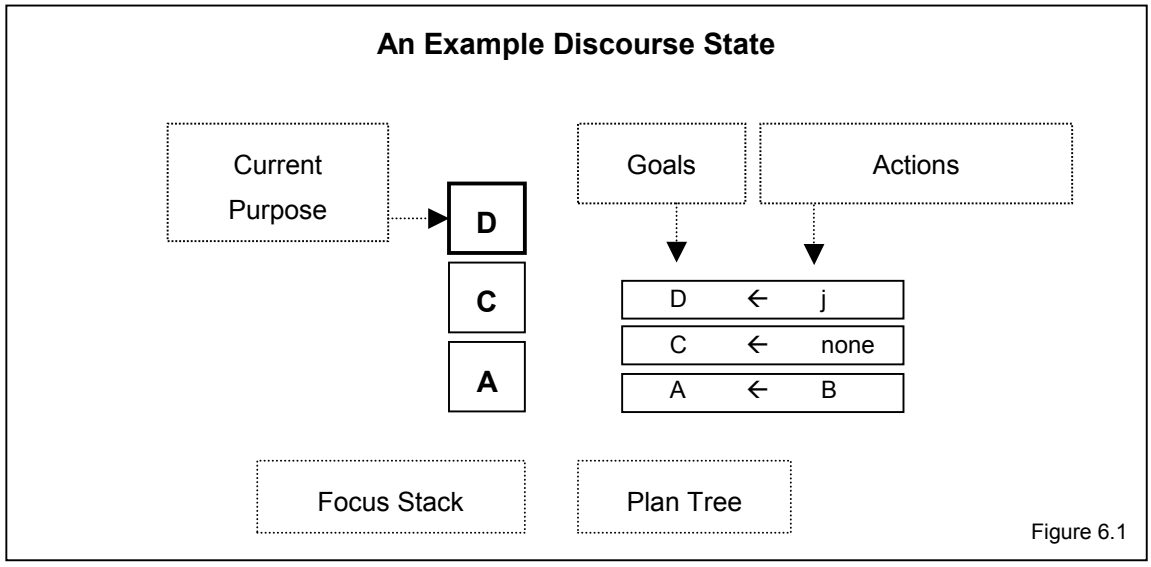

Figure 6.1

The plan tree is used to remember which of the user's actions contributed to which goals. It should be seen as a history explaining the user's previous actions. Figure 6.1 contains an example plan tree on the right. A plan tree consists of two columns: a goals column and an actions column. For each goal currently on the focus stack there is an element in the goals column of the plan tree. These are all the goals the user is currently working on. The list in the second column stores all the actions performed by the user that contributed to these goals. For instance if one looks up the goal **D** in the first column, the second column shows that the primitive action **j** contributed to this goal. The goal on the bottom of the stack, i.e. **A**, has the

non-primitive action **B** which contributed to it. This means that the user achieved the non-primitive action **B** earlier with his actions. The example task model in figure 6.2 shows that **B** is a step in achieving **A**.

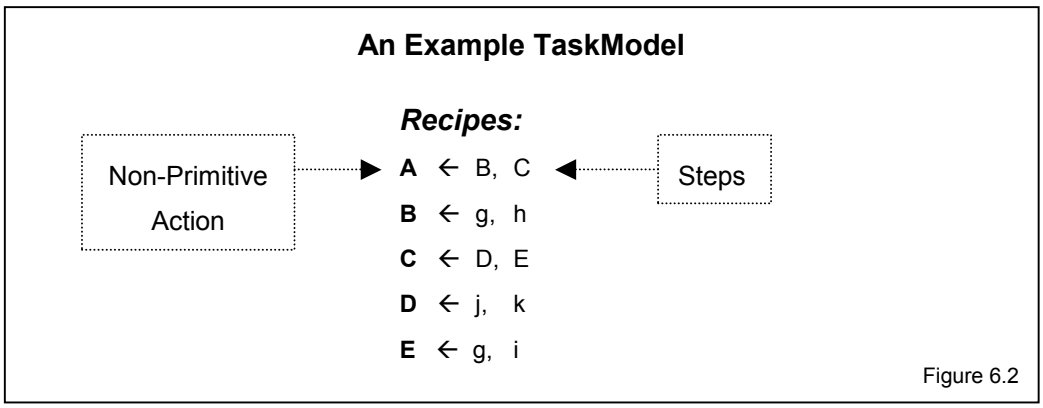

Figure 6.2

Every time the user performs an action, the discourse state is updated to include that action. The I-Advisor searches through the recipes in the task model starting at the current focus to discover which goals that specific action can contribute to. This is done by looking up the recipe for the current focus. The I-Advisor will search through the steps of this recipe to check if the user's action is a step in the recipe. The I-Advisor will start searching from the current focus to find the user's action, because it has deduced from the user's earlier actions that the user is working on that specific goal. That goal is thus the most likely place to find the new action. This narrows down the search space, making it easier to find a user's new actions.

If the current discourse state is the one shown in figure 6.1, the I-Advisor will search for a user's new action in the task model starting at **D**. It will retrieve the recipe for **D** from the task model. This recipe, displayed in figure 6.2, consists of the steps **j** and **k**. It will start searching at **D** from action **j** forward, since action **j** is the last action that contributed to the goal **D**. This can be learned from the plan tree.

Once the user's action has been found in the task model, the focus stack and plan tree are updated to include that action. The following paragraphs describe how the I-Advisor searches through the task model for the user's last primitive action and updates the discourse state accordingly. This is explained using an example.

Figure 6.2 shows an example task model. This task model will be used to demonstrate the initialisation of the discourse state and how it is updated. The example task model is deterministic; there is only one recipe for each non-primitive action. The actual task model used in the I-Advisor is not deterministic. For the searching through the task model for actions it does

not matter whether a task model is deterministic or not. If there are multiple recipes for a non-primitive action, the I-Advisor will start by searching through the first recipe. When the first recipe does not contain the action, the other possible recipes are searched through.

When the discourse state is initialised, as in figure 6.3, the focus stack has one goal pushed onto it and the plan tree has one element corresponding to that goal. This main goal is the root node of the whole task model. In our example task model the root node is **A**. The one element in the plan tree for the root node is empty since the user has not performed any actions yet.

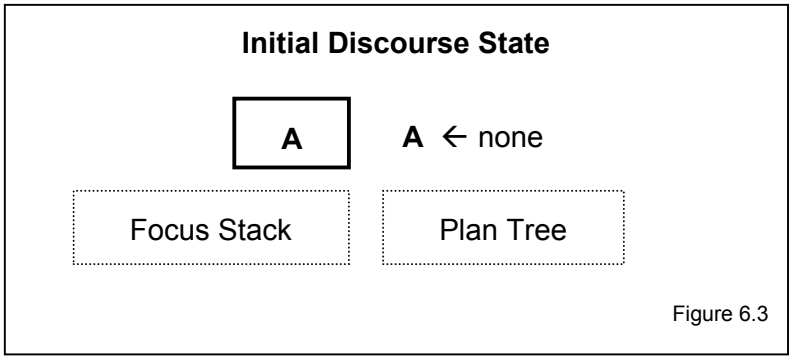

Figure 6.3

After the discourse state is initialised, the user performs the action **g**. The I-Advisor searches through the task model starting from the focus to find the action **g**. This search takes place in a depth-first fashion. Each time a new non-primitive action is found among the steps, this is immediately expanded and searched through. Figure 6.2 shows that the action **g** can be found in two different places in the task model. The primitive action **g** can be a step in achieving both the non-primitive actions **B** and **E**. At this moment there is no way to determine which of the two is the correct way to explain **g**. Thus there are two possible discourse states after this update, with **B** and **E** as the new current focuses, as displayed in figure 6.4. The two possible plan trees show that **g** can contribute to both **B** as well as **E**.

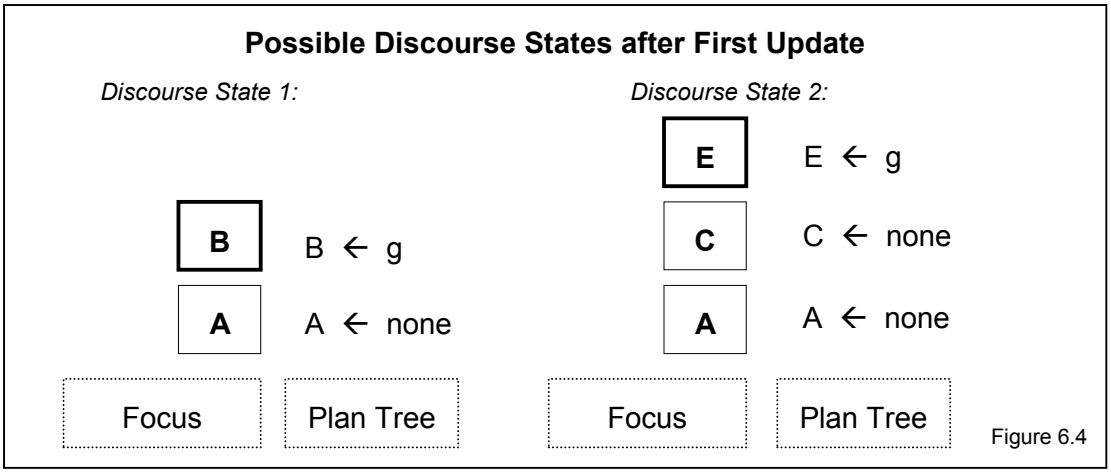

Figure 6.4

In order to narrow down the possible discourse states, the I-Advisor will look at the user's next actions. This is demonstrated with the user's next action, which in this case is **h**. The I-Advisor will go down the possible discourse states one by one and attempts to find the primitive action **h** in each of them. It searches through the task model starting from each current focus, from the last action which contributed to the current focus onward. This last action is retrieved from the plan tree, by looking which action was last added to the plan tree as contributing to the current focus. In the first possible discourse state, the I-Advisor starts searching at the current focus **B** from action **g** onward.

The search in the first possible discourse state is successful; action **h** is a step in achieving the current focus **B**. Action **h** is also the last step in achieving **B**, meaning that the sub goal **B** has been achieved. **B** can thus be removed from the focus stack. **B** is removed from the first column of the plan tree, since it is not a goal on the stack anymore. It is added to the second column of the plan tree, though, because it has been achieved and was a step in achieving the main goal **A**. The updated discourse state is shown in figure 6.5. Every time all the steps of a sub goal have been achieved, this sub goal is removed from the focus stack and added to the second column of the plan tree.

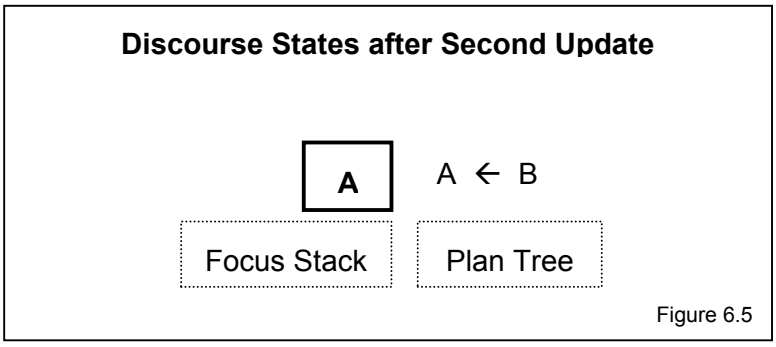

Figure 6.5

The second possible discourse state, with **E** as the current focus, is unable to find the action **h** below the focus. This possible discourse state is thus deleted, since it is not able to describe the user's action. A possible discourse state is deleted only if it is not the last possible discourse state left, meaning that there is always at least one possible discourse state.

It might happen that you have only one possible discourse state left and that you are unable to find the user's last action beneath that discourse state's current focus. This can happen for example when a user was distracted. He might then completely forget what he was doing. His new action will thus be completely unrelated to his previous actions. In this case the current focus is removed from the stack to broaden the search space. If the action is not found either then, the current focus at that time is also removed in the same way. This will continue until the

current focus is the main goal of the task model. This means that the whole task model is searched through for the action. Assuming that all of the user's possible actions are located in the task model, the action will now certainly be found.

In the example the I-Advisor is able to learn the user's current goal, i.e. **B**, after the user has performed two actions. He however had just achieved **B**, so the I-Advisor will give him advice on **B**'s parent goal, i.e. **A**. The user will now be working to achieve the goal **A**, because it is the new current focus after **B** was removed from the focus stack.

7 AdaptIt and the I-Advisor

The implementation of a prototype of the I-Advisor for the AdaptIt application can be divided into three main steps. First of all we had to decide how to listen to the AdaptIt application. It was important to choose which events were used to learn the task model and which details of these events were stored. Events are the changes in the AdaptIt tool caused by the user's actions. This will be discussed in section 7.1. Section 7.2 will go further into the details of the task model. It will show how the task model is learned and explain a part of the learned task model. The last section will describe the actual advising of the user. This section will try to answer questions such as how to advise the user and when exactly to provide him with advice.

7.1 Listening to AdaptIt

There are several ways to listen to an application and thus keep track of the user's activities. You can for instance log every movement of the mouse, or listen to all the input from the keyboard. We are however not interested in the exact path of the user's mouse cursor when it goes from one menu to another, we are only interested in which menu's the user has clicked on. The same goes for the keyboard. The exact text the user types in the text panels is not relevant, the important thing for the I-Advisor to know is which attribute of an object the user has changed by typing this text.

There are examples imaginable where the exact place of the mouse cursor is indeed important for an advisor. For instance when a user's mouse cursor has been hovering over a certain object for a long time, an advisor might want to give him some advice on that object. However in that case you need to regularly check where the mouse is, so that you know when the cursor has been standing still for a long time. In other words you get a lot of very detailed events, and in only a few cases, such as the one described above, they can help you give useful advice.

7.1.1 Types of Events

The AdaptIt application is programmed in JAVA. This programming language uses the *Model View Controller* structure to handle GUI events. The *model* includes all the data objects which together store the application state. The *view* consists of all the Graphical User Interface (GUI) elements which provide the user with a view of the data objects. The *controller* translates the user's interactions with the view into actions to be performed by the model.

The AdaptIt application provides a good framework for listening to both the model events and the controller events. The model events come in the form of *object update events*. The object

update events are fired when the user changes an attribute of a data object, for instance by giving a data object a new name or adding some text to describe it. These events are handled by the Object Update Manager. The controller events which one can listen to are called *selection events*. A selection event is passed whenever the user clicks on a data object in the application. These are handled by the Selection Controller.

Although using this framework makes it easy to implement the listeners, it has one major drawback. The view events are ignored by these two listeners. View events are for instance selecting a tabbed pane or opening a new menu. These events only change the view, thus no object update events or selection events are fired. This is a logical choice the programmers made which is a consequence of the Model View Controller structure used in JAVA. The view events would have been helpful though to discover what a user wants to do. In order to listen in on these events too, a specific listener needed to be written for each different type of GUI event. During the project there was not enough time to write all these listeners, therefore only the selection events and object updates were used for the I-Advisor. Another reason was that one of the requirements (Technical Requirement 7) was that no changes would be made in the AdaptIt application. Implementing these listeners and their specific events would have required a lot of changes in the application.

7.1.2 Properties of the Events

The selection events and the object update events have nearly the same form. They both consist of three properties which describe the event. The first property is the name of the object which is being updated or selected. This object is of a certain class and this is stored as the second property. The last property is the only one in which the selection event differs from the object updates. For selection events the last property is the editor panel in which the object is selected. For object updates this is the source of the update, i.e. the handler which processes the object's attribute change. From now on I will refer to this third property, the source of the update or the editor panel, as the source, since this is the source of the selection event or object update.

Figure 7.1 and figure 7.2 show an example selection event and an example object update. The example selection event in figure 7.1 is fired when the user clicks upon a Skill in the Skill Hierarchy Diagram which is called Driving. The name of the object selected in this case is Driving and the class of this object is LFSkill. The source is the SkillHierarchy Diagram. The example object update in figure 7.2 takes place when the user gives a Skill the new name Driving in the BasicAnalysisElementPanel. The BasicAnalysisElementPanel is the name of the text panel where the new name of the Skill is entered. In this case the name of the object updated is Driving and the class of this object is once again an LFSkill. The source is the BasicAnalysisElementPanel.

