Information Presentation in an Automatic Speech Processing System

Thomas Oldervoll

14 April 1998

Abstract

The design and implementation of the information presentation of Alparon system is presented. Alparon is an automatic speech processing system for offering train time table information by telephone. Tasks covered include formulating and executing queries from the information given by the customer, translating between data formats used in databases and the format chosen in Alparon, storing travel plans, choosing the best plan and preparing a presentation of this plan. Emphasis is put on the strategy for presenting travel plans, which is based on a study of how human operators perform this task [Rats, 1997b]. A mechanism for specifying templates for the presentations has been implemented. The effectiveness of the presentation strategy have not been studied, however, since testing requires other parts of the system yet to be implemented.

Preface

This report documents work done as the final project for the degree of sivilingeniør[1] in Computer Science. This work constitutes the end of the author's studies at the Norwegian University of Science and Technology (NTNU), in Trondheim, Norway.

The work was performed at Delft Technical University (Delft, the Netherlands) as part of the ERASMUS student exchange program. Supervisor in Delft was Robert van Vark, who is currently working on his Ph.D. thesis on the Alparon dialogue manager. Supervisor in Trondheim was assoc. prof. Tore Amble.

[1] Roughly equivalent to Master of Science.

Contents

Chapter 1

Introduction

My thesis work has been to design and implement the information presentation part of Alparon. The Alparon project develops dialogue management for the next generation of automatic speech processing systems. Current task is to develop a prototype for a system where the public can access train time table information via telephone. Typically, a customer will call the system and ask when the train leaves, stating where she wants to travel and when. The system will be fully automated, applying speech recognition and speech synthesis to converse with the customer. Focus in the Alparon project is on dialogue management (leading a conversation with the customer) since this is believed to be a crucial part for a successful system. The overall design was more or less decided before my work started and is explained in [Vark, 1997a].

The goals for my work has been to design and implement the modules needed for information presentation in Alparon. I was to assume that the customer had explained her travel plans in sufficient detail already, i.e. where and when to go. My task was then to provide the following functionality:

- Retrieve time schedule information from databases.
- Store travel information.
- Choose the connection best fitting the customer's needs.
- Prepare a presentation of this connection.

Design and implementation of these modules had to take into account ideas developed by Mieke Rats. As part of the Alparon project, Rats has developed a theory on how to present travel information [Rats, 1997b]. Rats shows how current automated systems present travel plans in a fundamentally different way compared to the human operators. She argues that the Alparon system will become more usable if it presents information in a more human-like manner. My work should implement some of these ideas.

Sentence generation and the production of actual speech is outside the scope of this thesis.

Being a thesis the purpose of this report is mainly evaluation of the work done. It is assumed that the readers have general computer science knowledge. Some knowledge of automated speech processing systems might also be useful, but basic theory will be introduced.

This report will present my work, the Alparon project and some relevant theory. The next chapter will explain autoamtic speech processing systems in general. In chapter 3 the Alparon project and system will be presented. Chapter 4 defines my part of the work more closely, outlining the information presentation part of the system. The three following chapters documents the three modules I have implemented: Database Manager (chapter 5), Travel Board (chapter 6) and Query Travel module (chapter 7). The Query Travel has been implemented in the Java programming language. A framework for implementing ALparon dialogue management modules in Java was developed, and this is desribed in chapter 8. Finally, chapter 9 outlines future work before the report ends with a summary. The appendixes will give and code listings and source documentation for the implemented modules.

Chapter 2

Automatic Speech Processing

In the Alparon project we aim at developing an automated speech processing system for information retrieval dialogues (i.e. the customer talks to the system with purpose of getting some information). This chapter explains what is meant by the term automatic speech processing and gives a general model for such a system. At the end of the chapter I will present some theory on the information presentation part of such a system, a central aspect of my work.

2.1 Talking to Machines

Speech is used for most of everyday communication between humans. No wonder, then, that it has been proposed that this should be the way to communicate with machines as well. Science fiction writers, as professional future speculators, often let people communicate with computers using speech - some of the most well-known examples are “2001 A Space Odyssey” and ”Star Trek”. However, reality cannot match this level of sophistication. Some even claim that there are philosophic reasons why a “conversing machine” can never be built [Button et al., 1995]. Still, there are lots of examples of useful applications using speech input and/or output.

2.1.1 Use of Speech Technology

Speech technology is being used in an increasing number of commercial products. Research projects try to adapt the technology to even more tasks.

Speech used in Desktop Applications

Speech is an increasingly popular modality for desktop application, especially for PC application. Speech input and/or output in such applications often come as an addition to traditional modalities like keyboard and mouse input and graphical output. Since PC applications have a potential of being sold in a large number of copies, such applications are becoming increasingly commercial. Two important categories are:

- **Dictation.** Lets users speak text into a word processor or other applications. The two products most often quoted as the leading in dictating are Dragon NaturallySpeaking and IBM ViaVoice. Use of dictation is popular in medicine, where hands may not be available for typing, and in law, as a substitute of a typing of secretary. An example of how speech technology is going mainstream: both of the two leading dictation products will be bundled with popular word processors in near future (Dragon with Corel WordPerfect and IBM with Lotus WordPro.

- **Command-and-control for PC programs.** Lets users give commands to their PCs by voice. Example: Kurzweil VoiceCommands, for controlling Microsoft Word with speech.

Automated Telephone Services

The telephone is an important medium for offering inforamtion services. Telephones can be found everywhere, and especially use of mobile phones is increasing rapidly. Many automated telephone services offer information via telephone, from allowing people to handle their personal bank accounts via the telephone to offering weather forecasts or stock quotes. Many of the existing services use the telephone keys as input medium and speech as output medium. As technology improves, speech is increasingly used also for input. There are not very many fully commercial services available, but several domains are investigated in research:

- access email, calendar and other desktop applications by using telephone [Yankelovich et al., 1995].

- automatic directory of phone numbers [Seide and Kellner, 1997].

- home banking [Larsen, 1997].

Even though speech technology has proven useful in a number of applications, it is not always appropriate for the task. For desktop applications speech is just another medium along with text, graphics, keyboard, mouse etc. Cohen and Oviat lists some areas where speech is particularly advantageous [Cohen and Oviatt, 1994]:

- When the user's hands or eyes are busy

- When keyboard input or screen output is limited or not available (as in telephone applications or hand-held computers)

- For disabled users

2.1.2 Why bother?

Lots of energy has been spent on making machines that can lead a conversation. For linguistics the goal might be to gain knowledge of language and human use of language. For computer engineers it might be to make a working application for a particular task. Allen summarises the goal of research in computer understanding of human language as "to create computational models of language in enough detail that you could write computer programs to perform various tasks involving natural language" [Allen, 1995] (p.1). This might again be seen a step on the path to the greater goal of making a general Artificial Intelligence. By taking this approach we would be following Nilsson's recommendation of "keeping the eye on prize" [N.J. Nilsson, 1995]. Nilsson criticises the field of Artificial Intelligence for having forgotten the original goal of producing a general-purpose intelligent program in its search for better performance on highly specialised tasks. However, Allen's general goal is criticised by Button et al. in [Button et al., 1995] [1] Button et al. claims that Allen's goal is based on a false understanding of human language use.

I choose to take a human-computer interaction approach, where the goal is to make more usable computer programs. The rationale behind using natural language[2] is then to utilise skills people have gained through everyday interaction with humans also in interaction with computers.

2.2 Automatic Speech Processing

A wide number of terms are used to describe systems that employ speech technology. In the Alparon project we have chosen to use the term Automated Speech Processing (ASP). An ASP system is a system that uses speech as input and output modality and performs actions dependent on the content of the speech input. Other modalities like keyboard input and graphics output might be used in addition to speech, but we focus on speech-only systems. The criteria that actions performed depends on speech contents means that speech has to be interpreted as words and those words have to be analysed to find the meaning in the utterance. This separates an ASP system from a voice mail application, where speech is only stored as sound and no attempt is done to figure out which words are being spoken, and from dictation systems, where speech is interpreted to find the correct words but not analysed to find the meaning intended in the speech.

A speech-only ASP system typically receives input as speech, performs actions dependent on the meaning of the input, and outputs speech dependent on the results of the actions. The main stages might be divided like this:

- Receive speech input through a microphone.

[1] Button et al.'s critique is based on the first edition of Allen's book. However, the paragraph in question is kept unchanged in [Allen, 1995] which is the second edition.

[2] The term "natural language" refers to languages used by humans, as opposed to a artificially constructed formal languages like programming languages.

- Find out which words are being spoken (Speech Recognition).
- Interpret the words to find their meaning (Natural Language Processing).
- Perform actions dependent on the meaning, and formulate an appropriate answer.
- Output speech through a loudspeaker (Text-To-Speech)

A general model for an ASP system is given in figure 2.1.
The parts of the model are outlined below.