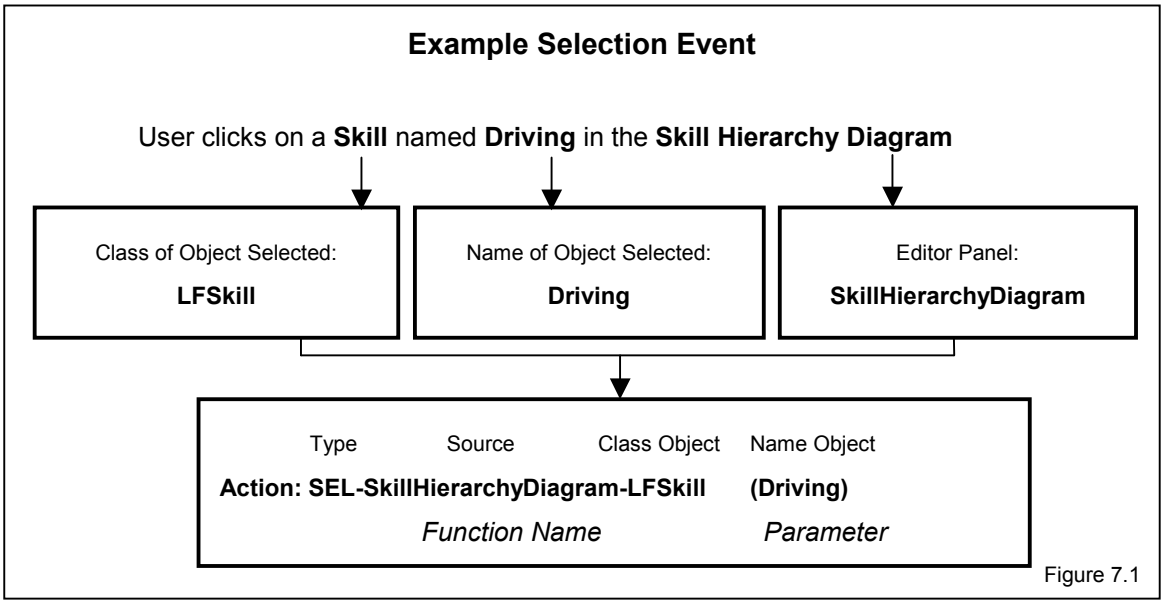

Figure 7.1

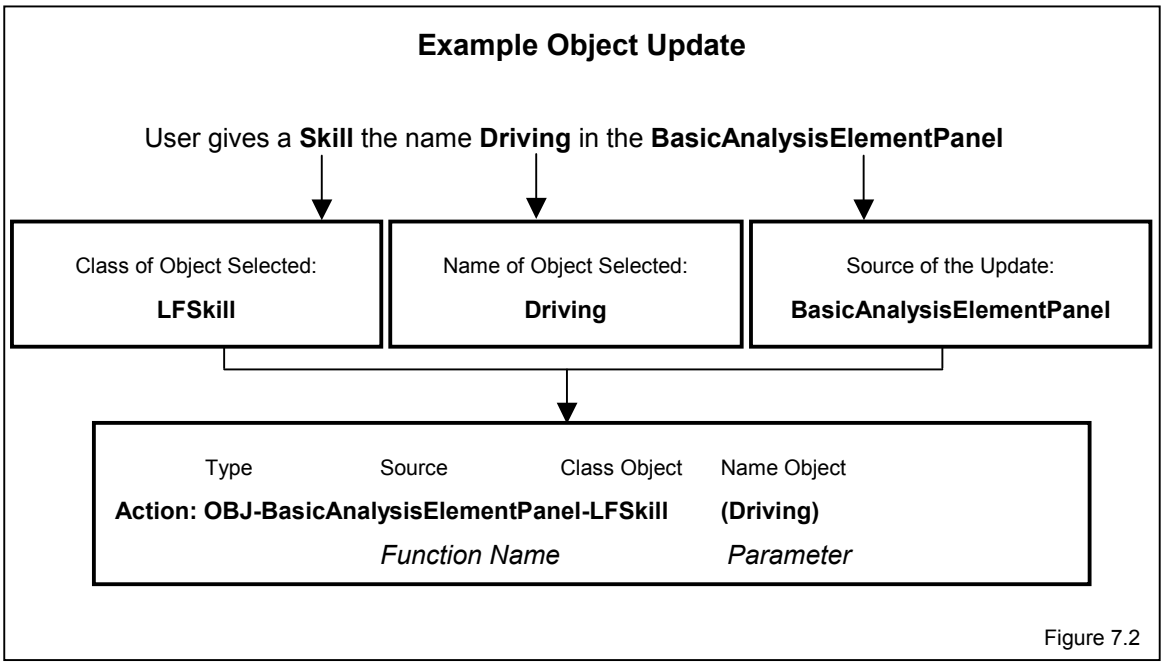

Figure 7.2

The figures 7.1 and 7.2 also show at the bottom how the events are stored by the I-Advisor. The source and the class of the object are combined into a function name with one parameter. This parameter is the name of the object. The function name also has the type of the event, either selection (SEL) or object update (OBJ) attached to the front of it, to make it easier for the domain expert to determine what kind of event it is. It has no other use, because the sets of the selection and object update events are disjoint. The name of the object was chosen as the parameter, since it was the only one of the properties which can vary considerably. Both the

class of the object and the source do not have enough possible values to be interesting as a parameter.

Figure 7.3 shows an example log of a user performing actions within AdaptIt. This log consists of four selection events and four object updates. For instance action 5 and 6 indicate that the user selected a new skill in the SkillHierarchyDiagram. For action 5 the skill selected is named Braking, and for action 6 this skill is called Steering. A log such as this will be used to learn the task model in the next section.

Example Log of the User's Actions

Action **1** Function="**SEL-ProjectBrowser-LFTrainingProject**"
Parameter="**DrivingLessons**"
Action **2** Function="**SEL-SkillHierarchyDiagram-LFSkill**"
Parameter="**Shifting Gears**"
Action **3** Function="**OBJ-LASkillHierarchyHandler-LFSkillHierarchy**"
Parameter="**DrivingHierarchy**"
Action **4** Function="**OBJ-LASkillHierarchyHandler-LFSkillHierarchy**"
Parameter="**DrivingHierarchy**"
Action **5** Function="**SEL-SkillHierarchyDiagram-LFSkill**"
Parameter="**Braking**"
Action **6** Function="**SEL-SkillHierarchyDiagram-LFSkill**"
Parameter="**Steering**"
Action **7** Function="**OBJ-LASkillHierarchyHandler-LFSkillHierarchy**"
Parameter="**DrivingHierarchy**"
Action **8** Function="**OBJ-LASkillHierarchyHandler-LFSkillHierarchy**"
Parameter="**DrivingHierarchy**"

Figure 7.3

7.2 Learning the Task Model: Practical Issues

The way the task model is learned was explained extensively in chapter 5. There are a few choices still to be made, such as which part of the AdaptIt application to use, what hierarchy is used in the task model, and how many actions are used to learn the model. It is also important to decide whose actions to learn the task model from. Is it possible to refine a task model with the user's actions or is it best to only use a domain expert's actions? These things are all discussed in this section.

First of all a choice had to be made which part of the AdaptIt application would be used to test the prototype in. As described in the requirements, the I-Advisor was first built for a small part of the tool. If these tests were successful, the I-Advisor's task model could always be expanded to include the whole AdaptIt application. The AdaptIt tool is divided up into two possible domains, i.e. the skill hierarchy and the blueprint domain. The skill hierarchy domain was chosen to start with since this domain had the most clearly defined order in the sub goals. The sub goals in the blueprint domain could more easily be changed in order.

The tests of the prototype in the skill hierarchy were quite successful. Therefore the I-Advisor's task model was expanded to include the blueprint domain. Surprisingly the I-Advisor worked better in the blueprint domain. This was caused by the diversity of the events fired in each domain. The skill hierarchy and the blueprint domain have different object update and selection events. The skill hierarchy domain however turned out to have 16 different events, whereas the blueprint domain consisted of 29 possible events. This made it easier to learn a task model in the blueprint domain. In the skill hierarchy domain some events could be caused by different actions performed by the user. For instance if the user clicks on a checkbox called “To Be Trained” this will fire exactly the same event as when he clicks on a checkbox called “Recurrent”. These two actions are located in two different places in the task model, thus making it harder for the I-Advisor to learn what the user's current plan is.

We also had to decide upon a hierarchy for the task model. This hierarchy is used as a general framework to help the domain expert in annotating the logs of the user's actions. In our case it was quite easy to decide upon a hierarchy since we want to map the advice from the I-Advisor to the pages in the Help System. The Help System already has a certain hierarchy contained in its pages, consisting of certain goals which it advises the user to do in a certain order. By giving the task model the same hierarchy, the user's goal deduced by the I-Advisor is easily mapped to a page in the Help System. The hierarchy was created with the help of Harmen Abma, the domain expert who wrote the Advisor pages.

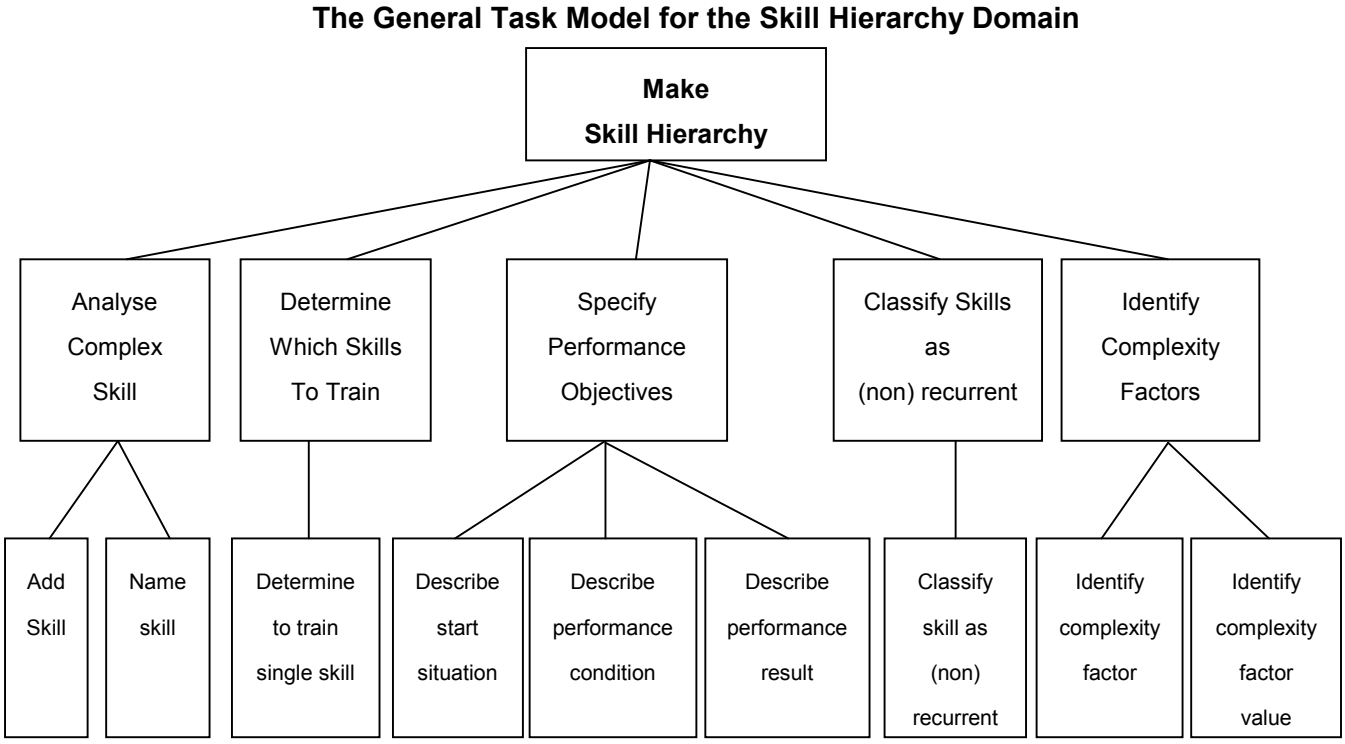

Figure 7.4

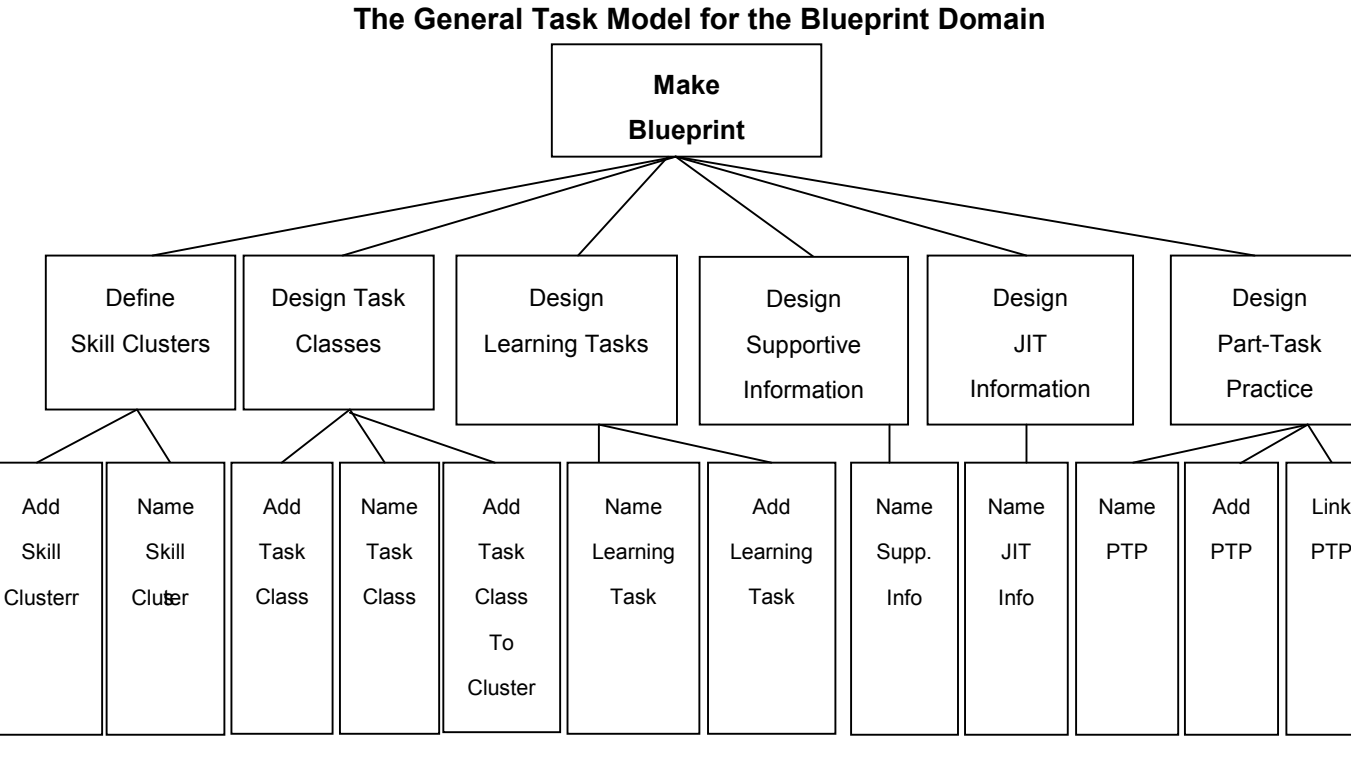

Figure 7.5

Figures 7.4 and 7.5 show the hierarchies we used for our task models. Figure 7.4 displays the hierarchy for the Skill Hierarchy domain. At the top of the task model is the main goal the user starts with in this domain, which is Make Skill Hierarchy. This goal consists of five different sub goals, and these sub goals can be divided into sub goals again. At the lowest level, the level of a goal such as Add Skill, the recipe consists solely of primitive actions and cannot be divided into sub goals any further. This hierarchy is just a general framework for the task model; no user will follow this task model exactly. It might be that a user will decide to jump back to Analyse Complex Skills after Specify Performance Objectives. This will result in an extra optional Analyse Complex Skills step after Specify Performance Objectives in the learned task model. Figure 7.5 displays the hierarchy for the other part of the AdaptIt application, the Blueprint domain.

The initial task model will be learned from a domain expert's interactions. This is due to the reason mentioned before that a new user will not have enough structure in his actions to infer a useful task model from them. A task model should preferably be learned from different domain experts. In many applications there are different ways, such as different menus or buttons, to accomplish the same task. A domain expert will often have adopted one way to accomplish certain goals and will always use that way. When a task model is learned from multiple domain experts, it is possible to capture many of the different ways of accomplishing the goals in the task model.

In my case there were only two domain experts available to learn the task model from. This is enough to learn a good task model since the AdaptIt tool does not have many different ways to do an action. Selecting a Skill for instance can be done in only two ways, in the ProjectBrowser and in the Skill Hierarchy Diagram. Also using different menus or buttons in AdaptIt often results in the same events, thus we do not have to take that into account when learning the task model.

Later on it should be possible to improve the task model with a specific user's actions when the user has discovered his own way of successfully using the application. A user might have discovered a completely new way to use the application, which the task model learned from the domain experts did not include. By also basing the task model on these actions, the I-Advisor will become better at following this specific user. The question remains whether a larger and more complex task model learned from many different users is more or less suitable for following new users with no structure in their actions. I will come back to that question at the end of this chapter.

Example Recipe for Name Skill

Recipe Goal="**NameSkill**":

Step **0**	Function="**SEL-ProjectBrowser-LFSkill**"	Primitive="**true**"	Optional="**true**"
Step **1**	Function="**OBJ-BasicAnalysisElementPanel-LFSkill**"	Primitive="**true**"	Optional="**false**"

Recipe Goal="**NameSkill**":

Step **0**	Function="**SEL-SkillHierarchyDiagram-LFSkill**"	Primitive="**true**"	Optional="**true**"
Step **1**	Function="**OBJ-BasicAnalysisElementPanel-LFSkill**"	Primitive="**true**"	Optional="**false**"

Figure 7.6

Figure 7.6 shows two example recipes for the non-primitive action Name Skill, which the I-Advisor has learned. The second step, the actual object update from when the skill's name is changed, is included in both recipes. The second and third column contain whether a step consists of a primitive action or not and whether the step is optional or not. The second step of both recipes is not optional. The first step, the selection of the skill whose name you want to change, is optional. It is possible to change the name of a skill already selected, so it is logical that this step is optional. The recipes also show that there are two different ways to select the skill. In the first recipe the skill is selected in the ProjectBrowser, in the second recipe this is done in the SkillHierarchyDiagram.

The finished task model learned by the I-Advisor was based on logs consisting of approximately 60 actions from two different domain experts. This task model is written away to a file, and will not be changed during the actual advising of the user. We don't want to slow down the user during his actions in the application, thus all the learning of the task model is done offline before the user starts his work in AdaptIt. All of the user's actions in AdaptIt are logged and stored to keep the possibility open to use them later on to refine the task model.

7.3 Advising the User

With the help of the task model the I-Advisor will now attempt to deduce the user's goals. Chapter 6 explained how the user is followed through the task model and the possible goals he could be working on are learned. The I-Advisor will not give the user advice about all the different possible goals he could be working on. Too much advice, and especially uncertain advice is misleading for a new user. This section explains the form of the advice given by the I-Advisor in more detail.

The I-Advisor's goal is to assist the user in learning to how to use AdaptIt in his own way by providing him with links to the help pages that correspond to his current tasks. It is not our intention to force the user to do anything, or to perform actions for the user. The I-Advisor is merely an advisor and not a wizard or tutor. In its panel the I-Advisor shows the user an overview of the general goals AdaptIt consists of. Whenever it has deduced the user's current goal, this goal is shown in a bold font. Figure 7.7 depicts the I-Advisor when it has learned that the user is currently working on Specify Performance Objectives. The user can ignore the advice if he feels that he knows enough about this goal to continue his work. If he gets stuck for some reason, he can click on the bold link. The help page corresponding to that goal will then be opened.

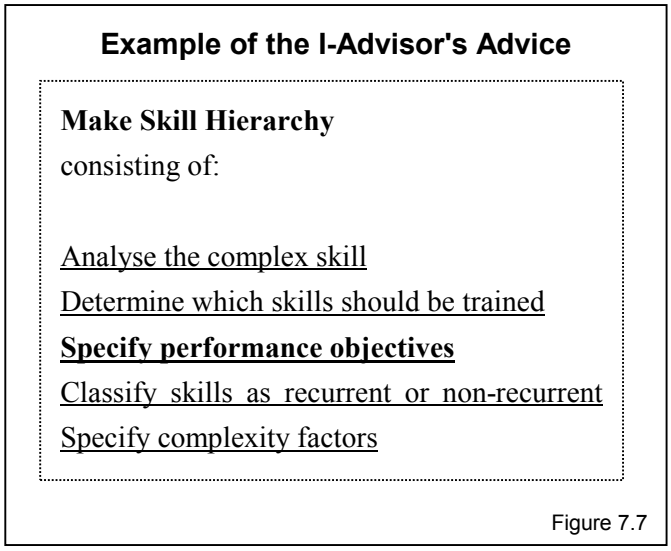

Figure 7.7

In some cases the I-Advisor will deduce many different possible goals. This is especially the case when a user starts to use the application, and there is no context to help learn the user's current goal. For instance selecting a skill is an action which is a part of almost every sub goal. A user who selects a skill as his first action can be trying to accomplish many different possible goals. In these cases the I-Advisor refrains from giving advice, since wrong advice can be very misleading for a new user.

If you compare figure 7.7 to figure 7.4 you will notice that the goals shown in the I-Advisor's panel are not the lowest level goals, but the goals one level higher up. There are three main reasons for this. The first reason is that showing these goals to the user gives him a good overview of the whole application. If he gets stuck in a detailed sub goal, he can look at this overview to regain the overall picture. The second reason is that moving one level higher up

makes it easier to follow the user and provide him with reliable information about his current goal. The events received from the AdaptIt tool are not detailed enough to follow the user at a lower level. There are too many identical events which can be fired by completely different actions performed by the user. The third practical reason for the higher level of advice is that the Advisor pages are not that detailed. There are no separate pages for the lower level goals, they are all described in one main page for the higher goal. It is thus not possible to provide the user with links corresponding to the lower level goals.

7.4 Lessons Learned While Building the I-Advisor

While building the I-Advisor certain advantages and disadvantages of the choices we made for its architecture appeared. This section will go over the effects of two choices, discussing their advantages as well as their disadvantages. These two choices are about which events the I-Advisor listens to and which structure is used for the task model.

7.4.1 Events Listened To

The choice of listening to the selection events and object updates was one which had one important advantage, it made it easy to implement the listeners. All that had to be done was adding two listeners; there was no need to build a lot of listeners for all the different kinds of events in the application. The events received from these listeners were all meaningful and a reasonable amount to learn a task model from.

However these existing listeners had another disadvantage apart from not reporting the view events (see section 7.1.1). The object updates are fired slightly later than they should for the I-Advisor. In AdaptIt when you type a new text in a text panel, there is no object update until you have finished typing the text and selected another object in the application. This makes the I-Advisor's advice somewhat late at times and advice given at the wrong time can confuse new users.

Another effect of the chosen listeners turned out to be an advantage. In every application there are many different buttons or menu options to accomplish the same task. Adding a SubSkill for instance can be done by pressing the "Add SubSkill" button, or right-clicking in the ProjectBrowser and selecting the corresponding menu option, or using the menu at the top, etcetera. If GUI events had been used, all these different GUI events would have to be mapped to one and the same event. Otherwise there would have been more than four different recipes for that simple sub goal. This would have lead to a huge task model, which would have taken a lot of time to search through for a user's action. It does not matter for learning the user's current

goal which type of GUI the user prefers to perform his actions. The important part is the effect of the GUI action, and this is the only thing reported by the listeners we use.

The ideal listener for the I-Advisor would have been one which included the GUI events. Actions such as selecting a new tabbed pane or opening a menu can give important clues as to what the user's current plan is. The listeners should however not fire different events for the different types of interfaces used. Adding a skill by clicking on a button should fire the same event as adding it by selecting an option in the menu. Also it would be best for the listeners to fire events immediately. They should not wait for the next event like the object update events do. Last of all, each action in the application should fire a unique event. You do not want to have events which can be caused by two different kinds of actions performed by the user.

7.4.2 Structure of the Task Model

The task model we used did not look at optional steps nor did it look at equality relations among the parameters. This choice was made based on a certain aspect of the structure of the task model learned by the I-Advisor.

The learned task model turned out to have two different types of non-primitive actions. The first type consisted of the non-primitive actions with recipes which had only primitive actions as steps, whereas the other type had recipes with only non-primitive actions as steps. There were no non-primitive actions which had both types of actions as steps. The first type of non-primitive actions, the one having only primitive actions as steps, all had recipes with a small number of steps and a lot of them had multiple recipes. The second type always had only one recipe, but this recipe was large with a lot of optional steps. If the task model was learned from more logs, these recipes became even larger. The other type of non-primitive actions with only primitive actions as steps however remained small in number of steps when more logs were used, and after a while no new recipes were added for these actions.

The fact that the recipes with only non-primitive actions as steps became very large with lots of optional steps indicated that almost any order of steps was possible to achieve this action. An example of this is when you build a skill hierarchy. A skill hierarchy can consist of any number of skills, thus the sub goal Add SubSkill can appear an arbitrary amount of times. Also you can Rename Skills and Delete Skills as many times as you want, and in any order. Clearly the current structure of our task model is not able to describe all ways to successfully build a skill hierarchy.

Possible ways to solve this using the current task model structure would be to introduce recursive rules, to add an unordered relation or to not make the task model too strict. Recursive

rules can capture things such as doing an action an arbitrary amount of times very well in one rule. This would however require changes to the search algorithms, in order to keep these from going into an endless loop. An unordered relation between the steps in a recipe signifies that these steps can appear in any order. Unordered relations combined with recursive rules could probably succeed at describing actions whose steps can appear an arbitrary number of times as well as in any order.

The last option is the one the I-Advisor uses, equality relations and the optional steps were dropped to make the task model less strict. By assuming that all steps are optional, which is the case when actions can occur in almost any order, you can still follow a user with the current type of task model. Otherwise the I-Advisor immediately lost a user when the user failed to perform an action which was not optional. It then had to start its search for the user's action from the beginning all over again. The equality relations turned out to not be useful in narrowing down the number of possible discourse states. The learned equality relations were exactly the same for each of the recipes, thus they were useless in discerning one goal from another.

There are still quite a few improvements possible on the current structure of the task model. The task model was able to follow the user, but at times it took a lot of searching through the recipes. Especially in the those recipes where the steps could appear in any arbitrary order the I-Advisor often lost the user. It then had to start searching for the user’s actions from the top of the task model again. The introduction of unordered relations and recursive rules could possibly solve this.

In the case of multiple possible focus stacks, the I-Advisor could do nothing more than give a list of the possible goals the user could be working on. This unfortunately happened quite often. It would have been better to have some kind of indication which of these possible goals is the most likely. You can then show the most likely goal as advice to the user, possibly accompanied by the second most likely goal. Chapter 9 explains Hidden Markov Models, a different possible architecture for the I-Advisor that can compute which possible goal is the most likely with the help of the Collagen-based task model.

8 LVNL Test Results

A prototype of the I-Advisor was tested by a training designer at Air Traffic Control Netherlands (LVNL). For a month she used an advanced prototype of the AdaptIt tool that contained the I-Advisor to help her design a training. This training was intended for the air traffic controllers to teach them how to use a new runway. The test user had no prior knowledge of the AdaptIt methodology. We had two meetings of two hours to introduce her to the AdaptIt methodology, just as all future users will get an introductory training. The AdaptIt tool was completely new to her, she had no prior experience with it.

For testing reasons she was given a prototype of the I-Advisor which only operated in one part of the AdaptIt application. The skill hierarchy domain had no I-Advisor support. In the blueprint domain the I-Advisor gave support in the way described in chapter 7. The plan behind this was that the test user could compare using the AdaptIt tool with the I-Advisor to AdaptIt without the I-Advisor's support. The I-Advisor also logged all her actions during the tests, as well as which advice the I-Advisor displayed to her. These logs were created primarily to determine how many times the I-Advisor was able to give advice. This could then be compared to the number of times the I-Advisor had to keep silent because it was not able to deduce the test user’s current goal.

8.1 Results

The test results consisted of an extensive debriefing with the test user. The debriefing was based on a questionnaire about all the different parts of AdaptIt, which is included in the appendix. The questions were about the methodology, the tool, the Advisor, and the I-Advisor. The next section is organised in the same way as the questionnaire was. The methodology and the tool were also discussed during the debriefing since the answers to these questions gave an indication of how well she understood the AdaptIt tool and the methodology behind it. We realize that the weakness of these tests was that there was only one designer involved.

8.1.1 Methodology

The methodology behind the AdaptIt tool was difficult to understand. The link between the skill hierarchy and the blueprint for instance was difficult to grasp. A workshop in advance was clearly necessary.

8.1.2 Tool

Although the AdaptIt tool is currently still a work in progress, there were few comments on the way the tool was constructed. Those comments she had were small bugs the tool designers had already detected and were working on. The system help from the Advisor was only needed a few times to discover how to accomplish certain goals in the tool.

8.1.3 Advisor

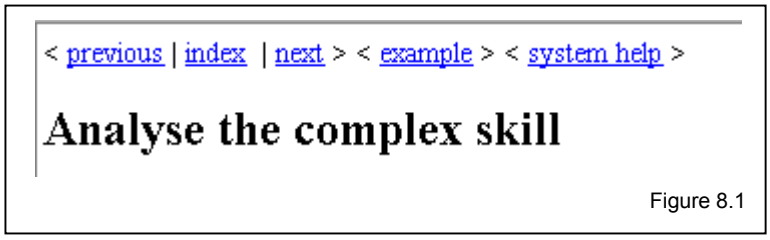

Figure 8.1

Before I explain the test results concerning the Advisor, I will first quickly describe all the different parts of the Advisor again. The Advisor consists of the parts shown in figure 2.3, a graphical overview of the methodology, the actual Advisor page, a table of contents and an overview of the phases (see section 2.2). An Advisor page contains text explaining the different goals in AdaptIt along with some simple examples in the text. The top of each Advisor consists of the links displayed in figure 8.1. These links direct the user to the Advisor pages about the previous and next goal in the tool, as well as a link to the index containing an overview of all the different goals. A link to a large example is also located at the top of most Advisor pages. If the user has trouble finding a certain button in the tool, he can click on the system help link.

During the debriefing we asked the test user about which parts of the Advisor she had used. The graphical overview of the methodology and the description of the phases she had mostly ignored. This is logical since she had indicated that she found it hard to understand the methodology. As mentioned in the previous section, she also seldom used the system help. The tool was built in such a way that it was not hard for her to find the different buttons and menus.

Most of the time she used the regular Advisor pages when she needed extra information about the tool. To navigate through these pages she used the **previous** and **next** links at the top of the pages. These two links, along with the **index** link gave her a good overview of the different goals which needed to be accomplished to design a good training.

She used two different ways to find the right pages in the Advisor. In the Skill Hierarchy domain, without the I-Advisor's support, she used the regular Advisor link to find the Advisor pages. This link leads to the main Advisor page, and from there she clicked on the index to find

the page she was looking for. Figure 8.2 shows the path she followed to find the right Advisor page. In the Blueprint domain she used the I-Advisor's links to find the right Advisor pages. As shown in figure 8.2, this takes her directly to the correct page.

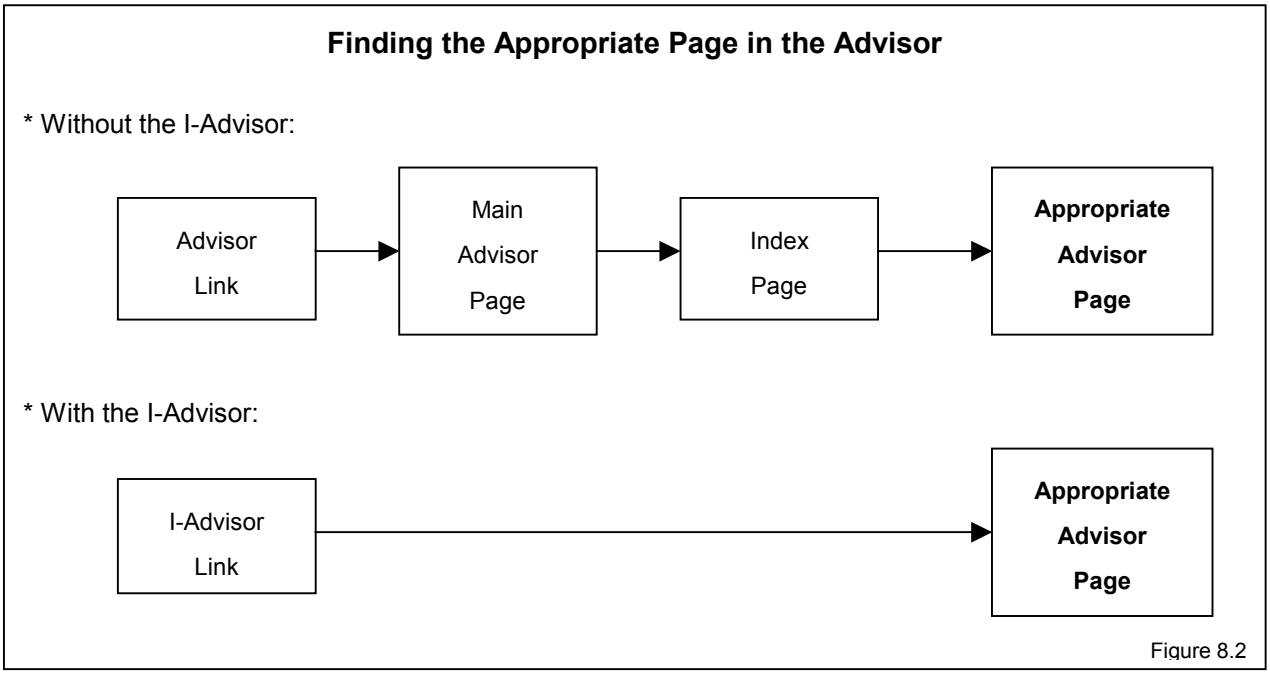

Figure 8.2

8.1.4 I-Advisor

The main object of interest during all these tests for this research project was the I-Advisor. There are three main questions we asked her about the I-Advisor's performance. First we wanted to know what she thought of the interface, i.e. the I-Advisor's panel with the overview of the goals. Second, we were curious when she used the links, and if the highlighted link ever led her to the wrong page. Last was the important question whether she felt the I-Advisor gave enough advice.