Figure 2.1: A general model of an ASP system

2.2.1 Speech Recognition (SR)

Speech Recognition is the process of converting speech in the form of a sound signal into a sequence of written words. This can range from the relative simple task of recognising the difference between "yes" and "no", to the considerably more difficult task of transcribing normal, fluent speech. Speech recognition is often classified in a number of dimensions:

- **Continuous vs. discrete:** In discrete speech recognition, the speaker has to insert a short pause between words.
- **Speaker dependent vs. speaker independent:** Speaker dependent recognition requires the user to go through a training session before the

system is able to recognise her voice. Speaker independent recognition can recognise any voice the first time. In between we might add speaker adaptive systems, that can recognise words the first time it hears a voice it hears it but performs better if it is allowed to train on the voice. Such a system will improve over time as it is learns more about the user's voice.

- **Large vs. small vocabulary:** Vocabulary size refers to the number of different words the system can recognise. This might range from 2 to tens of thousands.
- **Sound quality:** The system might require CD sound quality to be able to operate or might be satisfied with low-bandwidth telephone sound with lots of background noise.

State-of-the-art dictation systems claims to offer continuous, speaker adaptive, large vocabulary recognition, requiring relatively high sound quality. A public information service by telephone would require speaker independent, continuous, medium vocabulary recognition with low-quality sound.

The process of going from speech to words is complex and varies slightly between systems. To make a large vocabulary possible, it is common to use phonemes (the basic meaningful sounds units in a language) as units of recognition, and not words. The principle workings of a typical system can be summarised like this:

- **Feature extractions:** Pre-processing of the speech signal, extracting the important features into feature vectors.
- **Phoneme recognition:** Based on a statistically trained phoneme model, the most likely sequence of phonemes (the basic meaningful sounds units in a language) is calculated. This is usually done using Hidden Markov Models (HMMs).[3]
- **Word recognition:** Based on a statistically trained language model similar to the phoneme model, the most likely sequence of words are calculated. HMMs are usually used for this, as well.

Speech recognition performance can be measured in recognition rates, i.e. how high percentage of the input words are recognised correctly. There will always be recognition errors, so therefore some speech recognisers can output the n best (where n is a number, say, 4) matches with associated probabilities. The most likely match can then be chosen by the natural language processing part (see section 2.2.2) or even the dialogue manager (see section 2.2.3), taking context into account.

Not all information in speech is captured when it is written down word-for-word. Humans also get many clues from interpreting how the sentence is said. Some of these clues are collectively known as prosody. This includes intonation,

[3] Hidden Markov Models where first applied to speech recognition by Jim Baker [Baker, 1976], who later founded Dragon Systems.

speech rate and intensity [Allen, 1995]. At present, very few speech recognisers try to capture prosody information. For an example of one attempt, see [Lieske et al., 1997].

Speech cognition uses techniques from signal processing and statistics. The field therefore has an engineering flavour with a practical emphasis on reducing error rates. Lately it has also become a commercially important field, leading to secrecy around and copyright protection of techniques and models used in actual systems.

2.2.2 Natural Language Processing (NLP)

The task of the natural language processing sub-system is to translate a sentence into a logical form representing the context-independent meaning of the sentence. In practice this means translating from a string of words into some logical language. Since the logical form should be context-independent, it must cover all reasonable interpretations that could be valid in some situation. It is then the task of the dialogue manager to choose the right interpretation given the context.

NLP systems can be classified by the level of understanding they try to achieve from the input sentence:

- Word spotting: Looks for certain keywords and ignores all other words.
- Template matching: Tries to fit the input words to one of its templates. Each template specifies a short phrase but might contain unspecified words like a name. Words not fitting any templates are ignored.
- Full parse: All words have to be understood, otherwise the system rejects the sentence. Builds a syntactic parse tree to represent the function of each word in the sentence and uses this tree to make a representation of the meaning of the sentence.

The NLP system uses a lexicon which lists all known words and their meaning in logical form. A full parse system will also use a grammar that specifies syntactically correct sentences through a set of logical rules.

The field of Natural Language Processing has traditionally be focussed on written language. However, it is generally agreed that speech recognition could be improved if it had an understanding of the meaning of the sentence. [White, 1990] refers to an experiment where ordinary speech was recorded, words where played back to test persons in random order. Only 70% of the words were understood, even though 100% was understood when played in the correct order. This indicates that humans often need an understanding of the meaning of the words to be able to recognise them. Still there has traditionally been little coordination between SR and NLP research. White argues that this is one of the main challenges for achieving a computer understanding of speech. If a commercial speech recogniser is used, this integration is often made difficult. The inner workings of the speech recogniser is kept secret, making it impossible to utilise information from the NLP subsystem. The best one can achieve is often to let the speech recogniser output the n best words and let the NLP system choose between these. One of the reasons behind lack of integration of SR and NLP (apart from technical limitations like runtime considerations) might be a difference in research culture. Whereas speech recognition research often has had a practical outlook, natural language processing work has often been more theoretical. Achieving full coverage of every phenomena in language may seem more important than making a practically working system. However, as commercial systems pop up we might expect that practical considerations will come more in focus also in NLP.

2.2.3 Dialogue Manager (DM)

The dialogue manager serves a number of different purposes in our ASP model:

- **Adding context:** Deciding the final interpretation of input utterances, taking dialogue phenomena into account. An outline of some of these phenomena will be given below.

- **Problem solving:** Perform any actions and reasoning necessary to give an appropriate answer to the customer. The purpose of an ASP system is usually to offer some service to the user, like providing information or performing some action. When the meaning of the user's utterance is understood, the dialogue manager performs the needed action. Problem solving might be done by a separate module (often called "the application") on request from the dialogue manager.

- **Response generation:** Decide what to say to the customer. This includes both deciding the content of the next system utterance (what information to include) and its the form (what words to use). Usually this means formulating a sentence depending on the results of the problem solving.

Phenomena making dialogues difficult

Much of the work in natural language processing has gone into understanding sentences. However, in a spoken dialogue there are many phenomena that go beyond the level of a single sentence. These include [Oldervoll, 1997]:

- **Ellipsis:** Part of the structure of a sentence is omitted. The missing parts must be inferred by the listeners using previous utterances and other context information. Example: "Make it three". (If the previous utterance was "give me two beers, please", the interpretation is clear.)

- **Anaphora:** The use of referential words like "he", "that", "then" etc. These words usually refer to persons, things etc mentioned earlier in the dialogue. Example: "Make it a double". ("It" might refer to a whiskey.)

- **Indirectness:** People often mean something else or something more than the literal meaning of their utterance. A classic example is the utterance "it's cold in here" which may be interpreted as a request to close the window. This phenomena has led to the Speech Acts theory, invented by Austin [Austin, 1962] and developed further by Searle [Searle, 1969]. (See section 3.2.2 for more about speech acts.) To continue the bar example, "I could really need a whiskey" would be a valid order for a co-operative bartender.

- **Conversational fillers:** People will often fill in short phrases like "yes", "uh-huh" etc that do not have any literal meaning. The function of these utterances seems to be more communicative, like showing that one is still paying attention. In the bar example we might expect the bartender to use lots of conversational fillers as his customers pours all their personal problems on him.

- **Repairs:** People often re-decide in the middle of an utterance. New information might be given that is intended to override previously given information. Example: "Give me two, no, three whiskeys, please".

- **False starts:** Often people starts sentences that they never finish. Example: "I just feel so... But still... My wife just left me."

- **Turn-taking:** In a conversation people are expected to take turns in talking. To talk while others talk is considered impolite, and long pauses in the conversation will often feel uncomfortable. Humans seems to have no problems deciding when it might be appropriate to change turns in a conversation, but this has proved very difficult to formalise for computer use. Much of the work in formalising turn-taking has been base on [Sacks et al., 1974]. The use of this work as a basis for a formalisation of turn-taking is criticised in length in [Button et al., 1995].

Many of these phenomena occur for reasons of economy - utterances are shortened to speed up the conversation, relying on listeners to fill in the gaps using context. Other phenomena arise for politeness reasons, as when a request is disguised as a factual statement ("I could need a hand with this suitcase.").

Initiative in dialogue

Dialogue Managers can be classified by how they handle initiative:

- **System-driven:** The system has initiative. The system asks questions and the user answers.

- **User-driven:** The user has initiative. The user asks questions and the system answers.

- **Mixed-initiative:** Both the user and the system can take initiative by asking questions or introducing new topics in the dialogue. There are varying degrees of mixed initiative. For example, the system can have predermined initiative strategies (as when the user has initiative in query phase and the system has initiative in presentation phase). A more flexible system would let initiative to be "negotiated", allowing both parties to take initiative at any time [Allen, 1997].