The interface in the I-Advisor's panel was easy to understand. The links and the highlighted link explained themselves. The constant updating of the links was not disturbing. The overview of the links was detailed enough. The Advisor pages provided her with more detailed information when she felt she needed it. However she felt the I-Advisor showed the wrong link. The I-Advisor shows the goal the user is *currently* working on. She would much rather have been shown the *next* goal she should work on according to the I-Advisor.

Whenever she added a new object to the blueprint, she would look at the I-Advisor to check which link was highlighted. This was the main way she used I-Advisor's links. The Advisor pages provided her with enough information to know what to do with the new object then. She

was especially focused on the highlighted link, the other links were seldom used. The highlighted link almost always led her to the right page. She seldom had the feeling that she was working on a different goal than the one that was highlighted.

The highlighted link might have always led her to the right page, however she felt the I-Advisor did not highlight links frequently enough. In too many cases the I-Advisor remained silent. This made her feel like she had done something wrong, and that the I-Advisor therefore could not deduce what she was doing. She would much rather have seen two or three possible current goals instead of the I-Advisor remaining silent.

8.2 Summary

The tests at LVNL had three main conclusions:

1) The level of support in the AdaptIt tool with the I-Advisor was sufficient to learn how to use the tool, but it was not sufficient to understand the methodology behind it.
2) Whereas the I-Advisor highlighted the user's current task, the test user wished to see the advised next task.
3) The I-Advisor should give advice more often, even if this advice involves multiple possible goals. Remaining silent makes the user think he is doing something wrong.

The next section will discuss these three conclusions and describe possible solutions.

8.2.1 Understanding the Methodology

The level of support in the AdaptIt tool with the I-Advisor was thus sufficient to learn how to use the tool, but not enough to understand the methodology behind it. The workshop normally given to new users appears crucial in learning to how to use the methodology. The Advisor was not able to explain this to her. The test user needed more general feedback information about the training she had designed, information she would have received from a domain expert during a workshop.

The question is whether a more active intelligent help system is able to replace a workshop given by a domain expert. An elaborate tutorial could come a long way. Building such a tutorial presents certain difficulties though. This tutor will have to learn everything about the training the user is busy designing. Some of this information can be obtained by listening to the application more closely, other information only by asking the user for it. Even then there are still certain things only a human domain expert can determine that a user is doing wrong. An example of this is when the user is designing a skill hierarchy. A domain expert can easily

determine if the user is giving the skills the wrong names, too general or too specific for instance. An intelligent tutor will have a very hard time learning such information from the names of the skills.

The test user indicated that she wouldn't mind having a more active Advisor, as long as it would have interrupted her only when she indicated that she needed more help. Such a more tutor-like I-Advisor will have to gather a lot more information from the AdaptIt application or from the user himself. Information one can think of are the number of skills in the Skill Hierarchy or maybe the time the user spent designing the training up to the moment of giving advice. This will result in a more elaborate user model. With this user model it should be able to give more context-sensitive advice to the user.

Unfortunately this will all come at a certain cost. The information I talked about cannot be obtained using one listener for one type of events. All these pieces of information will need separate listeners and will give different events. The I-Advisor would then need to do a lot more computations to maintain this more elaborate user model. This could possibly slow the application down, something I want to prevent from happening at all costs. The fact also remains that there will still always be certain properties of a training which an intelligent help system cannot learn.

8.2.2 Current Task vs. Advised Next Task

Although it is an interesting suggestion to show the advised next task instead of the user's current task, this is not easy to accomplish. The recipes can contain optional steps. In that case there are many advised next tasks. If the I-Advisor has also learned multiple possible current tasks, the list of advised next tasks becomes even longer. This list will definitely confuse the user. If the user wants advice as to what task he should perform next, he can look at the overview of the steps in the I-Advisor's panel. The links below the currently highlighted link can give information about the tasks the user can possibly accomplish next.

8.2.3 I-Advisor Remaining Silent

The I-Advisor did not want to present the user with misleading information and therefore remained silent when it had multiple possible solutions. The test user however started to feel insecure about her actions when the I-Advisor remained silent. A weakness of the I-Advisor's current architecture is that often the I-Advisor learns two or three possible goals and it has no indication which of these possible goals is the most likely goal. With the help of Hidden Markov Models probabilities can be computed which give an indication of which of these possible goals is the most likely. This is described in detail in the next chapter.

9 Hidden Markov Model

A *Markov Model* is a way of describing a process that goes through a series of *states*. At discrete times, the process undergoes a change of state, according to a set of probabilities associated with the possible transitions between the states. In Markov Models these transition probabilities from any given state to another depend only on the state and not on the previous history. This is called the *Markov Assumption*.

A Markov Model could for example be used in a system that attempts to predict what the weather will be like tomorrow based on a history of earlier observations of the weather (Fosler, 1998). The system's possible states are the types of weather, such as sunny, rainy and foggy. The system undergoes a change of state each day. A transition probability matrix describes all the possible transitions from one day to the next. The example matrix in table 9.1 shows that the probability of tomorrow being rainy given that today is sunny, is equal to 0.05.

Probabilities of Tomorrow's Weather based on Today's Weather

		Tomorrow's Weather		
		Sunny	Rainy	Foggy
Today's Weather	Sunny	0.8	0.05	0.15
	Rainy	0.2	0.6	0.2
	Foggy	0.2	0.3	0.5

Table 9.1 (from Fosler, 1998)

This model makes use of a first-order Markov assumption, i.e. the transition probability depends only on the next state and the current state. In the case of a second-order Markov assumption the transition probability would depend on the next state, the current state and the state preceding the current state.For the above exanple this would have resulted in a probability matrix with 3^3 probabilities. All the Markov Models in this thesis use the first-order Markov assumption.

A *Hidden Markov Model* (HMM) also describes a process that goes through a sequence of states. In a Hidden Markov Model, though, the true state of this process is *hidden* from the observer. The only indications we have of this true state are *observations*. These observations are linked to the possible states of the process with a certain probability. In a HMM we can thus only compute the probability that the process is in a certain state, whereas in a normal Markov Model one can know for certain what state the process is in.

Our weather example is easily changed to a Hidden Markov Model. We add a person who is locked in a room and cannot determine what kind of weather it is outside. He does however have an indication of the weather outside, namely the observation whether or not a visitor coming into the room is carrying an umbrella. This observation is linked to the possible states of the system with a certain probability, as is shown in table 9.2. If the visitor is carrying an umbrella, it will be more likely that the state of today's weather is rainy. The probability that the visitor is carrying an umbrella given that it is raining outside is thus higher, i.e. 0.8, than the probabilities for the other possible states. The person inside the room can never know whether or not it is raining outside for sure, the visitor could always still be carrying his umbrella from when it was raining the day before.

Probabilities of Seeing an Umbrella Based on the Weather

	Probability of Umbrella
Sunny	0.1
Rainy	0.7
Foggy	0.2

Table 9.2 (from Fosler, 1998)

Hidden Markov Models are often used to characterize real-world signals in terms of signal models. These real-world signals can for instance be speech samples or sensor measurements from a mobile robot. In the case of speech samples, HMM's are used for speech recognition (Rabiner 1989, Kupiec 1992). The states of the HMM are then the different phonemes the speech samples could possibly consist of. If the real-world signal consists of sensor measurements, the HMM can be used for robot localisation (Fox et. al. 1999). The different states of the HMM are in that case the robot's possible locations. Lately HMM's are also used for user modeling (Orwant 1995, Lane 1999).

In the I-Advisor the real-world process we want to model is the goal the user has inside his head while using the AdaptIt tool. This goal is the true state of the process, and unfortunately this state is hidden from us. The only indications we have of this state are the actions the user performs in the AdaptIt tool. With the help of these actions and a HMM, we can compute the probabilities for each of the possible goals the user can have.

The HMM in the I-Advisor is initialised using the Collagen-based task model described in Chapter 5. There were two main reasons to use the task model instead of trying to learn the HMM directly from the user's actions. First, all the information necessary for the initialisation of the HMM, such as the number of states and the number of observations, is easily obtained

from the task model. Second, in most articles the HMM's used in user interfaces were also initialised manually based on prior knowledge (Orwant 1995, Seymore, McCallum, and Rosenfeld 1999). This led me to believe that it would not be an easy task to learn the HMM directly from the user's actions. Since the project's time was limited, the decision was made to use the Collagen-based task model.

In the following section the initialisation of the HMM using the task model is described in more detail. Section 9.2 explains how the probabilities for the possible states of the model are computed each time the user performs an action. Section 9.3 concludes with the test results of an I-Advisor prototype which makes use of a HMM.

9.1 Initialising the HMM

To initialise the Hidden Markov Model the following elements need to be defined:

- **N** , the number of possible states in the model.

The states in the I-Advisor's HMM are the goals the user has in his mind while performing his actions. The state of the model at time t is defined as q_t. The observations have to give an indication of the hidden state of the model, thus we have to be able to link the observations to the states with a certain probability. This means we have to define a probability for a certain *primitive action* (the observation) occurring given a certain *non-primitive action* or *goal* (the state). The easiest way to do this is by using those non-primitive actions as the states of the model whose recipes consist only of primitive actions. Figure 9.1 once again shows the hierarchy of the task model for the skill hierarchy domain. All the actions highlighted in bold have recipes which only have primitive actions as their steps. These actions are the possible states of the model. They are also the lowest possible level of non-primitive actions. By comparison, these actions were the most specific goals the user could achieve in AdaptIt and they were usually found on top of the focus stack as the current focus.

Using the task model from figure 9.1 N is defined as the total number of non-primitive actions highlighted, namely nine. The states $S = \{S_1 , S_2 , \ldots , S_9 \}$ are in this case:

S = { AddSkill, NameSkill, DetermineToTrainSingleSkill, DescribeStartSituation, DescribePerformanceCondition, DescribePerformanceResult, ClassifySkillAs(Non)Recurrent, IdentifyComplexityFactor, IdentifyComplexityFactorValue }

The General Task Model for the Skill Hierarchy Domain

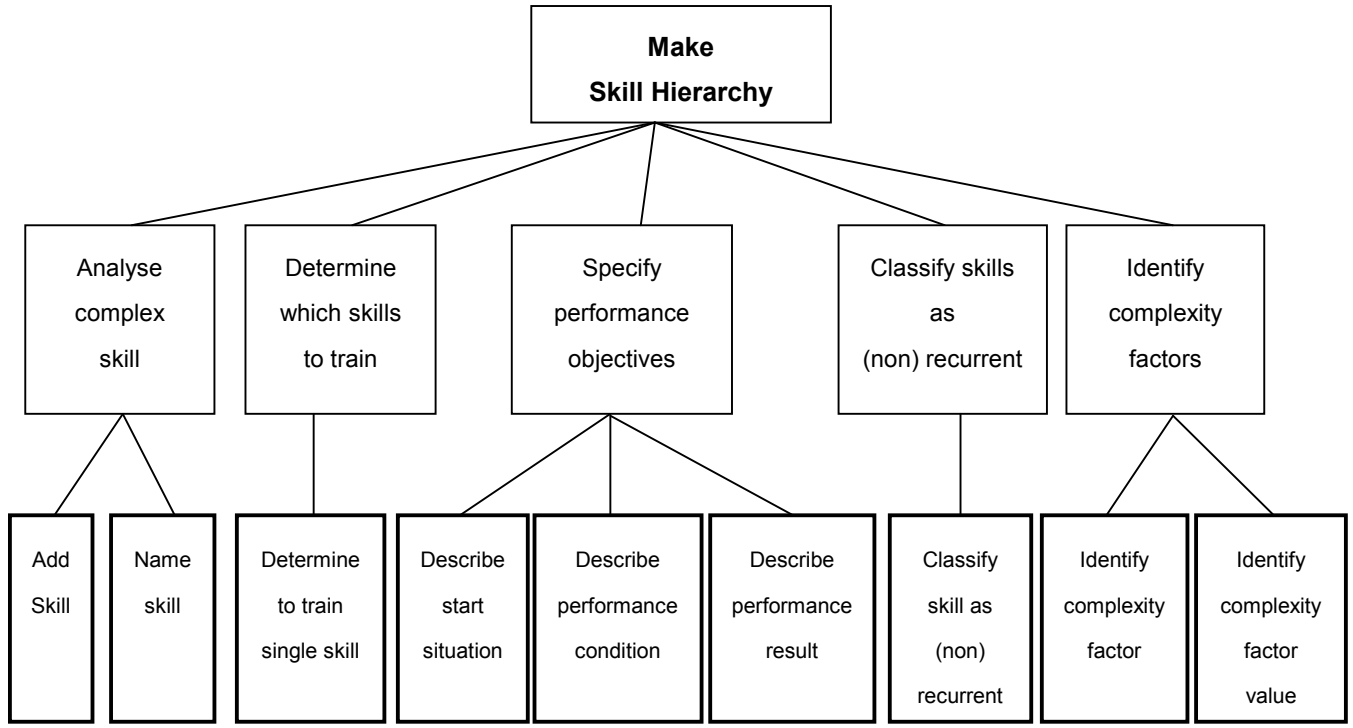

Figure 9.1

- **M** , the number of distinct observation symbols per state.

The observation symbols correspond to the physical output of the model. In our case the output of the model consists of the user's primitive actions in the AdaptIt tool. The number of observation symbols is the number of possible primitive actions the user can perform in the AdaptIt tool. In the skill hierarchy domain a user can do sixteen different primitive actions. M is thus defined as sixteen. The individual symbols $Y = \{y_1, y_2, \ldots, y_{16}\}$ are in this case:

Y = { SEL-ProjectBrowser-LFTrainingProject, SEL-ProjectBrowser-LFSkill,
OBJ-BasicAnalysisElementPanel-LFSkill, … , OBJ-ConditionPanel-LFSkill }

- The state transition probability distribution $A = \{a_{ij}\}$ where:

$$a_{ij} = P [q_{t+1} = S_j \mid q_t = S_i], \qquad 1 \le i, j \le N.$$

This distribution defines how likely it is to go from one state to another. For example if the current state q_t = AddSkill, then it is far more likely that the next state q_{t+1} = NameSkill occurs than q_{t+1} = IdentifyComplexityFactor. This can be deduced from the structure of the task model, as shown in figure 9.1. The state IdentifyComplexityFactor is located at the end of the task model, whereas NameSkill is in the beginning just like AddSkill.

Our HMM is fully-connected, i.e. any state can be reached from any other state in a single step. Some transitions are however highly unlikely, such as the one described above:

$$P [q_{t+1} = \text{IdentifyComplexityFactor} \mid q_t = \text{AddSkill}]$$

The model is fully-connected because the user can in fact go from any state to any other state in the AdaptIt tool. If the user is distracted, he will make very unlikely transitions. These transitions, no matter how unlikely, have to remain possible. Otherwise you cannot successfully model the user's behaviour. The initialisation of the state transition probability distribution with the help of the task model is discussed in section 9.1.2.

- The observation symbol probability distribution in state j, $B = \{ b_j (k) \}$, where:

$$b_j (k) = P [v_k \text{ at } t \mid q_t = S_j], \qquad 1 \le j \le N, \quad 1 \le k \le M.$$

This distribution defines how likely it is that a certain observation v_k occurs at time t given that the current state q_t of the model is S_j. In other words this signifies how likely it is that a user performs a certain primitive action in the AdaptIt tool when he has a certain goal in mind. We can learn these probabilities from the recipes in our task model. Section 9.1.1 describes how this is done in more detail.

- The initial state distribution $\pi = \{ \pi_i \}$ where:

$$\pi_i = P [q_1 = S_j], \qquad 1 \le i \le N$$

The initial state in the I-Advisor is the main goal the user has in mind when he starts to design a training. In the skill hierarchy domain this goal is MakeSkillHierarchy. Thus the probability is:

$$P [q_1 = \text{MakeSkillHierarchy}] = 1$$

For all the other states the initial probability is equal to 0. The action MakeSkillHierarchy is not in the lowest level of non-primitive actions contrary to the other actions we chose to use as possible states for the model. It has no primitive actions as steps in its recipe. Therefore there are no observations linked to this state. Although we do add MakeSkillHierarchy as one of the possible states in the model, it is only an initial state. Since all the observation probabilities for this state are 0, this state is quickly left once the user starts doing actions and the model will not return to it.

9.1.1 Observation Probabilities

The observation probabilities correspond to how likely it is that a user performs a specific primitive action in the AdaptIt tool given that he has a certain goal in mind. The user will only perform a primitive action while working on a certain goal if he thinks that an action is a step in achieving that goal. The information about which actions can contribute to which goals can be retrieved from the recipes in the task model.

If we select the state **NameSkill** for example, we will search through the task model for all the recipes describing this specific state. In this case there are two recipes, together containing the following three primitive actions as their steps:

{ **SEL-ProjectBrowser-LFSkill, SEL-SkillHierarchyDiagram-LFSkill, OBJ-BasicAnalysisElementPanel-LFSkill** }

The observation probabilities for the state **NameSkill** are now computed by dividing one by the total number of primitive actions that can contribute to this specific state. The sum of all the observation probabilities for a certain state has to sum up to one, therefore we divide one by this number. In our example the total number of primitive actions that can contribute to the state **NameSkill** is three. Thus the probabilities are:

$$P\,[\textbf{SEL-ProjectBrowser-LFSkill}\ \text{at}\ t \mid q_t = \textbf{NameSkill}] = 1 / 3$$
$$P\,[\textbf{SEL-SkillHierarchyDiagram-LFSkill}\ \text{at}\ t \mid q_t = \textbf{NameSkill}] = 1 / 3$$
$$P\,[\textbf{OBJ-BasicAnalysisElementPanel-LFSkill}\ \text{at}\ t \mid q_t = \textbf{NameSkill}] = 1 / 3$$

For all the other observations the probabilities are set to zero. In the same way all the observation probabilities for the different states are initialised.

9.1.2 Transition Probabilities

The transition probabilities indicate how likely it is that the user will go from one state to another. Similar to the observation probabilities, the transition probabilities are initialised with the help of the task model. The hierarchical tree-like structure of the task model contains a lot of information about which transitions from one state to another are the most likely. The location of a state in the hierarchical structure of the task model is characterised by four different properties. These properties are:

1) Is it the first / last state of a higher goal?
2) Which higher goal does it belong to?
3) Which other states were found following this state in the same higher goal?
4) Which higher goal followed this state's higher goal?

For example the state **Describe Start Situation** from figure 9.1 is described as being the first state (Property 1) of the higher goal **Specify Performance Objectives** (Property 2). The state **Describe Performance Condition** is found following this state in the same higher goal (Property 3). The higher goal following this state's higher goal, i.e. **Specify Performance Objectives**, is called **Classify Skills As (non) Recurrent** (Property 4).

These properties used by the I-Advisor only remember two levels of a hierarchy. They store the details of the level of the states themselves and the level above the states, i.e. the higher goals. The task models learned by the I-Advisor, both for the skill hierarchy as well as the blueprint, contain three levels. The third level does not contain any extra information though, since in this level all the goals belong to one main goal, namely Make Skill Hierarchy or Make BluePrint. Thus storing the details of two levels is enough to capture the whole hierarchical structure of the I-Advisor's task models.

With the help of these properties we can define the transition probabilities for each possible pair of states. The probability that the model will go from a certain given state to another state is defined as:

$$P [q_{t+1} = \text{NextState} \mid q_t = \text{CurrentState}]$$

The I-Advisor will select a certain state as the CurrentState and assign *ranks* to all possible transitions from this state to all possible NextStates. These ranks correspond to how likely this transition is and thus how high the probability for this transition should be when compared to the other probabilities. The highest rank which is given to the most likely transitions is equal to four; the lowest possible rank for the least likely transitions is equal to one. There are four different ranks in the I-Advisor because there are four types of transitions in the task model

which differ so much that they need to receive different ranks. It is good to remember that the model is fully-connected and all transitions are possible. It is even possible to stay in the same state, thus a rank is also given to a reflexive transition. If there are 10 possible states, this means you have to give ranks to 10 possible transitions for each state.

There are two types of states which have to be ranked in a distinctly different way: the *last states* and the *first / intermediate states*. The last states are all the states which are found as the last state of a higher goal in the task model. A state can appear in more than one place in the task model. If a state is found once as the last state of a higher goal, it will be ranked as a last state. The first / intermediate states are all the states which never appear as the last state of a higher goal in the task model. An example of a difference between the rankings for these two types is that the last state of a higher goal needs to have higher ranks for all the transitions to the first states of the other higher goals. It is very likely that a user will go from a last state of a higher goal to the first state of another higher goal. It is however less likely that a user will go from a last state of a higher goal to the intermediate or last state of another higher goal, or in other words jump right into the middle of another higher goal. There are few differences between the types of rankings for first and intermediate states. Therefore they use the same ranking system.

Figures 9.2 through 9.5 show the two ranking systems used to initialise the HMM. Figures 9.2 and 9.4 display which different ranking categories there are and in which cases they are given. The ranks are assigned starting at the highest ranks and moving down to the lower ranks. A transition can fit more than one ranking category, for instance if a certain Next State is a first and last element of a following higher goal. This can happen if a following higher goal consists of only one state. If we assume for instance that the Current State is a last state, we have to use the ranking system from figure 9.2. In this case the states of the first category, i.e. the first elements of a following higher goal, receive a rank of 4 and the states of the second category, i.e. the last elements of a following higher goal, receive a rank of 2. Since the ranks are given from high to low, this transition will receive a rank of 4.

The figures 9.3 and 9.5 both show an example of the ranks given to all the possible Next States in case of an example CurrentState. In figure 9.3 the CurrentState is a last state, i.e. **Describe Performance Result**. The example CurrentState chosen in figure 9.5 is a first / intermediate state named **Describe Start Situation**. Below this CurrentState the rank for the following, reflexive transition is shown:

$$P [q_{t+1} = \text{Describe Start Situation} \mid q_t = \text{Describe Start Situation}]$$

The Four Possible Different Ranks for Last States:

(**Rank 4**) The next state is one the following:

- **A)** A possible following state in the same higher goal.
- **B)** The first element of the following higher goal.
- **C)** The same as the given state.

(**Rank 3**) The next state is the first element of another or the same higher goal.

(**Rank 2**) The next state is one of the following:

- **A)** A intermediate / last element of a following higher goal.
- **B)** A state with the same higher goal.

(**Rank 1**) The next state is none of the above.

Figure 9.2

Example Ranks System for the Last State
"Describe Performance Result"

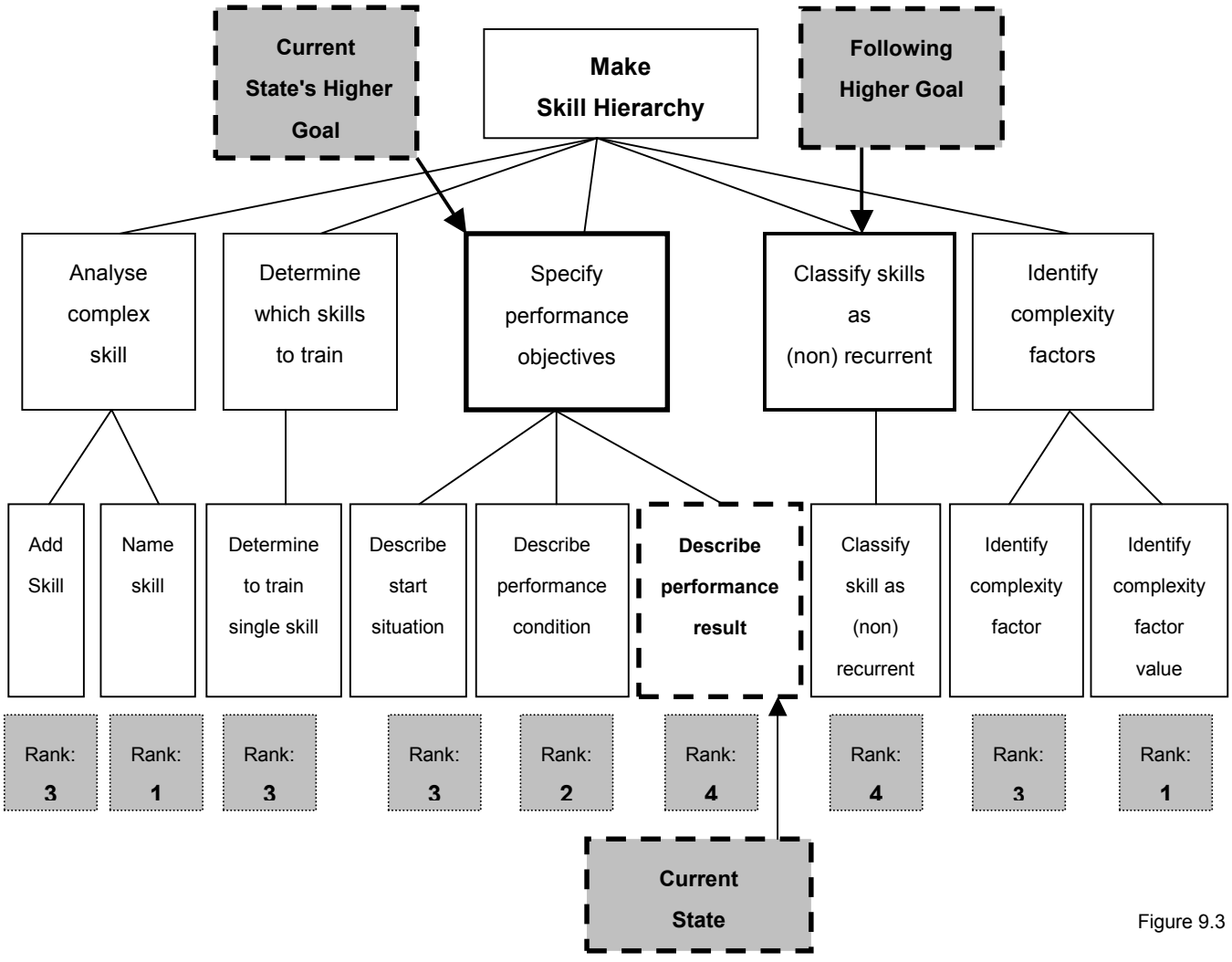

Figure 9.3

The Four Possible Different Ranks for First / Intermediate States:

(**Rank 4**) The next state is one of the following:

A) A possible following state.

B) The same as the given state.

(**Rank 3**) The next state is one of the following:

A) A state with the same higher goal.

B) The first state of a following higher goal.

(**Rank 2**) The next state is one of the following:

A) An intermediate / last state of a following higher goal.

B) The first element of another higher goal.

(**Rank 1**) The next state is none of the above.

Figure 9.4

Example Ranks System for the First / Intermediate State "Describe Start Situation"

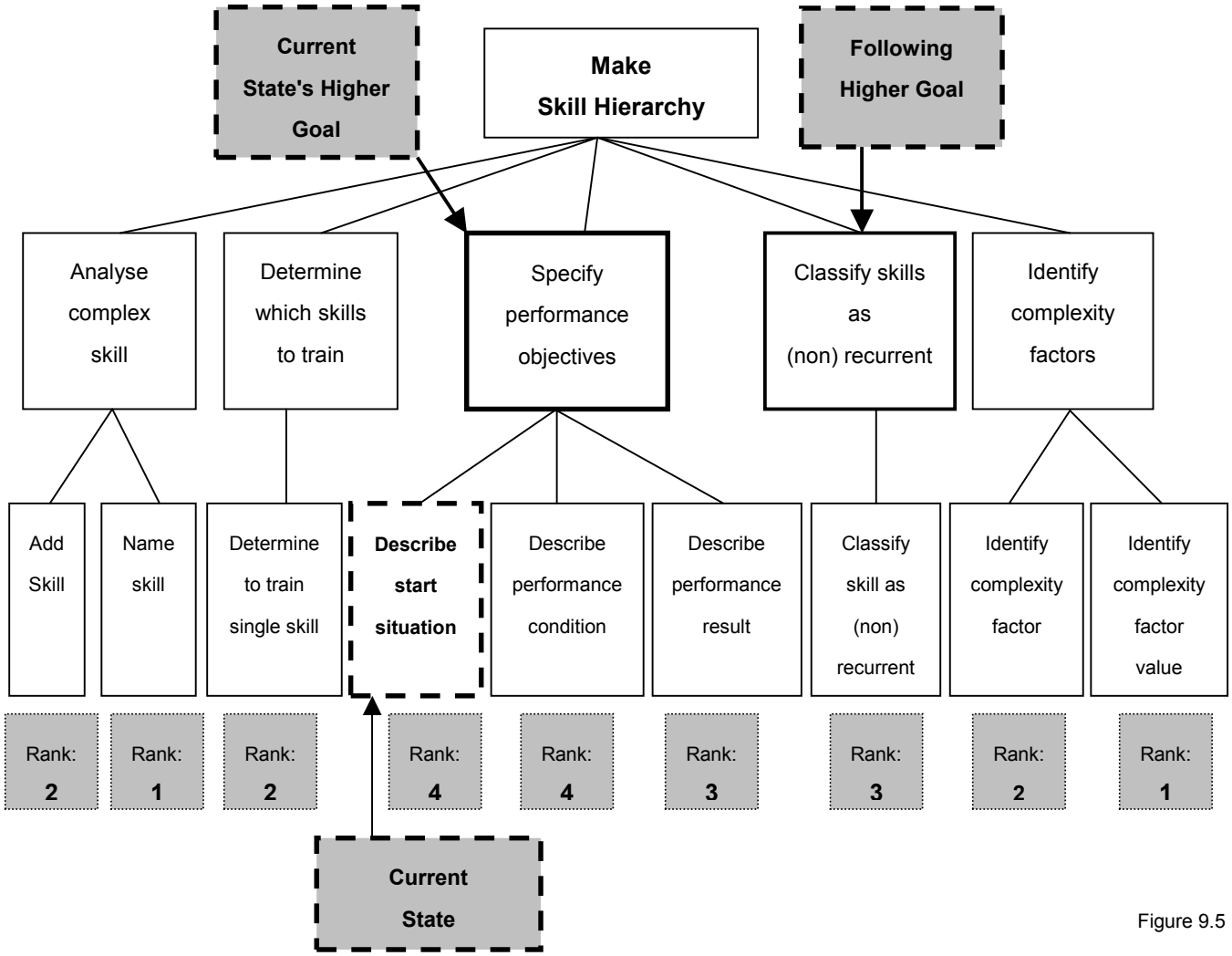

Figure 9.5

The rank for this transition is four. This transition has the highest possible rank, since it is very likely that user will stay in the same state for a while. He will tend to keep a goal in mind until he has performed all the primitive actions in the AdaptIt tool necessary to accomplish that goal.

Once all the ranks have been given to all the possible NextStates for a certain Current state, they are stored along with the sum of all the ranks learned for this Current state. This sum is used to normalize the probabilities so that together they sum up to one. We define this normalisation factor as being b:

$$b = \frac{1}{\sum_{\text{NextState}} \text{rank}(\text{NextState}, \text{CurrentState})}$$

If we sum up all the example ranks given in figure 9.5, we get a sum of 22. We compute the separate transition probabilities in the following way:

$$P[q_{t+1} = \text{NextState} \mid q_t = \text{CurrentState}] = b \cdot \text{rank}(\text{NextState}, \text{CurrentState})$$

For example the reflexive transition from the CurrentState DescribeStartSituation to the NextState DescribeStartSituation has a rank of four and will thus receive the following probability:

$$P[q_{t+1} = \text{DescribeStartSituation} \mid q_t = \text{DescribeStartSituation}] = \frac{4}{22}$$

In the same way all the probabilities are computed for all the possible transitions between the states based on the structure of the task model.

9.2 Computing the Probabilities of Each State

After each action performed by the user the I-Advisor uses the Hidden Markov Model to compute which states are the most likely. The I-Advisor will compute the probability for a state q at time t based on a sequence of observations $y_0, \ldots y_t$, namely:

$$P(q_t \mid y_0, \ldots, y_t) \qquad \text{(equation 9.1)}$$

We do not want to wait for future actions, we want to compute the probability immediately at time t. This problem is called the filtering problem. This name stems from the frequency domain. The observations $y_0, \ldots, y_t$ are viewed in this domain as being noisy information about

the underlying signal q_t. By computing the probability P (q_t | y_0 , … , y_t) you can "filter" the noise from the observations.

In the I-Advisor we move through the observations, i.e. the user's actions, one by one. Thus equation 9.1 is reduced to P (q_t | y_t). Using Bayes' rule we can reverse the terms q_t and y_t:

$$P(q_t \mid y_t) = \frac{P(y_t \mid q_t) \cdot P(q_t)}{P(y_t)} \qquad \text{(equation 9.2)}$$

We are going to define a new variable c which is the normalisation factor:

$$c = \frac{1}{P(y_t)} \qquad \text{(equation 9.3)}$$

To ensure that the probabilities for all the possible different states for the I-Advisor sum to one we use the following normalisation constant:

$$P(y_t) = \sum_{q_t} P(y_t \mid q_t) \cdot P(q_t) \qquad \text{(equation 9.4)}$$

Using the total probability theorem for P (q_t) and the fact that the probabilities for the possible states are conditioned upon the observations, we can rewrite equation 9.2 as:

$$P(q_t \mid y_t) = c \cdot P(y_t \mid q_t) \cdot \sum_{q_{t-1}} P(q_t \mid q_{t-1}) \cdot P(q_{t-1} \mid y_{t-1}) \qquad \text{(equation 9.5)}$$

This recursive formula can be used to compute the filtering posterior probabilities P (q_t | y_t) for all states q_t and all time steps t of the HMM. The probability P (y_t | q_t) is the observation probability which we computed in section 9.1.1. The transition probabilities, i.e. P (q_t | q_{t-1}), were learned from the structure of the task model in section 9.1.2. The quantity c is the normalisation factor which makes certain that all the probabilities over all the possible states sum to 1. Finally the probabilities P (q_{t-1} | y_{t-1}) are the posterior probabilities for the different states from the previous time step, namely step t-1.