One often wants an ASP system to be a mixed initiative system. The user should be allowed to user speak freely, but the system should take initiative when needed to clarify details, if the dialogue is about to fail etc. Still, many systems use a strictly system-driven dialogue. This makes user utterances more predictable, something that makes speech recognision easier.

2.2.4 Text-To-Speech (TTS)

The Text-To-Speech subsystem (also known as "speech generation") realises as sentence as speech, i.e. translates from a string of written words to a sound signal.

There are two main techniques currently used for text-to-speech conversion:

- Speech concatenation: Pre-recorded words are concatenated to form the output sentence. Apart from intonation this sounds just like a human since the words are spoken by a real human voice. The disadvantage is that speech output is restricted to the words that have been pre-recorded. Another disadvantage is the work needed to record all the words.

- Speech synthesis: Speech is produced through software simulation of the speech production process in humans. This is a complex process, but it has the advantage that the system is able to speak any word and does not have to be restricted to a fixed vocabulary. The speech produced in state-of-the-art systems is quite intelligible but still sounds a bit metallic and "robot-like". Problems may arise with words that are not pronounced according to the standard rules, for example words imported from another language. Many TTS system uses a pronunciation dictionary that gives the special pronunciation for such words. However, if unrestricted text is input to the system, an unknown dissident word may appear and will be wrongly pronounced.

Text-To-Speech has been used successfully for years as an aid for blind people. In the last years it has also become popular in telephone applications. Even though one can easily hear that computer speech is not human, the performance of Text-To-Speech is already good enough to make practical applications possible. Humans are namely superior to computers when it comes to adapting to less than optimal conditions for understanding. Also, we might not want a computer to sound perfectly human. Having a slight machine-like characteristic helps the user keep in mind that she is talking to a machine.

Chapter 3

Alparon

In this chapter Alparon project will be presented. This project is now involved in making a prototype for a dialogue manager. I will refer to this prototype as the Alparon system. The problem to be solved by this system will first be presented, along with a history of the project. An outline of the design of the system will then be given with an emphasis on the parts I have been responsible for.

3.1 The Alparon project

Alparon is a project at the Knowledge Based Systems group of the Department of Technical Mathematics and Informatics at Delft University of Technology. The Alparon project aims at developing dialogue management for the next generation of automatic speech processing system. Current domain is to provide time table information for trains, and I will limit my focus here to this domain.

3.1.1 Providing train time-table information by telephone

Alparon started in the early winter of 1994 when cooperation with the company Openbaar Vervoer Reisinformatie[1] (OVR) started. OVR offers information on all public transportation (trains, busses, ferries, taxis) in the Netherlands as a telephone service to the public. Typically the customers want to travel somewhere and want information about how to travel, departure times, arrival times, ticket prices etc.

Manual solution: Human operators

OVR employs a staff of over 400 human telephone operators to offer this service. The operators use a database query system to find the information needed. In total, more than 12.000.000 calls a year are handled [Manintveld and Rothkrantz, 1997].

[1] Public transport travel information

Automated solution: VIOS

This service is very popular, so customers often have to wait to be served. Therefore, OVR wants to supplement human operators with an ASP system. For this purpose a system called VIOS has been developed. Since vocabulary size is an important factor for the difficulty in making an ASP system, it was decided that the automated system should only handle travel by train. VIOS has a restricted, system-driven dialogue structure. The system uses Philips speech technology and is based on a system in use for the Swiss railways [Aust et al., 1994]. VIOS was made available to the public in a limited filed trial in January 1998. Nation-wide deployment will come later.

Improved automated solution: Alparon

The Alparon project was given the task of evaluating customer appreciation of VIOS. Also, a dialogue manager for a next-generation ASP system should be developed since it was believed that dialogue management is crucial for the success of such a system.

3.1.2 History of the project

Alparon started with the deal with OVR in the winter of 1995. In spring 1995, a corpus of dialogues was collected. This was done by recording 5000 dialogues between the human operators and customers and transcribing them word-for-word. A coding scheme to capture the information in the dialogues was developed, and 500 of the dialogues where coded using this scheme [Vark et al., 1997a].

In the autumn of 1995 a Wizard-of-Oz experiment was held to study user appreciation of an ASP system and evaluate different strategies for dialogue management [Manintveld and Rothkrantz, 1997]. In Wizard-of-Oz experiments a human "Wizard" simulates the system. The test users believe they are interacting with a computer, but actually all responses are generated by the human wizard. By varying the dialogue strategies followed by the wizard, it is possible to evaluate different strategies even before they are implemented.

After an analysis of the coded dialogues, the coding scheme was adapted for use in an automatic coder [Rats, 1997a]. A combined parser and coder was made that translated from input sentences to this coding scheme [Rats and Vreught, 1997]. Instead of trying to make a general parser that could handle unrestricted language the parser/coder was designed to handle the vocabulary and grammatical constructions the test users used in the Wizard-of-Oz.

Using the coded corpus a range of analyses was performed. The dialogues where classified according to the level of sophistication it would require from an ASP system to handle them [Vark et al., 1997b]. The prompts used by VIOS where analysed [Vark et al., 1997a]. The way the operators presented information was studied and compared to the presentation in VIOS (as mentioned in the introduction) [Rats, 1997b].

In the summer of 1997 work began on designing and implementing a prototype for a dialogue manager that would allow a more flexible dialogue than

VIOS. An overall design was developed [Vark, 1997a]. A protocol for modules running on potentially different computers co-operating through message-passing was developed [Vark, 1997b, Vark, 1997d]. In the autumn of 1997 the implementation of the prototype started, and this work is still in progress.

Future plans include finishing the dialogue manager implementation. A demonstrator using text input and output will be set up. One possibility is to use the World Wide Web for this, similar to how it is done in a project in Trondheim [Bratseth, 1997] but with dialogue capabilities[2]. More work is needed on speech input and output before this can be integrated with the dialogue manager.

Furthermore, the dialogue manager should be adapted to more domains. This will require some effort in rewriting parts of the system, but the much of a system should be directly applicable to other domains. We would like to focus on building relatively simple but highly reliable system since we see it as important to gain the trust of the public if automated speech processing shall gain acceptance.

3.1.3 Corpus-based approach

The approach towards language understanding we take in Alparon is to consider how people actual use language rather than trying to define the "real" meaning of expressions. This emphasis on the practical language use and colloquial meaning can be seen as being inspired by Wittgenstein [Wittgenstein, 1958]. Also, we view language (in our domain, at least) as being goal directed and as a way of performing actions. This is based on the speech acts theory of Austin and Searle mention in section 2.2.3 [Austin, 1962, Searle, 1969].

To study how people use language we base our work on corpora, collections of recorded language use. Rationale for this approach is discussed in [Twente95, 1995]. The following corpora are used in Alparon:

- Database log: The database queries done by the telephone operators have been logged from the service started in 1991. We have the log of one year, containing 11 million calls. This log does not tell us how people used language but merely what information they asked for.
- Transcribed corpus: 5000 dialogues is transcribed to text.
- Coded corpus: 500 of the transcribed dialogues were coded using the coding scheme developed for this purpose.

From the transcribed corpus we have got an indication of the vocabulary and language constructions involved in this task. From the coded corpus we have studied what level of sophistication an ASP system must be on to be able to cover each of the dialogues. The results indicated that even a simple scheme will cover the majority of the cases [Vark et al., 1997b]:

- 28%: The users presents all necessary information to perform a database query.

[2] Bratseth's system Bustuc is a pure question-and-answer system with no dialogue memory

- 60%: The operator asks for some missing information.
- 4%: The operator introduces extra information through suggestions.
- 1%: The query cannot be answered, mainly because the user does not have enough information (e.g. doesn't know exactly where she wants to go).

Work on the database log is still in progress [Vark et al., 1997c], [Vreught and Rothkrantz, 1997], but we hope to be able to use this data to predict information, hereby improving speech recognition rates. For example it might be possible to let the probability of arrival places be dependent on where the customer is travelling from.

The Wizard-of-Oz experiments also gave some interesting results [Manintveld and Rothkrantz, 1997]:

- A non-directive dialogue structure is highly appreciated (i.e. the user should be allowed to have initiative).
- Travel advice should be presented in small chunks rather than as one long utterance.
- The users strongly dislike apparent mistakes by the system.

3.1.4 Ethical considerations

The Alparon project tries to automate a service currently operated by humans. Replacing humans with machines is problematic, and doing research on this without the considering would in my opinion be irresponsible. Although what is presented here is clearly speculations without much scientific hold, I still think it is valuable to reflect on the consequences of the technology we are trying to develop.

Why

Why are we trying to automate the train table information service? From a strictly scientific point of view, developing new technology can be seen as a goal in itself, pushing the horizon of human knowledge further. However, the Alparon project would probably not have been started if OVR hadn't been paying for it. So why do OVR want to automate this service? Vark et al. seems to be worried for the well-being of the telephone operators, judging by the closing sentence of [Vark et al., 1997d]: "It is hoped that the ASP system will supplement the current operators such that their workload can be reduced." This is either naive or downright manipulative. With 400 operators and 12 million calls a year OVR is sure to adjust the number of operators working to the actual traffic, so there will always be a new call to answer for every operator. OVR's main reason for automating is of course money. A mature ASP system will be far cheaper than humans. Hiring people is expensive, and although the costs of developing an ASP system might be considerable it will be very cheap to run.

One might argue that adding an automated system would improve service to the customers by making information available 24 hours a day and eliminating queues. This is still a matter of costs since this could also be achieved by hiring more people. There might be technical problems involved in adding more operators (hardware restrictions on the number of lines, for example), but I doubt that these problems are significant compared to the problems of developing an ASP system. So what it all boils down to is that we are developing an ASP system because it is cheaper to replace humans with machines. This is no shocking insight, but it is important to keep in mind.

Approaches to foreseeing consequences

When commenting on the effects of new technology, authors often take a one-sided view, seeing either only the positive effects or only the negative effects [Kling, 1995]. Kling uses the term "technological utopianism" for the view of new technology as being the key enabling an ideal world. "Technological anti-utopianism", on the other hand, sees new technology as a key cause for human suffering. A more balanced view is what Kling calls "social realism". In this approach, empirical data is used to examine computerisation as it is actually practised and experienced. I will in the following try to outline possible future effects of a mature ASP system, as seen from the three different views. The social realistic approach is problematic, however, since I have no empirical data to build on. I will try to compensate for this with common sense, but the result is of course pure speculations.

An utopian view

By letting computers do routine task, jobs become more interesting. The work of the telephone operators is very repetitive and does not allow much initiative or creativity. With a successful ASP system in place, the number of human operators can be reduced. All standard questions will be handled by the system, but it should be possible to talk to a human operator for special requests. The remaining operators will then get more interesting work, whereas the rest will get other, more fulfilling work.

The technology developed will have positive effects outside of the train table domain. Speech technology will make computer interfaces more "natural" in the sense that no special skills will be needed to use a computer. Skills acquired from everyday communication with people will be enough to interact also with computers. The contrast between those who are "computer literate" and those who are not will then be gone as everyone have access to fruits of the information revolution.

There will also be additional scientific benefits. We will gain more knowledge of language and language use. The dialogue capabilities will be useful for work on that big goal of the field Artificial Intelligence, namely implementing *an* Artificial Intelligence, an autonomous agent with general problem solving and communication capabilities.

An anti-utopian view

Telephone operators will loose their job. If an ASP system is to be successful, it will have to be cheaper than human operators, and this implies firing people. There will also be some new jobs created, for example in developing and maintaining the system. Few of the telephone operators will be able to compete for these jobs, however, since they require higher education and very specialised skills.

In general, automatisation will lead to a greater divide between skilled and unskilled workers. The requirements for being classified as skilled will be ever increasing, and those who qualify will form a new upper class. The rest will compete for an decreasing number of unskilled jobs as routine tasks are taking over by machines.

For OVRs customers, the quality of the service they are offered will become lower. ASP systems will be inferior to humans in communication tasks, at least for the foreseeable future, so misunderstandings will be inevitable. One of the results of the Alparon Wizard-of-Oz experiments was that people preferred talking to a human [Peters and Kock, 1997]. This is consistent with a recent survey done by Dybkjær et al [Dybkjær et al., 1995].

Widespread use of speech technology might also have unwanted effects for language use. Language might become more standardised and less rich as people adapt their way of talking to make the computers understand them. This lead to standardisation of dialects and ways of expression.

A social realistic view

The effects mentioned so far might give speech technology too much importance. Success of ASP will probably lead to new services rather than replacing operators since ASP cannot match the flexibility of a human. Services that van be automated can be offered at a lower price, compensating for the loss of quality. Still, reflecting on possible positive and negative consequences will be important when developing this new technology.

3.2 The Alparon System

Focus in the Alparon project is on improving dialogue management [Vark et al., 1997d]. The design of the dialogue management in the Alparon system was based on a number of prerequisites [Vark, 1997a]:

- mimicking important strategies found in our human-human corpus
- modularity to allow flexible development
- extendibility to be able to incorporate future developments

This section will outline the design of the Alparon system. A sketch of the overall system will be given before the dialogue manager is explained in more detail. An implemented framework for module communication will be presented. This is all based on [Vark, 1997a]. The coding scheme used for representing the meaning of both user utterances and system utterances will be outlined in the end of the section.

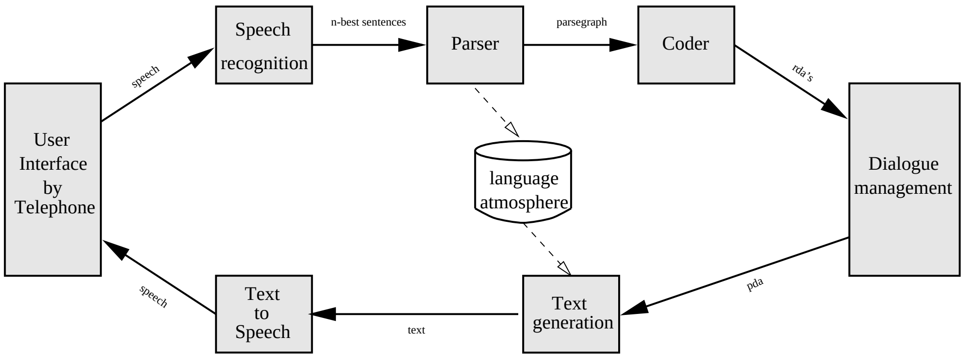

Figure 3.1: Overall model of the Alparon system

3.2.1 A model of the Alparon system

Figure 3.1 shows a model of the overall Alparon system. This model in similar to the general ASP model of figure 2.1, but a few notes can be made:

- Speech input and output is provided through a telephone interface.
- The Natural Language Processing part in Alparon consists of a parser and a coder.
- A language atmosphere database is placed between the parser and the text generation module.

Since the different parts of this model work much like describe for the general ASP model, the description here will be brief (see [Vark, 1997a] for more details).

User interface by telephone

Speech input and output will be done by telephone, using a-law, the European coding standard for telephone communications. Telephone speech offers a limited sound quality, a fact that will have consequences for the speech recogniser.

Speech recognition

The speech recogniser will need to offer continuos recognition with a medium vocabulary (a few thousand words), given the restrictions of the telephone interfaces. It should offer output in the form of n-best sentences for some small number n.

Parser

The parser takes n-best sentences as input, checks for grammatical correctness, and outputs a parse graph. This parse graph contains all possible combinations of uttered words. The parser will have to be robust, being able to compensate for speech recognition errors. It will also need to take into account phenomena of spoken language, such as accepting single words, sentence fragments, incomplete sentences, repairs and false starts.

In addition, we want the parser to extract characteristics of the language of the customer and store this in a "linguistic atmosphere" database. This database will be used to adapt the output of the system. Through this we hope to gain higher customer acceptance.

Coder

The coder will take the parse graph from the parser as input. Output will be a set of recognised dialogue acts (rda's) with probabilities above a certain threshold. The coder only has access to the sequence of words of the input so it can not distinguish between all possible meanings of the utterance. The dialogue manager will have to do the final interpretation using dialogue context.

Dialogue manager

The dialogue manager takes as input a set of recognised dialogue acts representing the meaning of the customer's last utterance. Output is a proposed dialogue act, representing the meaning of the next utterance of the system. It is thus responsible for controlling the system's dialogue behaviour. This includes interpreting recognised dialogue acts in context, looking up information in databases, performing problem solving and deciding how to present this information. An important task is response generation, deciding what to say next. The dialogue manager is the core of the Alparon system and will be explained in more detail in the next section.

Text generation

The text generator transforms proposed dialogue acts into grammatically correct sentences. A sophisticated text generator will use the linguistic atmosphere to adapt the system utterance to the customer's language.

Text to speech

The TTS module transforms written text into spoken words. Output will be presented to the telephone interface as a-law coded sound.

3.2.2 The coding scheme

A coding scheme was developed to code a corpus of 500 human-human dialogues recorded at OVR's call centres [Vark et al., 1997a]. This scheme was later adapted for use in an automatic coder [Rats, 1997a]. We intend to represent the meaning of both customer utterances and system utterances in this scheme. The design of the coding scheme is inspired by Conversational Games Theory [Carletta et al., 1995] and the coding scheme of the VERBMOBIL project [Alexandersson and Reithinger, 1995]. However, the Alparon coding scheme has a more hierarchical structure.