9.3 HMM and the I-Advisor

The I-Advisor uses two separate Hidden Markov Models for the Skill Hierarchy and the Blueprint domain due to the fact that the I-Advisor has two separate task models for these domains. The Blueprint HMM is initialised using the Blueprint task model, and has the goal

MakeBluePrint as its initial state. In the same way the Skill Hierarchy task model serves to initialise the Skill Hierarchy HMM with the initial state Make Skill Hierarchy. Since the collection of possible actions in these two domains is disjoint, the I-Advisor can easily determine in which of the two domains of the AdaptIt tool the user is working. Whenever the user switches to a different domain, the I-Advisor switches to the corresponding HMM.
In the next paragraphs a few example situations will be shown to illustrate how the Hidden Markov Model is used in the I-Advisor. These examples show the probabilities for all the states which are not equal to 0 and the entropy. The entropy is used to measure the peakness of the probability distribution. The entropy of a distribution P(S) is computed using the following formula:

$$E = -\sum_{i} P(S_i) \cdot \log P(S_i)$$

This formula is used with the convention that 0 log 0 = 0. If the entropy is high, the probabilities are distributed evenly among the possible states of the model. In this situation it is difficult to determine which is the most likely state. The HMM can thus not give more information about the user's possible goal than the Collagen-based approach. If the entropy is low, the probability of one state is clearly higher than that of the others. The I-Advisor can then show this most likely goal to the user.

Example Situation 1

---- The current probabilities are: ----
* MakeSkillHierarchy = 1.0
The entropy is: 0.0

---- The current probabilities are: ----
* NameTrainingProject = 1.0
The entropy is: 0.0

Figure 9.6

The example situation from figure 9.6 shows the initial state of the model and one following state. The user is working in the Skill Hierarchy domain. The HMM is thus initialised to be in the initial state MakeSkillHierarchy. The entropy is equal to 0 since the probability is equal to 1 for one single state. The user's next action changed the state to NameTrainingProject. The I-Advisor is certain that the user is in this state, the probability is once again 1 for a single state. Clearly the action the user performed could only contribute to this one goal.

Example Situation 2

---- The current probabilities are: ----

* NameSkill = 0.281
* AddSubSkill = 0.327
* DetermineSingleSkillToBeTrained = 0.210
* DescribeStartSituation = 0.089
* MakeSkillRecurrent = 0.093

The entropy is: 1.486

---- The current probabilities are: ----

* NameSkill = 0.152
* AddSubSkill = 0.118
* DetermineSingleSkillToBeTrained = 0.269
* DescribeStartSituation = 0.228
* MakeSkillRecurrent = 0.233

The entropy is: 1.569

Figure 9.7

Figure 9.7 shows two example situations where it is less clear which goal the user has in mind. In the first situation three possible states are almost equally likely, leading to a very high entropy. However, the I-Advisor gives his advice about the goals one level higher up. The two possible states with the highest probabilities, NameSkill and AddSubSkill, belong to the same higher goal, i.e. Analyse Complex Skill. The I-Advisor can thus safely conclude that the most likely state is Analyse Complex Skill.

The second situation from figure 9.7 is a clear case where the HMM can give no extra information about the user's most likely goal. The entropy is high and the probabilities are spread equally among the possible states. If we look at the higher goals, they are still spread equally. The possible states are almost all from different higher goals.

As can be concluded from the examples, the HMM cannot always give extra information about the user's possible goals. In some cases different goals are simply equally likely, and the I-Advisor can do nothing more than highlight the links to all these goals. If there are states that are clearly more likely, the I-Advisor can highlight the most likely state with another colour

than the other states. The I-Advisor can even use a different colour for the most likely state after that, and so on.

This added information could be very helpful for a new user. Our test user's main complaint was that the I-Advisor remained silent in too many situations. This can be solved by highlighting multiple goals, and colour–highlighting the most likely goals. If a user is lost and has no idea what goal he should accomplish at that moment, he can turn to the I-Advisor for help. The I-Advisor shows the links to the goals experienced users would have been trying to accomplish at that point. The lost user can read about these goals in the Advisor pages. With this extra information he will hopefully regain understanding of his actions in the AdaptIt application.

The Hidden Markov Model currently used by the I-Advisor is initialised using the Collagen task model. Although this task model was easy to use to initialise the HMM, certain changes to the model would make it even more suitable to initialise HMM's. Currently the Collagen task model only remembers if a step in a recipe is optional or not optional. For the HMM it would be interesting to know how optional a step is. If a step appears in only one of the five segments for this recipe, it should get a lower observation probability for this recipe than if it had appeared in four of the five segments. Especially if a lot of logs of experienced users are used to learn the task model, this can give a lot of extra information for the observation probabilities. In the same way the transition probabilities can use this added information. Non-primitive actions can also be optional, and transitions to optional states are less likely than transitions to states that are not optional. One of the added advantages of remembering how optional steps are, is that the influence of "purpose-less" steps is filtered out. Even experienced users will sometimes make a misclick. These actions will not occur often and will thus become very optional. The task model will remember that these actions are very optional and thus their influence in the HMM is significantly reduced.

10 Conclusion

This thesis described two possible architectures for an intelligent help system for the AdaptIt application. The first architecture was based on Collagen and the second made use of Hidden Markov Models. Both of these architectures were implemented in a prototype. The Collagen-based prototype was tested by a possible future user of the AdaptIt tool, i.e. a training designer at LVNL. The prototype of the I-Advisor using Hidden Markov Models was not extensively tested. This chapter will evaluate both of these I-Advisor architectures by quickly passing over all the requirements we set up in chapter 4 and describing whether or not they have been satisfied. This is followed by some final remarks and several ideas for future research.

10.1 Requirements

The main requirement for the I-Advisor was that it should be able to provide a user of the AdaptIt tool with context-sensitive support. Users of the AdaptIt tool had trouble learning how to use the tool. They felt they did not have a good overview of the whole process of designing a training. The I-Advisor provided the user with links to the pages in the help system based on the user's current task context. These Advisor pages then contained more detailed information about the user's current task, as well as worked-out examples and system help.

Tests with a prototype using the Collagen-based architecture were successful. They were even so successful that a prototype of the I-Advisor has been added to the AdaptIt application. The I-Advisor made it easy for users to access the information in the help system and it gave the user a good idea of where he was. The test user did however complain that although the I-Advisor nearly always gave the correct advice, it did not give advice often enough. In many cases the I-Advisor remained silent when it had deduced multiple possible tasks the user could be working on. Therefore a prototype was built using Hidden Markov Models. This prototype was able to distinguish which task was the most likely in the case that the I-Advisor had deduced multiple possible goals the user could be working on.

10.1.1 Functional Requirements

The I-Advisor showed his context-sensitive support in the form of an overview of the different tasks involved in designing a whole training. All the different tasks in this overview were displayed as links to the corresponding pages in the help system. The link to the user's current task was highlighted. This way the overview gave the user an idea of the context of his current task, and the opportunity to easily access more information about these surrounding tasks.

The I-Advisor deduced the user's current task with the help of a user model and a previously learned task model. The user model explained the user's actions in AdaptIt and was updated every time the user performed a new action. The task model was learned offline from the actions of experienced AdaptIt users. The primary difference between the two architectures discussed in this thesis is the type of user model they contained. Both architectures did however use the same Collagen-based task model. The following two paragraphs will describe the two different user models.

The first architecture discussed in this thesis used the discourse state from Collagen to model the user's actions. This discourse state consisted of a focus stack and a plan tree. The focus stack was used to store all the goals the user was trying to accomplish at that moment. The most general goal was on the bottom of the stack; the most specific one was located on the top of the stack. The plan tree was used to store what goals the user's previous actions contributed to.

The second architecture contained Hidden Markov Models. Each time the user performed a new action this I-Advisor computed probabilities for all the different possible goals the user could be working on. The goal with the highest probability could then be shown to the user.

The I-Advisor contained two task models which together described the whole AdaptIt application. Both these task models were learned from two logs of approximately 60 actions performed by a domain expert in the AdaptIt application. These logs were annotated by the same domain expert. This annotating consisted of the domain expert adding to the actions the tasks he was trying to accomplish with these actions. The logs of actions were XML files which could easily be viewed and edited by the domain expert.

10.1.2 Technical Requirements

The prototype of the I-Advisor was implemented in Java, just like the rest of the AdaptIt tool. The files used by the I-Advisor, which stored such things as the task models and the logs of the user's actions, were written in XML. The I-Advisor's prototypes all operated successfully within the whole AdaptIt application.

The requirement that no changes shall be made to the AdaptIt application to make interaction between AdaptIt and the I-Advisor possible had a lot of consequences. The I-Advisor could easily operate without any changes to the AdaptIt application, but there were quite a few changes possible in AdaptIt which would probably have improved the I-Advisor's performance considerably. Both the prototypes now made use of the existing listeners in AdaptIt to discover which actions the user had performed. These existing listeners made it very easy to connect the

I-Advisor prototypes to the AdaptIt application. They however had two properties which at times made it difficult for the I-Advisor to follow the user successfully.

The first property was that the view events were ignored by these two listeners. View events consist of the changes to the GUI elements which provide the user with a view of the data objects in the application. Examples of view events are selecting a tabbed pane and opening a new menu. Both events contain important information about the user's intentions, but were unfortunately not reported by the listeners in AdaptIt.

The second property consisted of the fact that two different actions performed in AdaptIt by the user could lead to the same event being received by the I-Advisor. The I-Advisor had no way of determining which of the two actions the user had actually performed, thus it would constantly deduce multiple possible current tasks. In some situations the I-Advisor could use the context of the user's action to discover which of the two actions the user had performed. In other situations the contexts of both possible actions were almost the same and it was thus not possible to discern one from the other. This property of the listeners became apparent during tests in the two different domains in AdaptIt. While operating in the Blueprint domain, the I-Advisor was able to provide the user with a lot more advice than in the Skill Hierarchy domain. The Skill Hierarchy domain turned out to have a lot more situations where different actions led to the same event than the Blueprint domain.

The I-Advisor interacted with the AdaptIt application in the same way as the Q-Advisor. The I-Advisor displayed its advice in the panel which was previously used by the Q-Advisor. It also displayed the same kind of links to the Advisor pages as the Q-Advisor did.

10.2 Final Remarks

In a lot of applications it is often difficult to find the help pages you are looking for. You have to know the right name for the task you want help information on, as well as have an idea of how the help pages are organised. The information you are looking for is nearly always located in the help system, only the hard part is discovering where it is located exactly.

Intelligent help systems such as the I-Advisor can make finding the right help pages a lot easier for users. They merely look over the shoulder of the user, and attempt to deduce what task the user is working on. If they are successful, they can provide the user with an easy, instant access to the corresponding help pages. They never disturb the user while he is performing his actions, the user will not even notice he is being watched.

Both the Collagen and the Hidden Markov Model architectures were appropriate for such an advisory, passive intelligent help system. The Collagen task model combined with the HMM user model was the most successful at providing the users with advice. In most situations it could compute a most likely goal the user was working on. This approach could probably become even more successful with the help of more logs from test users and with certain changes to the Collagen-based task model to make it more suitable for initialising the HMM.

There was one aspect of the AdaptIt application which at times made it difficult to discover the user's current task. This was the fact that the user had a large amount of freedom in the AdaptIt tool. The different tasks in AdaptIt could often be achieved in almost any order. In most cases it was possible to switch tasks around, or perform a certain task an arbitrary amount of times. This amount of freedom the user had made it difficult for the I-Advisor to capture all the different ways AdaptIt users could behave in its task model. In general intelligent help systems will be more successful in providing users with advice in applications that have a more clearly-defined order in the separate tasks they consist of.

In the Collagen-based task model the recipes which attempted to explain this freedom became large with lots of often optional steps. This had different effects on the two architectures discussed in this thesis. The Collagen-based architecture searches through the recipes in the task model to find the user's last actions. Using more logs of test users' actions to learn the task model made this architecture slower and led to it coming up with multiple possible current tasks more often. The Hidden Markov Model only used the task model to initialise all its probabilities. A larger task model thus did not slow it down, it only led to the initial probabilities becoming more clearly-defined. Basing the task model on more test logs thus slowed down the first Collagen-based architecture, whereas it made the Hidden Markov Models more effective in giving advice.

10.3 Future Work

This research project could easily have been twice or three times as long. During my research many interesting topics for future work appeared which I could not research further due to the limited time of my project. This section will quickly pass over some of these research topics.

The most interesting topic for further research is whether or not it is possible to create an intelligent help system which can automatically learn its task model with the help of statistical learning methods. Is it possible for example to let the I-Advisor generate its Hidden Markov Models based solely on the user's actions? Large amounts of test data will certainly be

necessary to test this. Even then it will still not be easy to learn task models automatically without any added domain knowledge.

It would also be interesting to test the current I-Advisor with more test data. This test data can consist of logs from more different test users, or of more different kinds of actions from the AdaptIt application. One could think of including all the view events in the learning of the task model, or all the separate mouse clicks. This however would require making quite a few changes to the AdaptIt application.

The current structure of the task model also has room for further research. It would for instance be interesting to determine if any improvements to the current task model structure are possible which would make it more suitable to explain the degree of freedom users have in the AdaptIt application. One could think of adding recursive rules or unordered relations to the recipes. Both of these suggestions would require changes to the search algorithms currently used by the I-Advisor. Another interesting direction for research is whether the task model structure can be changed to make it more suitable for initialising the Hidden Markov Models. A possible change to the task model could be for instance storing in its recipes how often a certain action appeared in the logs of previous users.

Finally, it would very interesting to see the I-Advisor's architecture tested in other applications. Certain properties of the AdaptIt application, such as the structure of the listeners and degree of freedom for the user, had a great deal of effect on the performance of the I-Advisor. Different kinds of applications might require other types of help systems. For the AdaptIt application we decided to build a passive advisor. In other applications more active intelligent help systems might very well be more successful in helping users.

Appendix A Glossary

action	there are two kinds of actions, primitive and non-primitive. *Primitive* actions can be executed directly within an application. *Non-primitive* actions are achieved indirectly by achieving other actions.
adaptable interface	an interface that lets the user choose how the system should adapt.
AI	stands for Artificial Intelligence. Artificial Intelligence is a branch of science where they try to model the way a human thinks in order to create a computer system that can do intelligent actions.
alignment phase	the first of the two phases of task model learning. In the alignment phase the actions are mapped to a task model, which only contains recipes with required steps and non-primitive actions without parameters.
annotate (verb)	to indicate in a list of primitive actions which subsets of the actions contribute to the same subtask, along with possible other annotations such as marking an action as optional or which actions could have occurred in another order.
annotated examples	a log of primitive actions that has been annotated by a domain expert.
collaboration	a process in which two or more participants coordinate their actions in order to achieve shared goals.
collaborative interface agent	a software agent that collaborates with the human user of a (often complex) computer interface.
current (discourse) purpose	the shared goal two (or more) participants in a collaboration are working on. Represented by the goal on top of the focus stack, also known as the focus of attention.
discourse	the communication between two or more participants involved in a collaboration.
discourse interpretation	the process of considering how the latest communication or observed action can be viewed as contributing to the current discourse purpose.

discourse state	a mental model of the status of the collaborative tasks and the conversation about them. The discourse state consists of a focus stack and a plan tree.
event	an utterance or primitive application action performed by a user or an agent.
focus of attention	goal on top of the focus stack, also known as current (discourse) purpose.
focus stack	a stack of goals, which is part of the discourse state. The goal on top of the focus stack is the current purpose or focus of attention of the discourse.
HCI	stands for Human-Computer Interaction. Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.
induction phase	the second of the two phases of task model learning. In the induction phase the task model from the alignment phase is generalized to be consistent with all the input examples. This is done by inducing the optional steps of the recipes, the ordering constraints between the steps and the propagators.
information filtering	aims to find a structure in the massive amounts of information available today that can be used to aid users in finding the information that is useful for them.
intelligent help system	Program that assists a user in completing a certain task. A help system lets the user plot his own course through an application. From the user's interactions with the application the help system gathers enough information to aid the user.
intelligent tutor	Program that teaches a user how complete a certain task, this is done by inferring the user's understanding of an application from his performance on specific subtasks.
IUI	stands for Intelligent User Interface. IUI's are human-machine interfaces that aim to improve the efficiency, effectiveness and naturalness of human-machine interaction by representing, reasoning and acting on models of the user, domain, task, discourse or media.