The coding scheme also uses the notion of speech acts [Austin, 1962], [Searle, 1969]. The title of the book where Austin introduced this theory says quite a lot: "How to do things with language". Speech acts theory views language use as performing actions, as opposed to earlier theories where language was seen mainly as referring to objects and states of the world, being "true" if this corresponded with the real world and "false" otherwise [Vagle et al., 1993]. There have been many classifications of speech acts – in the Alparon coding scheme we use "check", "clarification", question", "reconfirmation" and "statement". As basic building blocks we take speech acts with propositional contents of the utterance added (i.e. the literal information contents).

Codes in the Alparon coding scheme have a hierarchical structure consisting of moves, codes, sub codes and arguments [Vark, 1997c]. Moves corresponds to speech acts. Codes, sub codes and arguments will here colloquially be named "information codes". An overview of the codes used is given in figure 3.2. As an example, the utterance "The trains leaves at 11:30" will hence be coded as "Statement(Time(Departure(Exact(absolute(11:30)))))"%[3]. A set of abbreviations has been developed (see table 3.1). Using these, the example will be written as "Sta(Tm(DeT(Exact(absolute(11:30)))))". See [Rats, 1997a] for more details.

q

3.2.3 Design of the Dialogue Manager

Most of the parts outlined above are merely in the idea stage and need lots of work before they are implemented. (An exception is the Parser/Coder, which is implemented and running.) The Dialogue Manager has been design in detail and is now not far from being a demonstrable prototype. The Dialogue Manager is prioritised since we believe that an improvement here is a key to success for ASP systems.

The design of the Dialogue Manager is explained in [Vark, 1997a]. The top-level model is shown in figure 3.3. The dialogue manager is designed as a set of modules that interact through sending messages to each other. For scalability it should be possible to run different modules on different computers. The modules will be described here briefly. Emphasis will be on the modules I have had responsibility for, namely The Database Manager, the Travel Board and the Query Travel module.

[3]"The train" skipped for simplicity

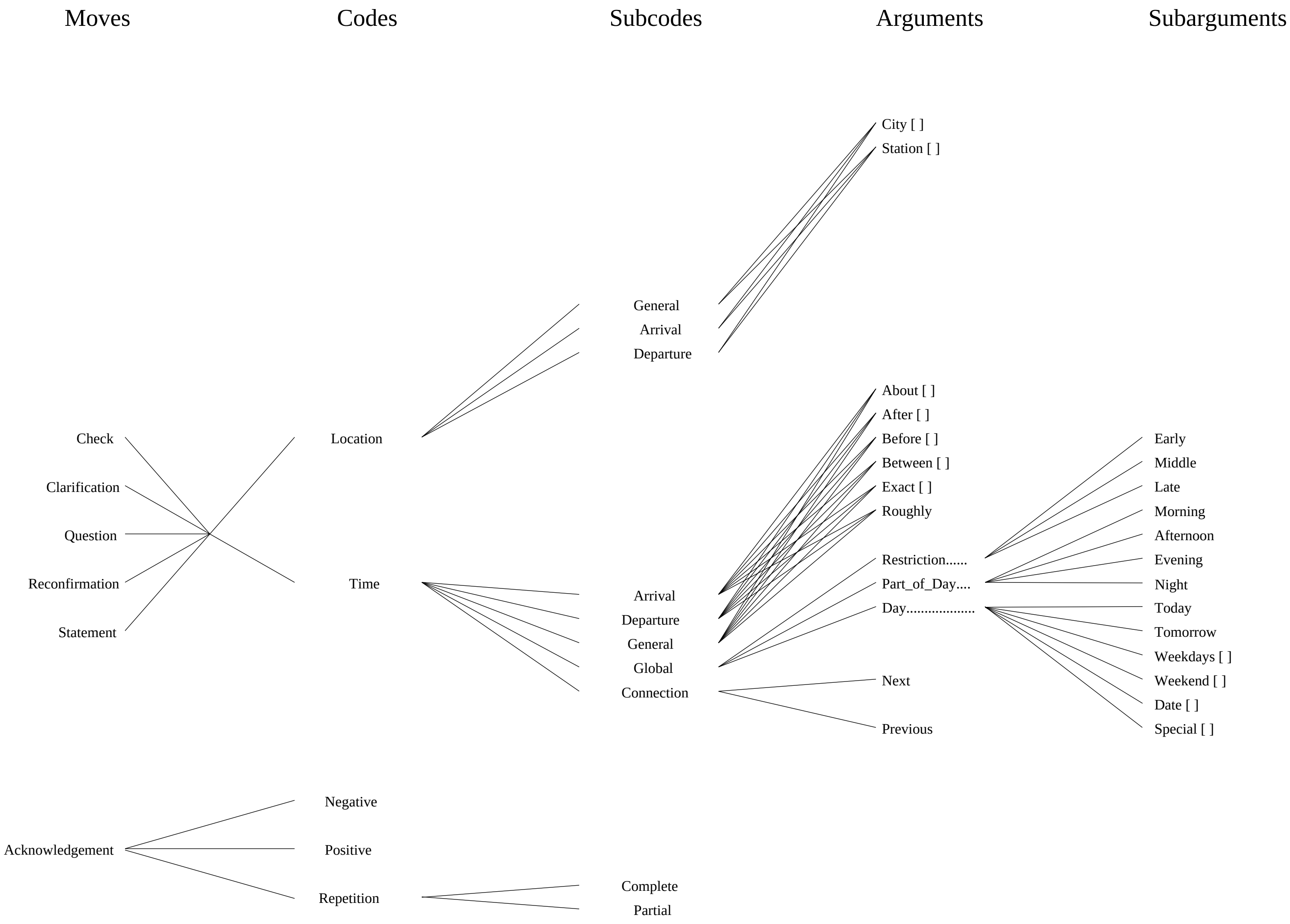

Figure 3.2: Overview of the coding scheme used in Alparon

Phase	ab.	Move	ab.	Code	ab.
Bye	B	Acknowledgement	Ack	Acknowledge	Ack
Greeting	G	Alignment	Ali	Announce	Ann
Information	I	Bye	Bye	Break	Bre
Pause	P	Check	Che	Bye	Bye
Query	Q	Clarification	Cla	Garbage	Gar
Subquery	S	Greeting	Gre	Location	Loc
		Other	Oth	Other	Oth
		Pause	Pau	Personal	Per
		Question	Que	Price	Pr
		Reconfirmation	Rec	Ready	Rea
		Statement	Sta	Route_Information	RI
				Time	Tm
				Timetable	Tt

Subcode	ab.	Subcode	ab.	Subcode	ab.
Amount	Amo	Global_Time	GlT	Repetition	Rep
Announce	Ann	Goodbye	Goo	Residence	Res
Arrival_Place	ArP	Irregular_Service	IrS	Return_Grace	ReG
Arrival_Time	ArT	Mumbling	Mum	Season_Ticket	SeT
Company	Com	Name	Nam	Stopover	Sto
Connection	Con	Negative	Neg	Telephone_Number	TeN
Connection_Type	CoT	Nonverbal	Non	Thanks	Tha
Daily_Schedule	DaS	Opening_Hours	OpH	Transport_Type	TrT
Departure_Place	DeP	Other_Information	OtI	Travelling_Time	TTm
Departure_Time	DeT	Positive	Pos	Verbal	Ver
Direction	Dir	Regular_Service	ReS	Via	Via
Foreign_Information	FoI	Regular_Ticket	ReT		

Table 3.1: Abbreviations used in the Alparon coding scheme

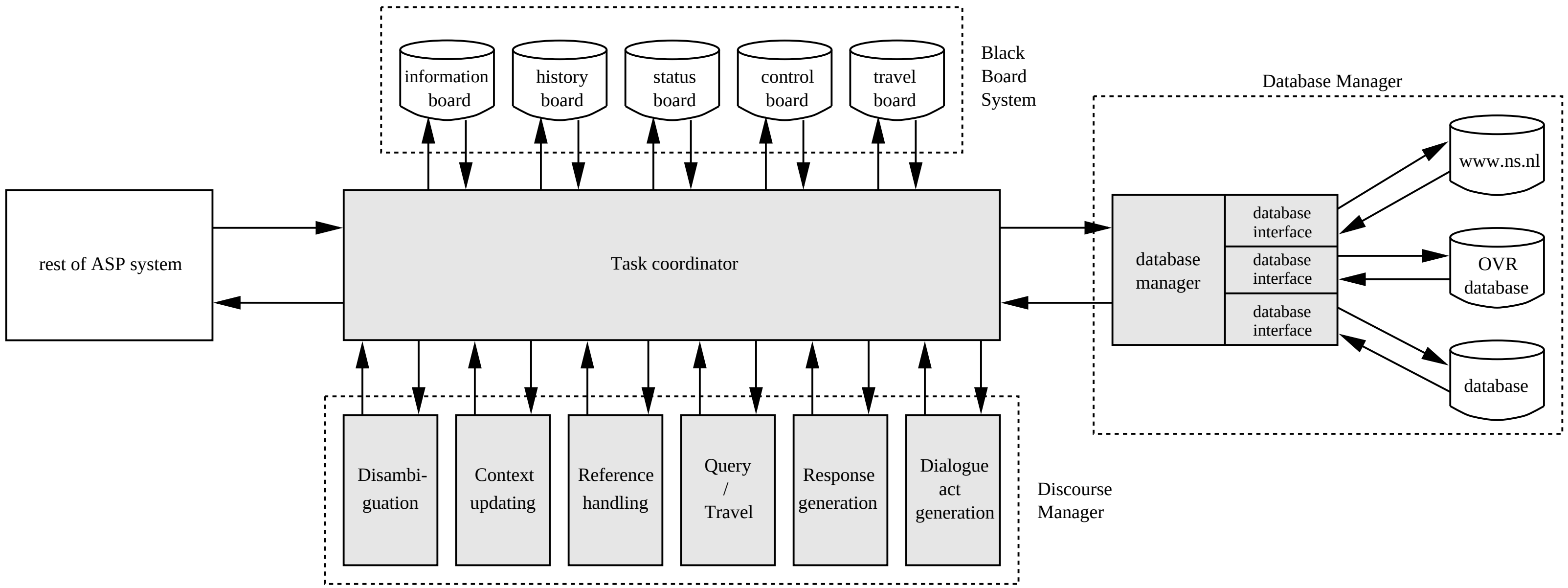

Figure 3.3: A model of the Alparon Dialogue Manager

Task Coordinator

The contribution of the Task Coordinator is to lets the other modules communicate with each other. This module does not do any dialogue processing in itself. Instead, it merely routes messages to and from modules. The Task OVRinator consists of many portmappers, one for each computer running any of the Dialogue Manager modules. When a module wants to send a message to another module, it sends the message to the portmapper on the sender's machine. This portmapper sends the message to the portmapper on the receivers machine, which sends the message to the receiver module. The modules refer to each other with logical names, and it is the task of the portmappers to keep track of which modules are running on which machines. The message format used is purely text-based, so any module capable of writing and reading text to network sockets can be Dialogue Manager modules. This ensures freedom to experiment with different implementation strategies and programming languages, which is considered important given the research nature of the project.

Discourse Manager

The Discourse Manager is active part of the Dialogue Manager. The Discourse Manager is in charge of the dialogue and uses the other modules for services. However, the Discourse Manager modules are not allowed to have any memory from one turn[4] to another. The Black Board System serves as memory. This restriction is introduced to make it easier to keep all information consistent by ensuring that it is only stored in one module.

The Dialogue Manager consists of a number of modules [Vark, 1997a]:

- **Disambiguation Module:** Chooses the most probable dialogue act from the set of recognised dialogue acts proposed by the coder. This is done on by taking dialogue context (i.e. what has been said earlier) into account. The Disambiguation Module can also change dialogue acts, as when dialogue history shows that a "statement" really is an "acknowledgement".

- **Reference Handling module:** Checks whether information contained in the customer's utterance is a reference to a already known piece of information or is new information. For example, "Rotterdam" may be resolved to "Rotterdam Central Station" if this has been mentioned earlier in the dialogue.

- **Context Updating:** Stores information in the Black Board System. This means storing information contained in customer utterances, but also the actual dialogue acts plus the alternative interpretations that were discarded by the Disambiguation Module (in case the decision turns out to be wrong). Also, if certain goals are satisfied, the control board has to be updated.

[4]A turn lasts from one party in the conversation (the customer or the system) starts talking to the other party takes over. A turn consists of one or more utterances.

- **Query Travel:** Phrases database queries using information given by the customer. Also chooses the fitting travel plan from those returned by the Database Manager. Stores travel plans (including which was chosen) on the Travel Board. Prepares chosen plan for presentation and stores presentation on the Travel Board. This module is part of information presentation and will be discussed further in the next chapter.

- **Response Generation:** Decides what next system response should be. Alternatives might be to ask a follow-up question or present a piece of information. Much of the intelligence of the system will be in this module. The generated response is stored on the Control Board.

- **Dialogue Act Generation:** Generates a dialogue act suited for giving the response stored on the Control Board. This dialogue act will then be sent to Text Generation to be transformed into a sequence of words.

- **Dialogue Act Prediction:** Predicts future customer dialogue acts. This can be done during idle time. The motivation is to be able to better choose the right interpretation of customer utterances.

3.2.4 Black Board System

The Black Board System is the memory of the Dialogue Manager. Black boards should be passive and merely serve the active Discourse manager modules, but in return they are allowed to have a memory than spans several turns.

All information of lasting value (i.e. of value in later turns) will be stored on blackboards. Blackboards are memory modules that can offer more or less intelligence in the form of organising and reasoning over their own contents. By storing information in separate storage modules it is easier to keep all information consistent while it still is available to all modules.

A module that wants to store some information that might be useful later in the dialogue sends the information in a message to the appropriate board. Analogously, to get information the module sends a request message and receives the information in a reply message.

New boards can easily be added as new needs arise. The current list of boards is:

- **Information Board:** Stores the pieces of information given in customer utterances. Examples are names of stations and time values. The values stored should also be commented, saying if the value has been confirmed, has changed etc.

- **History Board:** Stores dialogue acts uttered by the customer as well as those uttered by the system. This can be useful in handling later misunderstanding. Customer dialogue acts rejected by the Disambiguation Module should also be stored to allow disambiguation decisions to be redone if needed.

- **Status Board:** Stores the current state of the Dialogue Manager. This can include both the dialogue state (like what phase the dialogue is in) and state of the outside world (like if databases are online or not).

- **Control Board:** Stores goals that define the structure of the dialogue. Modules store goals on the Control Board to influence the progress of the dialogue. For instance, the Query Travel module might store a goal to present a travel plan when a new travel plan has been chosen. The Response Generation module uses these goals to decide the next system utterance. By changing the rules for managing goals on the Control Board and for putting them there we can change the structure of the dialogue. Experimenting should be done to find a solution that gets high customer acceptance.

- **Travel Board:** Stores travel plans (i.e. how to get from one place) to another as returned from the Database Manager. One of the plans will be marked as being the most suitable. For this plan, how to split the plan up in pieces when presenting it to the customer will also be stored. This module is part of information presentation and will be elaborate on in the next chapter.

3.2.5 Database Manager

The Database Manager is responsible for retrieving data from timetable databases. Since various databases might be used, possibly at the same time, the Database Manager has to serve as a common interface, translating between different syntax as needed. This module is also part of the information presentation and will be covered in the next chapter.

Chapter 4

Information Presentation

This chapter will give an overview of my work, namely the design and implementation of the Alparon Dialogue Manager. After reviewing ASP technology in general in chapter 2 and Alparon in chapter 3, I have finally got down to a level of detail where I will describe what I have actually done. I will start this chapter by trying to define exactly what my responsibility has been in developing the Alparon Dialogue Manager. I will then outline some theoretical foundations for my work, where the Alparon report on information presentation [Rats, 1997b] is central. Towards the end of the chapter I will give some general notes on the design and implementation I have done, before presenting some related research. For each module I have implemented there is a separate chapter that goes more in depth.

Each of the modules I have made has got a separate chapter, so details on design and implementation will be given there.

4.1 Phases of the Dialogue

The dialogue between a customer and the Alparon system can be divided into the following Phases [Vark, 1997a]:

1. **Greeting phase:** Mutual exchange of opening phrases
2. **Query phase:** The customer gives information needed to state a query
3. **Problem solving phase:** The system executes a query and chooses a travel plan
4. **Information phase:** The system presents the chosen travel plan to the customer
5. **Closing phase:** Mutual exchange of closing phrases

Phase 2, 3 and 4 can be repeated until the customer is satisfied with the information presented and does not have more queries. My work has been to design and implement phase 3 and 4: problem solving and presentation. Note that no utterances are made in the problem solving phase. The actual execution of a presentation will be done by the Response Generation Module and is hence outside of the scope of my work. Since I have already presented the model for the Dialogue Manager and outlined the tasks of each module, it might be more precise to define the scope of my work as the module I have been assigned responsibility for:

- Database Manager
- Travel Board
- Query Travel module

I have here assumed that the information asked for by the customer is a travel plan, i.e. a description of how to go from one place to another. Although this certainly is a common type of question for this kind of service, other questions like ticket prices, opening hours of ticket offices, changes in schedules because of construction work etc should also be handled. However, to keep things simpler I will continue to refer to the presentation to be presented as a travel plan.