Markov Assumption	this assumption holds if the transition probabilities from any given state depend only on the state and not on the previous history.
Markov Model	a way of describing a process that goes through a series of states. At discrete times, the model undergoes a change of state, according to a set of probabilities associated with the possible transitions between the states. The Markov assumption holds for these transition probabities.
Hidden Markov Model	a way of describing a process that goes through a sequence of states, just like a Markov Model. Only the true state of the process is *hidden* from the observer and the only indications we have of this true state are observations. These observations are linked to the possible states of the model with a certain probability.
multimodal interaction	interaction between the user and the interface along different modalities, i.e. ways of communicating such as natural language, video, etcetera.
non-primitive action	actions that are achieved indirectly by achieving other actions, also known as abstract or composite actions or intermediate goals.
observation probability	probability that a Hidden Markov Model gives a certain observation as output given that the model is in a certain state.
plan recognition	the process of inferring intentions from actions.
plan tree	an encoding of a partial SharedPlan between a user and an agent participating in a collaboration.
primitive action	actions that can be executed directly within a program.
rank	numbers assigned to all the possible transitions in the Hidden Markov Model that correspond to how likely this transition is and thus how high the probability for this transition should be when compared to the other transition probabilities.
recipe	describes a set of steps that can be performed to achieve a non-primitive action. Recipes can contains constraints on the temporal ordering of the steps, as well as other logical relations among the parameters of the steps.
segment (noun)	a sequence of actions that serve the same purpose, also known as non-primitive actions. Open segments are segments on the focus stack, closed are ones popped off the focus stack.

segment (verb)	the process of indicating in a log of actions which subsets of actions contribute to the same purpose, same as *annotating*.
self-adaptive interface	an interface that adapts to the users autonomously.
SharedPlan	the formal representation of the mutual beliefs about the goals and actions to be performed and the capabilities, commitments and intentions of the participants involved in a collaboration, based on work by Grosz and Sidner.
steps	elements a recipe is made up of. Steps are (non-) primitive actions.
task model	a library of recipes that specifies the typical steps and constraints for achieving non-primitive actions.
transition probability	probability which defines how likely it is to go from one state to another in a (Hidden) Markov Model.
type	each action has a certain type associated with a set of parameters
user adaptation	techniques that allow the user-computer interaction to be adapted to different users and different usage situations.
user modeling	techniques that allow a system to maintain knowledge about a user.

Appendix B References

de Croock, M. ; van der Pal, J. ; Abma, H. ; van Merrienboer, J. ; Paas, F. and Eseryel, D. *D3.2 Design of the ADAPT Method: ADAPT Methodology* Prepared for the European Commission, DGXIII under Contract No. IST-1999-11740 as a deliverable from WP 3, activity 3.2, Date of issue: 23 April 2002

Fosler-Lussier, E. *Markov Models and Hidden Markov Models: A Brief Tutorial* ICSI Technical Report TR-98-041, December 1998.

Fox, D. ; Burgard, W. and Thrun, S. *Markov Localization for Mobile Robots in Dynamic Environments* Journal of Artificial Intelligence Research Vol. 11 (1999), pages 391-427

Garland, Andrew and Lesh, Neal *Learning Hierarchical Task Models By Demonstration* (Submitted to 17th Int. Joint Conf. on Artificial Intelligence, Seattle, WA, August 2001)

Grosz, B.J., and Sidner, C.L., *Attention, Intentions, and the Structure of Discourse, Computational Linguistics*, 12:3 (1986)

Hearst, M., Allen, J. F., Guinn, C. I., Horvitz, E. *Mixed-Initiative Interaction* IEEE Intelligent Systems, Trends & Controversies feature, 14 (5), September-October 1999.

Horvitz, E. , J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. *The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software*. Users Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, July 1998.

Kupiec, J. 1992 *Robust Part-of-Speech Tagging Using a Hidden Markov Model.* Computer Speech and Language, 6:225-242.

Lane, T. *Hidden Markov Models for Human/Computer Interface Modeling* Proceedings of the IJCAI-99 Workshop on Learning about Users, pp 35-44. 1999.

Lesh, N. , Rich, C. and Sidner, C. L. *Using Plan Recognition in Human-Computer Collaboration* (7th Int. Conf. on User Modeling, Banff, Canada, July 1999, pp. 23-32)

Lieberman, Henry. *Introduction to Intelligent Interfaces*. 1997 WWW: http://lieber.www.media.mit.edu/people/lieber/Teaching/Int-Int/Int-Int-Intro.html)

Lochbaum, Karen E. *A Collaborative Planning Model of Intentional Structure*. Computational Linguistics 24 (4): 525-572 (1998)

Maybury, M. T. and Wahlster, W. *Intelligent User Interfaces: An Introduction*. Readings in Intelligent User Interfaces. 1998. pp 1-14. Morgan Kaufmann Press. ISBN: 1-55860-444-8.

Orwant, J *Heterogeneous Learning in the Doppelgänger User Modeling System.* User Moedling and User-Adapted Interaction, 4 (2): 107-130, 1995.

Rabiner, L.R. *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, in Proceedings of the IEEE, vol. 77, no. 2, February 1989, pp. 257-286.

Rich, C. and Sidner, C. L. *COLLAGEN: A Collaboration Manager for Software Interface Agents* (User Modeling and User-Adapted Interaction, Vol. 8, No. 3/4, 1998, pp. 315-350)

Rich, C., Sidner, C.L. and Lesh, N.B., *Collagen: Applying Collaborative Discourse Theory to Human-Computer Interaction*, Artificial Intelligence Magazine, 2001

Russel, Stuart and Norvig, Peter *Artificial Intelligence: a Modern Approach* Prentice Hall; ISBN: 0131038052; 1st edition (January 15, 1995)

Seymore, Kristie, McCallum, Andrew, and Rosenfeld, Ronald *Learning Hidden Markov Model Structure for Information Extraction* AAAI 99 Workshop on Machine Learning for Information Extraction, 1999.

Annika Wærn, *What is an Intelligent Interface?* Notes from an introduction seminar, March 1997. WWW: http://www.sics.se/%7Eannika/papers/intint.html

Annika Wærn, *What is an Intelligent Interface?* Slides. 1999. WWW: http://www.sics.se/~jarmo/kurser/iuiuu99/slides/introduktion/index.htm

Annika Wærn. *Recognising Human Plans: Issues for Plan Recognition in Human - Computer Interaction*. Ph. D. Thesis. 1996.

Appendix C Questionnaire

This questionnaire was used for the debriefing of the test user at Air Traffic Control Netherlands (LVNL).

A. The Methodology

1. What did you think of the methodology?
   - Was it (too) complex or easy to understand?
   - Was the way it was built logical?
   - Did it work well in practice?
   - How hard / easy was it to learn to work with?
   - What did you think of the order of the steps in the methodology?

B. The AdaptIt Tool

2. What did you think of the tool itself?
   - Was it hard / easy to use?
   - Did the tool correspond to the methodology?
   - Were there things you would have liked to see differently in the tool?

C. The Advisor

3. What did you think of the help (the Advisor) in AdaptIt?
   - What did you think of the normal Advisor pages?
   - And the worked-out examples?
   - And the system help?
   - And the graphical overview of the methodology at the top of the page?

D. The I-Advisor

4. What did you think of the contents of the I-Advisor panel?
   - Was this clear / easy to use?
   - Did the overview of the steps help you or would you rather have had more detailed information?

5. What did you think of the links in the I-Advisor?
   - Did you find the answers to your questions in the pages the I-Advisor linked to?
   - Which links did you use the most? (the highlighted link or the other links?)
   - In what situations did you use the links? (to prepare yourself for a next step or only when you did not know what to do anymore)
   - Were there times when clicking on the highlighted link gave you information which did not help you?
   - Did the I-Advisor display any strange behaviour, which you did not expect?
   - Would you mind if the I-Advisor bothered you with questions to improve its advice?

E. Miscellaneous Comments

6. For example suggestions or remarks about:
   - The tool
   - The Advisor
   - The I-Advisor

Appendix D Traceability of the Requirements

Functional Requirements

(1) *The support given by the I-Advisor shall consist of links to the pages in the help system corresponding to the user's current task context.*

The support given by the I-Advisor consists of a list of links to the Advisor pages, with the user's current task context highlighted in bold (described in section 7.3).

(2) *This support given by the I-Advisor shall be based on:*

- *the user's actions within AdaptIt.*
- *task model*

The I-Advisor's support is based on a user model learned from the user's actions within AdaptIt, as well as a task model. The user model is based on the discourse state from Collagen (described in chapter 6). The task model is also based on Collagen (described in chapter 5).

(3) *The I-Advisor shall contain a task model, which describes a part of the AdaptIt application.*

The I-Advisor contains two task models, one for the skill hierarchy domain and one for the blueprint domain of the AdaptIt tool. Together they describe the whole AdaptIt application (described in section 7.2).

(4) *A domain expert shall be able to view and annotate the logs of previous users, in order to aid the learning of the task model.*

The logs of previous users are written to a XML file. This file can be viewed and edited by a domain expert to add the annotations. The file with the annotations is then read in by the I-Advisor and used to learn the task model (described in the Appendix, the Implementation section).

Technical Requirements

(5) *The I-Advisor shall be implemented in Java just like the AdaptIt tool.*

The I-Advisor is implemented in Java.

(6) *The I-Advisor shall operate within a small part of the AdaptIt application.*

The I-Advisor operates successfully in the whole AdaptIt application (described in section 7.2). The tests at LVNL were done with a prototype, which only operated in the blueprint domain of the AdaptIt tool (described in chapter 8).

(7) *No changes within the original AdaptIt application shall be necessary to link the I-Advisor to the AdaptIt application.*

No changes were made to the AdaptIt application, the I-Advisor listens to the existing ObjectUpdateManager and SelectionController to obtain the user's actions (described in section 7.1). The I-Advisor shows its advice in an already existing panel, which was previously used by the Q-Advisor (described in section 2.3 and 7.3).

(8) *The I-Advisor shall interact with the AdaptIt application in the same way as the Q-Advisor and the Advisor.*

The I-Advisor replaces the Q-Advisor in the AdaptIt application and interacts with the application in exactly the same way. It uses the same panel in the bottom left corner to display its advice (described in section 2.3), and its advice consists of the same kinds of links to the Advisor pages as the Q-Advisor (described in section 7.3).

Appendix E Implementation

The I-Advisor's main objective is to provide a user of the AdaptIt tool with advice based on his current task context. The I-Advisor can also be used in two other ways, namely to create logs of a user's actions and to learn a new task model based on these logs. The I-Advisor's behaviour is set with the help of four booleans, which can be found in the **IAdaptIt** and **IAdvisor** classes. The next three sections describe each of these three types of behaviour generally. In the last section the organisation of my code, as well as all the different classes in the I-Advisor are explained in more detail.

- **Creating Logs**

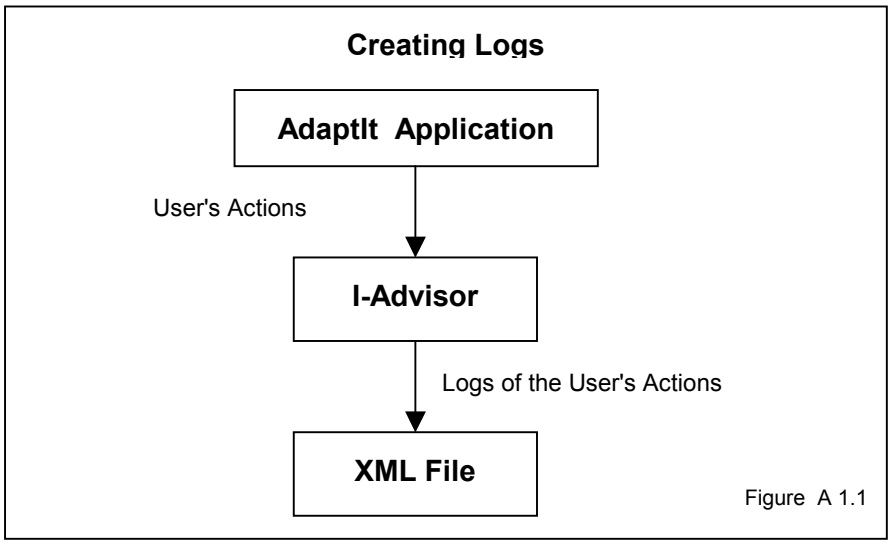

Figure A 1.1

When the I-Advisor is set to write logs, it will store all of the user's actions and write them to a file (shown in figure A 1.1). Each time the user performs an action in the AdaptIt tool, that action is stored in a **PrimitiveAction** object. All the user's primitive actions are stored together in a Vector. When the AdaptIt tool is closed, all of the user's actions are written to a XML file in the form of a DOM Tree.

Four properties of a user's action are stored in the XML log file:

(1) *Index* = an unique index which can range from 0 up to the total number of actions - 1
(2) *Function* = the type of action the user did
(3) *Parameter* = the object upon which the user did an action
(4) *Time* = time of the action

More information on user's actions and the way they are stored is found in section 7.1.

- **Learning Task Models**

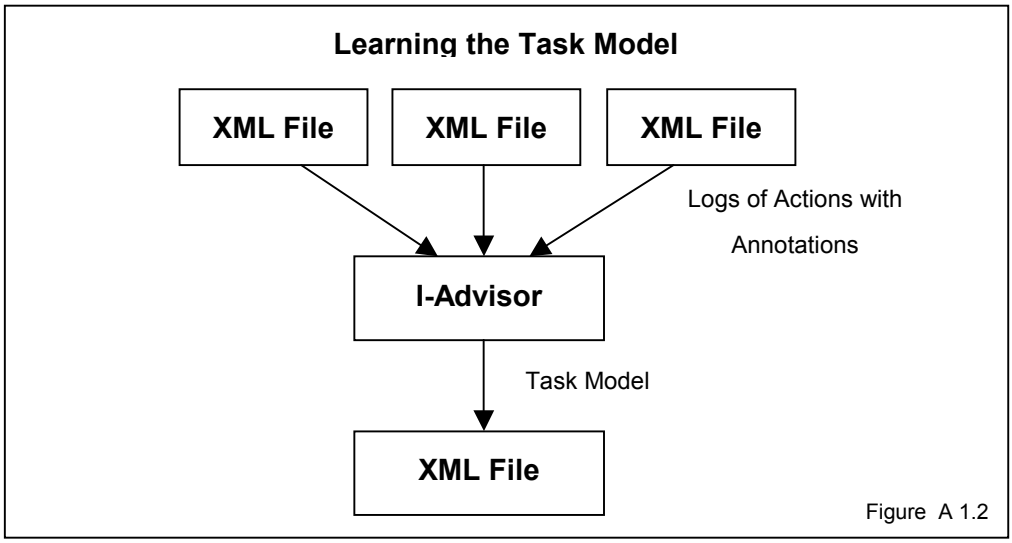

Figure A 1.2

The general behaviour of learning a task model is displayed in figure A 1.2. The I-Advisor first reads in all the XML files containing the logs of previous users, along with a domain expert's annotations. These annotations are crucial for learning a new task model, without them it is not possible to learn a new task model! The new task model is learned from these logs and the annotations and subsequently written to a XML file. The class that does this is called **PreTaskModel**. More information on the learning of the task model is found in the description of this class, below the annotating of the logs by the domain expert is explained

The annotating of the logs of the user's actions has to be done manually by a domain expert or an experienced AdaptIt user. Annotating is the process of indicating which segments of actions contribute to the same purpose. A more detailed description of why this is necessary is found in sections 5.2 and 7.2. Practically annotating consists of adding Segments to the SegmentsList in the log file. An example Segment could be:

This Segment indicates that all the actions from Start index 2 up to End index 4 (please note that the action with index 4 is *not* part of the segment) contribute to the same purpose, namely "**NameSkill**". This means the following two actions stored in the list of the user's actions, i.e. the ActionsList, contribute to this purpose:

<Action Index="**2**" Function="**SEL-SkillHierarchyDiagram-LFSkill**" Parameter="**Skill**" />
<Action Index="**3**" Function="**OBJ-BasicAnalysisElementPanel-LFSkill**" Parameter="**Rijden**" />

All of the actions in the ActionsList have to be given a purpose. The number of layers in the hierarchy can differ, some actions will not have a very specific purpose. They will only contribute to the main goal, which in AdaptIt is either Make Skill Hierarchy or Make Blue Print, depending on which part of AdaptIt the user is in. Figure A1.3 shows part of the example annotations used for the learning of the task model, which is currently used by the I-Advisor.