The tasks of the problem solving and presentation phases can be summarised as formulating a query, performing the query, choosing one of the travel plans and preparing it for presentation. The execution of the query is the job of the Database manager. Formulating it, choosing one of the plans and preparing the presentation is the job of the Query Travel module. Given the restriction that discourse modules should not have any memory stretching over several turns, the Travel Board is needed to store the travel plans and presentations. This also separates data from discourse modules, making it easier to adapt the dialogue manager to other domains.

The Travel Board and the Database Manager are fairly trivial and not very interesting. Giving them a chapter each in this report has been done more for completeness than for anything else. I have emphasised the Query Travel module and especially travel plan presentation.

4.2 Theoretical foundations

Our system should mimic human strategies in information strategies. I will here outline a general theory of how humans present information in dialogues before the corpus study of Rats is summarised.

4.2.1 The Conversational Maxims of Grice

We want to design a system that feels as natural as possible to converse with for the customers. Then it is important that the system follows rules that humans take for granted in conversation. Grice has formulated a set of conversational maxims which reads, somewhat simplified: [Grice, 1975] (quoted in [Allen, 1997])

- Quality: truthful and accurate information
- Quantity: neither more nor less information than is required
- Relation: information appropriate to the task
- Manner: clear and unambiguous information

Grice claims that rational, co-operative humans follow these guidelines and expect others to follow these in conversations. Assuming that the customers are rational and co-operative this would imply that we should try to make Alparon follow these rules. This will have some consequences for information presentation. The quality and relation maxims are least problematic since this is simply a matter of having a correct database. The system will not start talking about anything else than database results. The quantity maxims is far more tricky, since what how much information is required depends on the specific user. The manner maxim also requires care.

4.2.2 Conclusion From the Corpus Study on Information Presentation

As part of the Alparon project, Mieke Rats performed a study of information presentation in VIOS and in the human-human corpus [Rats, 1997b], [Rats et al., 1997]. She also gives recommendations for how information presentation should be done in the Alparon system. Since one of the purposes of my work was to put some of this ideas into practice, her work will be presented in relative detail.

Rats built her analysis on a study of 100 human-human OVR dialogues. This is a subset of the 500 dialogues that was coded to form the coded corpus for Alparon [Vark et al., 1997b]. She first quotes results from appreciation studies of VIOS, the ASP system OVR use today for train table information. She then analyses the human-human dialogues, pinpointing differences from the VIOS presentation. Recommendations for how VIOS can be improved is then given. This will be summarised here.

Information Presentation in VIOS

Appreciation studies of VIOS have shown that the information presentation is far from perfect [Blasband, 1997], [Manintveld and Rothkrantz, 1997], [Peters and Kock, 1997]. Rats lists a number of problems:

1. All information is given at once. This overwhelms the user.
2. The use of full names for stations is very unnatural and confusing. For example, if the user mentions the station "Schiedam", the system will refer to this with its official name "Schiedam Rotterdam West".
3. The customer is not allowed to interrupt if she does not comprehend the given information.

4. The speed of the system's speech is not appropriate for all users. It is too fast for those who wants to write down the information, and too slow for those who only wants the most important information.

5. The speech is unnatural. The tempo and loudness is uneven and a the intonation is not natural.

Problem 4 and 5 will have to be handled by speech generation and are hence outside of the scope of my work. Problem 2 have been addressed in the Alparon system by introducing a separate module for reference handling. As discussed later, my work will address problem 1, whereas future extensions will address problem 3.

Information Presentation in the Human-Human Dialogues

The study of how the human operators present information showed that their presentations were more interactive:

- Travel plan were presented in a number of steps.
- The customer was expected to give an acknowledgement after each step.

The steps generally followed the temporal order of the travel plan. 87% of the operator turns contained only one utterances and 10% contained two utterances. Rats divides information into given information and new information. Given information is assumed to be already known by the customer, either because it was part of her query or because it has been presented to her in a previous utterance. New information is information not assumed to be given. Rats found that the operators tend to present at least one piece of new information in each turn.

When trying to extract rules from the operator's behaviour, Rats groups the 100 dialogues into six situations. A situation is defined by the kind of information given in the query and the kind of information returned from the database. See figure 4.1 for the list of situations used. For each situation, Rats classifies operator's presentations into a set of different scenarios. A scenario specifies what utterances should be used and what information should be included in each utterance. Rats then collects statistics on how often each scenario is chosen by the operators. The most common scenarios are used in defining templates for each situation. A template specifies how an ASP system should present information in that situation to be conform with human practices.

The data material for some of the situations are weak. Rats does not specify how many dialogues fall into each situation or follow each scenarios. However, it is possible to reconstruct these numbers by analysing the relative percentages each scenario is used for a situation[1]. My estimates are given in table 4.1.

[1]For example, for situation 6 there are 5 scenarios listed, each used in 20% of the cases. Hence the number of times this situation has appeared in the corpus is a multiple of 5.

Figure 4.1: Situations used by Rats when classyfying dialogues

1. The query phase has brought up a departure place, an arrival place, and a global indication of the departure or arrival time, and the data base query yields one direct train connection.

2. The query phase has brought up a departure place, an arrival place, and a departure time, and the data base query yields one direct train connection.

3. The query phase has brought up a departure place, an arrival place, and a global indication of the departure or arrival time, and the data base query yields one train connection with one change.

4. The query phase has brought up a departure place, an arrival place, and a global indication of the departure or arrival time, and the data base query yields one train connection with two changes.

5. The query phase has brought up a departure place, an arrival place, and a global indication of the departure or arrival time, or a time span within which the travelling has to take place, and the data base query yields two or more direct train connection.

6. The query phase has brought up a departure place, an arrival place, and a global or exact indication of the departure or arrival time, and the data base query yields two train connections of which at least one of them involves a change.

Table 4.1: Estimates og how many times the each situation occured in the corpus of 100 coded dialogues

Situation	Estimated number of cases	Number of scenarios
1	43	7
2	2	1
3	28	4
4	3	3
5	19	6
6	5	5

From these number we see that situation 2, 4 and 6 appeared very rarely and were difficult to classify in scenarios. Only situation 1, 3 and 5 seems to have a sound enough statistical foundation to be used.

Suggestions for improvement of ASP Information Presentation

Rats suggests improving the VIOS information presentation in three stages:

1. Formulate templates for each situation that presents travel plans stepwise, distinguishing between new and given information. Pay attention to what terms the customer uses when referencing station names etc.

2. Formulate shorter templates that only gives the most important information. Allow the caller to ask for elaboration or ask questions at the end of the presentation.

3. Allow the caller to interrupt the presentation.

I have taken up some of these suggestions in my work. A mechanism for using templates for presentations have made, so it is possible to use the templates Rats suggest. Stage 1 can then be reached by implementing the templates and the Reference Solving module. Stage 2 may be reached by implementing the shorter templates and implementing a Response Generation module that handles follow-up question.

4.2.3 Following rules vs. involving rules

The corpus-based approach to making an ASP dialogue system is to study how human use language and extract rules describing this behaviour. A computer can be programmed to follow such rules, but does that give it human dialogue skills? This point has been raised by Button et al, who are harsh in their critique of attempts to make such ASP systems with natural dialogue capabilities: "[...] we consider the development of a conversing computer to be another overweening ambition within computational linguistics that compounds a theoretical confusion over what it is to engage in dialogue or conversation, [...]" [Button et al., 1995] (p. 184). The main point in Button et al.'s critique seems to be the difference in how the rules captured from corpora relate to the behaviour of humans and machines. The distinction is the one Wittgenstein makes between "a process being in accordance with a rule" and "a process involving a rule" [Wittgenstein, 1958]. This distinction is explained by Bogen: "An activity *accords* with a rule if it exhibits regularities expressed by the rule. But it *involves* a rule only if agents actually use the rule to guide and assess their action" [Bogen, 1972]. The rules we find from corpus analysis will be *followed* by a computer, but they only *accord* with human behaviour. The computer follow the rules, humans only act in a way that can be described with them. Even though the end result ideally is the same (if we manage to capture all behaviour in rules and follow these rules perfectly), the process leading to this behaviour is completely different.

As long as our goal is to make a better working system, this point does not harm our case. However, we must be careful with claiming that anything about human language use can be learnt from how our system treats language. For example, it will be explained later how the Query Travel module plans a presentation by searching for a matching template that specifies how the information should be presented. Even though our system is based on human language use, we do not claim that humans uses templates like this.

4.3 Implementation

There are many dependencies between modules of the Dialogue Manager. Priority in implementing the modules has been on having basic versions of all modules ready quick. This is necessary to make it possible to test modules that depends on other modules. The approach taken is to do a top-down design of a first version of any module, but making sure that the design is scalable. This prototype will later be refined through iterations, producing increasingly sophisticated modules. The downside of this strategy is that modules of low priority might get stuck in an early version. This is the case for the Database Manager. For the Travel Board, a first prototype could be made fairly quickly by copying most of an early version of the Information Board. Later iterations have cleaned up the code, making it more maintainable. With the Query Travel module, it was more work to get the first version ready since I decided to implement it in Java. The existing parts of the Dialogue Manager was implemented in Perl, and choosing another language did lead to a lot of extra work. But I tried to make generic solutions, so later modules can gain quite a lot by reusing the Java code.