More information on the annotating of the logs and the learning of the task model can be found in chapter 5.

```
<SegmentsList>
    <Segment Index="0" Name="MakeSkillHierarchy" Start="0" End="65" />
    <Segment Index="1" Name="AnalyseComplexSkills" Start="1" End="9" />
    <Segment Index="2" Name="NameSkill" Start="1" End="3" />
    <Segment Index="3" Name="NameSkill" Start="3" End="4" />
    <Segment Index="4" Name="AddSubSkill" Start="4" End="7" />
    <Segment Index="5" Name="NameSkill" Start="7" End="9" />
    <Segment Index="6" Name="DetermineSkillsToBeTrained" Start="9" End="14" />
    <Segment Index="7" Name="DetermineSingleSkillToBeTrained" Start="9" End="11" />
    <Segment Index="8" Name="DetermineSingleSkillToBeTrained" Start="11" End="13" />
    <Segment Index="9" Name="DetermineSingleSkillToBeTrained" Start="13" End="14" />
    <Segment Index="10" Name="SpecifyPerformanceObjective" Start="14" End="21" />
```

Figure A 1.3

- **Providing the User with Advice**

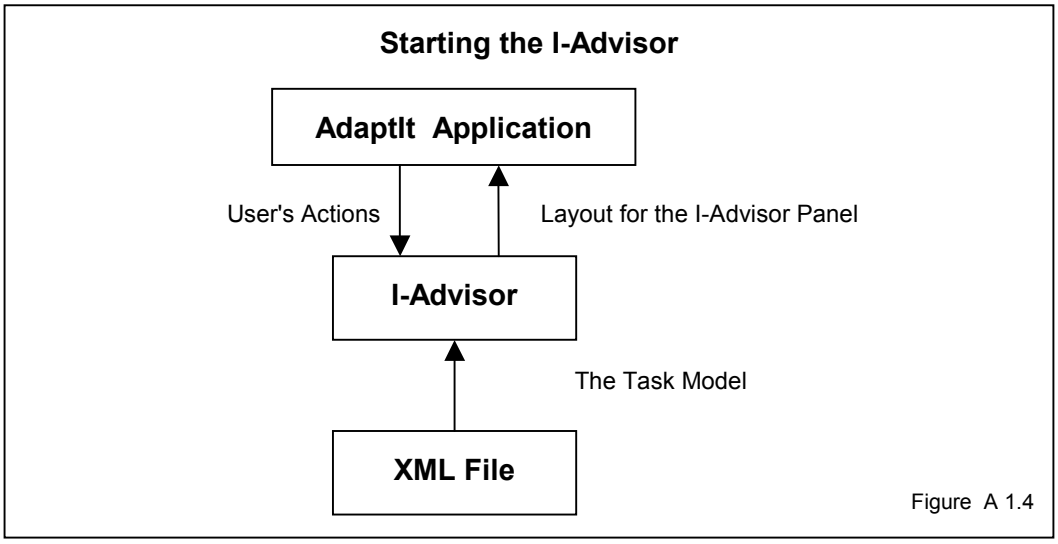

Figure A 1.4

The I-Advisor main behaviour consists of looking at the user's actions and providing him with advice based on these actions. Figure A 1.4 displays this behaviour. The I-Advisor receives one of the user's actions from the AdaptIt application. It will update its user model, i.e. a **DiscourseState** object, based on this action and the task model, which is stored in a **TaskModel** object. If the I-Advisor successfully learns the user's current goal, he will update the I-Advisor's panel in the AdaptIt tool. This is done by sending a new HTML layout to the AdaptIt tool.

Description of the Code

My code is located in three different places: in nlr.iadvisor, com.adaptit.components.advisor and in the folder for the property files.

- **The following classes are contained in nlr.iadvisor:**

AnnotatedExample.java	PreTaskModel.java
DiscourseState.java	PrimitiveAction.java
DOMSerializer.java	Recipe.java
DOMTreeReader.java	Segment.java
DOMTreeWriter.java	SegmentElement.java
IAdaptIt.java	SegmentSet.java
IAdvisor.java	Step.java
Plan.java	TaskModel.java
PreRecipe.java	TaskModelLearner.java

- **The following classes are found in com.adaptit.components.advisor:**

IAdvice.java
Layout.java
HelpMapping.java

- **The following XML files are found in the property files folder:**

(obtained by ApplicationManager.getApplicationManager().getPropertiesDirectory())

bluePrint.xml
skillHierarchy.xml
layoutblue.xml
layoutskill.xml
AdaptitEditorPreferences.xml

Description of the Classes

This section will give a brief description of what all the different classes do.

Classes in nlr.iadvisor

- **AnnotatedExample.java** (design described in section 5.2)

An object of this class stores an annotated example, which consists of:

(a) A vector with the primitive actions (objects of the **PrimitiveAction** class) performed by a user in AdaptIt.
(b) A vector with the segments (objects of the **Segment** class) added by the domain expert to a user's primitive actions.

- **DiscourseState.java** (design described in chapter 6)

The DiscourseState class initialises a new discourse state and updates it every time the user performs an action in AdaptIt. It performs the following steps:

(1) *Reading in the task model*
Both the task model XML files contained within the PropertiesDirectory are read in using an object of the **DOMTreeReader** class. The I-Advisor uses separate task models for the skill hierarchy (*skillHierarchy.xml*) and for the blueprint (*bluePrint.xml*) domain. The task models contained in these files are stored in an object of the **TaskModel** class.

(2) *Initialising the DiscourseState*
This is achieved by creating a new focus stack and a new plan tree. The new focus stack has only the main goal pushed on it, which is either MakeSkillHierarchy for the skill hierarchy domain, or MakeBluePrint for the blueprint domain. The plan tree, an object of the **Plan** class, consists of a Hashtable.

(3) *Updating the DiscourseState*
The discourse state is updated every time the user performs an action in AdaptIt. This consists of the I-Advisor expanding the focus stack and plan tree so that they explain the user's action, with the help of the task model.

(4) *Updating the I-Advisor slide*

All the possible goals the user could be working on are retrieved from the discourse state and sent on to the active instance of the **IAdvice** class in the AdaptIt application. This panel can then be updated accordingly.

- **DOMSerializer.java**

This class serializes a Document Object Model (DOM) and writes it to a file. It is used by the **DOMTreeWriter** class.

- **DOMTreeReader.java**

This class reads in an XML file containing a DOM Tree and stores all the strings from the tree in a data structure. Currently it can read in three types of files:

(a) The logs of previous user's actions with the domain expert's annotations (*log.xml*).
→ Stored in an object of the **AnnotatedExample** class

(b) The task models for the skill hierarchy (*skillHierarchy.xml*) and the blueprint domain (*bluePrint.xml*).
→ Stored in an object of the **TaskModel** class.

(c) The layouts for the I-Advisor panel in the AdaptIt tool, one for the skill hierarchy domain (*layoutskill.xml*) and one for the blueprint domain (*layoutblue.xml*).
→ Stored in an object of the **Layout** class.

- **DOMTreeWriter.java**

This class takes a certain data object as input, and returns a XML file containing that data object in a DOM tree. The two data objects the I-Advisor can write away are:

(a) Objects of the **TaskModelLearner** class which contain a learned task model.
→ Written away to a *taskmodel.xml* file

(b) Logs of previous user's actions, consisting of vectors containing all the actions the user performed in AdaptIt, as well as all the I-Advisor links the user clicked upon.
→ Written away to a *log#.xml* file, where # signifies the how many-th log it is in the *logs* directory.

- **IAdaptIt.java**

The IAdaptIt class contains two booleans, which determine the general behaviour of the I-Advisor:

(a) *LearnTaskModel* (design described in chapter 5)
If LearnTaskModel is set to true, then the I-Advisor will learn a new task model based on the logs of previous users' actions. This is done by creating an object of the **PreTaskModel** class.

(b) *StartIAdvisor* (design described in chapter 7)
If StartIAdvisor is set to true, the I-Advisor will add listeners to the AdaptIt application and follow the user by maintaining a discourse state. This is done by creating an object of the **IAdvisor** class.

- **IAdvisor.java**

The IAdvisor class is the part of the I-Advisor, which listens to all of the user's actions in AdaptIt and updates the discourse state based on these actions.

It contains two booleans, which describe its behaviour:

(a) *writeLog* (design described in section 7.1)
If writeLog is set to true, then the I-Advisor will write away all the user's actions to a log file when the user closes the AdaptIt tool. This log file is a XML file and is written in the *logs* directory.

(b) *updateDiscourseState* (design described in chapter 6 and section 7.2)
If updateDiscourseState is set to true, then the I-Advisor will update the discourse state each time the user does an action. If you are interested in only logging a user's actions and not in advising the user, then this boolean is useful.

An object of the IAdvisor class will perform the following steps:

(1) Add listeners to the SelectionController and ObjectUpdateManager.
(2) An object of the **DiscourseState** class is instantiated.
(3) Each time an event is received from the listeners, two things happen:
  (a) The event is stored in an object of the **PrimitiveAction** class.
  (b) The object of the **DiscourseState** class is updated based on this event.

- **Plan.java** (design described in chapter 6)

The Plan is part of the discourse state, and consists solely of a Hashtable. This Hashtable stores which of the user's actions contributed to which goals. The goals are the keys of the Hashtable, the user's actions are stored in Vectors in the Hashtable. An object of the Plan class can be viewed as some form of history of the user's previous actions and goals.

- **PreRecipe.java** (design described in chapter 5)

The objects of the PreRecipe class are only used during the learning of the task model to store the learned recipes. Whereas objects of the Recipe class consists solely of the goal and a vector containing the steps which have performed to accomplish that goal, the PreRecipe class contains a lot extra methods which are used to learn recipes.

These methods are used to accomplish the following:

(a) *Create the steps of new recipe*
The steps of a new recipe are learned from the user's actions and the segments added by the domain expert.

(b) *Learn optional steps*
It is learned which steps are optional, and which are not.

(c) *Learn the equality relations (also called propagators)*
The equality relations define which parameters of the user's actions have to equal for a goal to be achieved successfully.

- **PreTaskModel.java** (design described in chapter 5)

The PreTaskModel class learns a new task model. This consists of 3 steps:

(1) *Reading in all the XML log files*
All the XML log files contained within the logs directory are read in using an object of the **DOMTreeReader** class. The users' actions and the domain expert's annotations of these actions contained in these files are stored in an object of the **AnnotatedExample** class.

(2) *Learning a new task model based on these logs*
This is done by creating an object of the **TaskModelLearner** class.

(3) *Writing the learned task model away to a XML file*
The file is written away using an object of the **DOMTreeWriter** class.

- **PrimitiveAction.java** (design described in section 5.1)

The objects of the PrimitiveAction class store the actions performed by the user in AdaptIt. They store these events received from the SelectionController and ObjectUpdateManager using the following two variables:

(a) *The name of the function*
This name is a combination of the properties of the event received, namely the type of the object (Object Update or Selection Event), the class of the Object updated or selected and the updating Object or Component.

(b) *The parameter*
This is the name of the object, which was selected or updated.

- **Recipe.java** (design described in section 5.1)

The objects of the Recipe class are used to store the recipes which together form the task model. They consist mainly of three parts:

(a) The name of the goal that this recipe describes.
(b) A Vector containing objects of the **Step** class which have to performed to accomplish the goal.
(c) A Vector containing the other possible recipes that also describe this goal.

- **Segment.java** (design described in section 5.2)

A Segment is a sequence of actions or other segments, which together accomplish the same goal. An object of this class stores:

(a) The *SegmentType* (the name of the goal)
(b) A Vector containing all the *SegmentElements* (the actions or other segments, which together accomplish the goal).

- **SegmentElement.java** (design described in section 5.2)

The SegmentElement object is an element of an object of the **Segment** class. A SegmentElement can either be a primitive action or a segment itself. It stores the class of the element (either **primitiveAction** or **Segment**), the type of the element (the function in case of a **primitiveAction**, segmentType in case of a **Segment**), and a mapping from the element to a step in the corresponding PreRecipe.

- **SegmentSet.java** (design described in section 5.3)

A SegmentSet consist of set of segments of the same type. It is used in one of the first steps of the learning of a task model, called alignment.

- **Step.java** (design described in section 5.1)

A Step is an element of a **Recipe** and describes an action, which has to be performed before a goal can be accomplished. A step can be either a primitive action, which can be accomplished directly in the AdaptIt tool, or a non-primitive action, in which case a recipe describes how to accomplish that non-primitive action.

A Step consists of three parts:

(a) A String named *type* containing the type of the step
(b) A boolean named *primitive* signifying whether or not this step is a primitive action.
(c) A boolean named *optional* signifying whether or not this step is optional.

- **TaskModel.java** (design described in chapter 5)

A TaskModel consists of two parts:

(a) A Hashtable containing the recipes (objects of the **Recipe** class).
These recipes are stored in the Hashtable using their goals as keys.
(b) A Hashtable containing the primitive actions (objects of the **Primitive Action** class)
This Hashtable contains all the primitive actions a user can do within a certain part of the AdaptIt tool.

- **TaskModelLearner.java** (design described in chapter 5)

The TaskModelLearner class learns a new task model from input annotated examples (objects of the AnnotatedExample class) . The learned task model consists of recipes (objects of the PreRecipe class). The learning of the task model consists of the following phases:

(1) *Alignment Step 1:*
The segments from the annotated examples are sorted into sets of segments, which are of the same segmentType and in which each segment is a subset of the other segments in the set.

(2) *Alignment Step 2:*
The steps of the preRecipes are created for each set of segments.

(3) *Induction of the Optional Steps:*
The optional steps of the PreRecipes are learned.

(4) *Induction of the Equality Relations:*
The equality relations between the parameters of the steps in the PreRecipes are learned.

Classes in com.adaptit.components.advisor

- **IAdvice.java** (design described in chapter 7.3)

This class initialises and updates the I-Advisor's panel in the AdaptIt tool. When a new or existing project in AdaptIt is opened an object of this class is created and the I-Advisor is started up by creating an object of the **IAdaptIt** class. Each time the user does an action in AdaptIt, the panel is updated based on the possible goals the I-Advisor has learned the user could be working on.

The contents of the I-Advisor's panel are links to the pages in the full Advisor. The layout for the panel, including descriptions of the different links, is contained in two XML files found in the PropertiesDirectory. The first XML file, *layoutblue.xml*, is for the bluePrint domain, the other one, *layoutskill.xml* is for the skill hierarchy domain.

If the I-Advisor has deduced the goal the user is working on, the link to the Advisor page corresponding to this goal is highlighted. If there is more than one possible goal, then the panel

will only show an overview of links corresponding to the different steps in the domain the user is working in.

- **Layout.java**

A Layout object stores the HTML layout of the I-Advisor's panel. It consists mainly of descriptions of the links to the full Advisor pages in the form of objects of the **HelpMapping** class.

- **HelpMapping.java**

A HelpMapping describes a link to a full Advisor page, which is to be displayed in the I-Advisor's panel. It consists of three parts:

(a) The name of the goal, for example *"AnalyseComplexSkills"*
(b) The text to be displayed in the panel, for example "*Analysing Complex Skills"*
(c) The link to the page in the Advisor corresponding to the goal, for example "*[a2]Analyse_the_complex_skill"*.

Files in the PropertiesDirectory

- **bluePrint.xml**

This XML file contains a DOM tree with the task model for the bluePrint part of AdaptIt.

- **skillHierarchy.xml**

This XML file contains a DOM tree with the task model for the skillHierarchy part of AdaptIt.

- **layoutblue.xml**

This XML file contains a DOM tree with the HTML layout for the I-Advisor's panel in case the user is working in the blueprint part of AdaptIt.

- **layoutskill.xml**

This XML file contains a DOM tree with the HTML layout for the I-Advisor's panel in case the user is working in the skillHierarchy part of AdaptIt.

- **AdaptitEditorPreferences.xml**

I only changed two lines in this file to start up the I-Advisor instead of the Q-Advisor, and to make the I-Advisor the first of the tabbed panes in the bottom-left corner.

Traceability of the Requirements on the Implementation

The Requirements:	**Satisfied in:**
Functional Requirements	
(1) *The support given by the I-Advisor shall consist of links to the pages in the help system corresponding to the user's current task context.*	**Classes used to design the support:** IAdvice.java, Layout.java, HelpMapping.java
(2) *This support given by the I-Advisor shall be based on:*	
• *the user's actions within AdaptIt*	**Class used to listen to user's actions:** IAdvisor.java **Class used to store user's actions:** PrimitiveAction.java
• *task model*	See functional requirement 3.
(3) *The I-Advisor shall contain a task model, which describes a small part of the AdaptIt application.*	**Actual task models:** bluePrint.xml, skillHierarchy.xml **Classes used to store task model:** TaskModel.java, Recipe.java, Step.java, PrimitiveAction.java **Classes used to learn task model:** TaskModelLearner.java, PreRecipe.java, PreTaskModel.java, AnnotatedExample.java, Segment.java, SegmentSet.java, SegmentElement.java

(4) *A domain expert shall be able to view and annotate the logs of previous users, in order to aid the learning of the task model.*	**Classes used to write the logs:** IAdvisor.java, DOMSerializer.java, DOMTreeWriter.java, PrimitiveAction.java **Classes used to read in and store the logs with the domain expert's annotations:** DOMTreeReader.java, AnnotatedExample.java, Segment.java, PrimitiveAction.java
Technical Requirements	
(5) *The I-Advisor shall be implemented in Java just like the AdaptIt tool.*	All classes are implemented in JAVA.
(6) *The I-Advisor shall operate within a small part of the AdaptIt application*	The I-Advisor operates within both the Skill Hierarchy and the Blueprint domains of Adapt-It. It contains task models for both domains, as well as layout for the support given in the I-Advisr's panel. **The task models for both domains:** skillHierarchy.xml, bluePrint.xml **The support layouts for both domains:** layoutblue.xml, layoutskill.xml
(7) *No changes within the AdaptIt application shall be made to make interaction between the AdaptIt application and the I-Advisor possible.*	No changes were made in the AdaptIt application.
(8) *The I-Advisor shall interact with the AdaptIt application in the same way as the Q-Advisor.*	In the **AdaptitEditorPreferences.xml** the Q-Advisor is replaced by the I-Advisor, and they interact with AdaptIt in the same way.