4.4 Related work

Although there is has been a lot of ASP research projects, little work seems to have been done specifically on information presentation. There is some work done, though – for example, Qu describes a system for giving information about hotels [Qu, 1997]. Experiments were done to find the ideal amount of information to include per utterance. Qu found that 10 content words per utterance was a good estimate. However, this study did not separate new and given information, and the system had no understanding of the meaning of its utterances. Still the use of usability testing from the field of human-computer interaction to decide information presentation scheme is an interesting approach.

Cawsey has done some interesting work on generation interactive explanations [Cawsey, 1991]. The domain she uses is the explanation of electrical circuits (and later medical records [Binsted et al., 1995]), so the explanations are a fair bit more complicated than our travel plan. An interesting part of Cawsey's work is her handling of interruptions and follow-up questions, re-planning the explanation dynamically. She achieves this by maintaining a goal stack and interleaving planning with execution.

There has been many projects on ASP system for offering travel information by telephone. VIOS has already been mentioned. This is a system now put into limited commercial use, offering information about Dutch trains. VIOS is based on a research system called OVIS. This was a translated of a similar German prototype [Aust et al., 1994]. OVIS2 is an improved prototype based on OVIS [Veldhuijzen van Zanten, 1996]. All these projects fall under the ARISE project, which is part of the European Union's research programme. Participant in ARISE is also Philips, who has developed a commercial product for developing ASP system [SpeechMania, 1998]. This product, named SpeechMania, allows application programmers to program the dialogue using a custom language called HDDL (High-level Dialogue Definition Language). This greatly reduces the work needed to make an operative system. In Norway, work is done by NTNU, Sinter and Telenor to make a system offering information about busses [SPODIS, 1997]. There is an extra difficulty involved in speech technology for Norwegian language since there are no general speech recogniser available.

Chapter 5

Database Manager

This chapter describes the design and implementation of the Database Manager. It is closely based on a technical report [Oldervoll and Vark, 1997a] that was written during the design phase and served as requirements and design specification for the implementation of the module. This chapter will first define closer the requirements for the Database Manager. Next, some design decisions will be discussed, in particular which syntax to use for queries and results. Towards the end a fairly detailed description of the messages the Database manager accepts and sends will be given.

5.1 Problem Definition

The task of the Alparon system is to offer time schedule information for public transport through a speech interface. The information will be read from databases containing the time schedule. The databases might be offered by the transport companies, and therefore we cannot rely on the database format and the interface to be fixed.

By using a separate Database Manager, we achieve a flexible access to the data. This also makes the system more extensible by making it easy to add more databases. The main task of this module will be to provide information from external databases by acting as an interface between the Discourse Manager and the databases. The Database Manager provides the opportunity to use a common query language no matter what underlying database one wants information from, thereby hiding the idiosyncrasies of the different database management systems. It will also be able to merge information from different databases in a transparent way. In short, the Database Manager should give the Discourse Manager the impression that one custom-made database is available whereas the actual databases present may vary. This section elaborates upon these tasks.

5.1.1 One interface to heterogeneous databases

We want to have a fixed way of querying the underlying databases, independent of which databases are actually present. To achieve this, a common query language will have to be defined. For each underlying database, the Database Manager will have to translate from the general query language to whatever query language the database uses.

5.1.2 Support multiple databases

More than one database might be needed if the system should handle different modes of transport or services from more than one company. If multiple databases are present, this should be transparent for the Discourse Manager. Given a query, the Database Manager should decide which database to query. It might be necessary to query more than one database, and the Database Manager should then merge the output from the queries into one query reply to the Discourse Manager.

5.1.3 Perform domain dependent data processing

It might be desirable that the Database Manager offers more than just raw data but also results of reasoning over this data. If the reasoning is simple, like summing ticket prices or distances or giving the shortest travel time between to stations, we can expect a general RDBMS (Relational Database Management System) to be able to do the reasoning. If the reasoning incorporates domain knowledge, this functionality will have to be a part of the Database Manager. We will also have to rely on the Database Manager to do the reasoning if data from more than one database is involved.

5.2 Design

This section outlines the design choices taken and implementation strategy chosen for the Database Manager. The approach taken is to start with a very basic design and extend later as needed. The implementation will therefore consist of increasingly sophisticated prototypes. Requirements for a first prototype is outlined. An important decision is which format to use for queries and results. This is discussed in the final section.

5.2.1 One Coordinator, several Interfaces

The Database Manager will be Alparon's bridge to the several, heterogeneous databases. We will split the functionality into a general part common for all databases, and several database independent parts, one for each database. The Database Manager will hence consist of one Database Coordinator and several Database Interfaces. This was shown in figure 3.3

In:	Departure place Arrival place Departure time *or* arrival time
Out:	Departure time and arrival time for complete travel Departure time and arrival time for train changes, if any Optional: Any other information about the travel plan

Table 5.1: Specification of queries supported in the first prototype

Database Coordinator

The Database Coordinator will be responsible for coordinating all databases. This includes deciding which database(s) to query and merging output from several databases. The coordinator is also responsible for doing high-level reasoning on data that the underlying database managers cannot be expected to do.

Database Interfaces

The Database Interfaces will be responsible for querying the underlying databases, translating queries from an agreed-upon Alparon query format to a query format the underlying database can understand. The results must be translated back to the standard Alparon result format in a similar way. Each underlying database has its own Database Interface to facilitate communication between the database and the coordinator.

5.2.2 Prototyping

The implementation of the Database Manager will be done through prototyping, where new prototypes will be made as new databases become available and demands for new functionality arise.

First prototype

First priority is to get a simple prototype up and running so other modules can be tested with real database queries.

Database Interface: At present, there is only one database readily available. This is an electronic timetable service available on the Internet from the Dutch railway company[1]. Data from this database is retrieved by sending input to a web page form and parsing the web page with the results. This limits the type of queries that are possible to fully specified queries, as illustrated in table 5.1.

We refer to the description of how to get from one place to another as a travel plan. A travel plan minimally contains a place to go from, a place to go to, departure time and arrival time. If a direct connection is not available, it might

[1] Nederlandse Spoorwegen's "Reisplanner" at http://www.ns.nl/reisinfo/

also contain a list of stations where one have to change, with corresponding arrival and departure times. Any other information that is available about the travel might be included. For a train database this might be train name and type, platform number, direction of the train etc.

Using a public web service is clearly an inefficient way of accessing data, but the speed is sufficient at this stage of the development. Later a local database should be used to increase speed.

With only one database available, only one Database Interface is needed. The job of this interface will be:

- Receive a query in Alparon format
- Send the corresponding query to the query web page
- Receive a web page with the results
- Parse the page, extracting the relevant information
- Translate information into Alparon query format[2] and return it

Database Coordinator: The main tasks of a database coordinator is to coordinate databases and perform high-level reasoning on data. With only one database, there is no need for any coordination between databases. At present, no high-level data reasoning is needed, either, since we are just interested in basic queries at this stage. The Database Coordinator thus deteriorates into nothing, so no coordinator is needed in the first prototype. Database Interfaces accepts queries in the same format as the overall Database Manager module, so at this stage all queries to the Database Manager can be sent directly to the Database Interface for the web database.

Future improvements

Later prototypes will certainly add more Database Interfaces as new databases become available. A coordinator will then have to be implemented. Future prototypes may also offer services beyond basic queries:

Check station names

During idle time the Database Manager can check whether the names of places the customer mentions are in the time-schedule databases. At present this functionality is not needed since the parser will only recognise the names that are in the database.

[2] The query format is explained later in this section

Not fully specified travel schemes

The Database Manager may also answer queries about travel schemes that are not fully specified, by instance when no time is indicated. The customer may want to know whether there are trains running between two stations, the fare, distance or duration of a certain travel, etc. The Database Manager can find this information by trying a typical departure time or a couple of different departure times.

Calculate frequencies

The Database Manager may also provide information that requires the coordinator to do some computations on the information returned from the external databases. For example, an answer to a question like "how often do trains from Delft to Den Haag leave in the evening?" may be given by querying the appropriate database and performing some statistical calculations on the results to come up with an answer like "Before midnight there is about four trains an hour, and after midnight there is one train an hour".

5.2.3 Format for queries and results

Isolating the idiosyncrasies of the underlying databases from discourse modules gives us freedom to define any format we want for queries and results.

A query can be considered a request for information satisfying certain constraints. The result of the query will then be the information itself. Both constraints and information can be coded on the same format. We will here use the term information format to denote this format.

Criteria

In choosing a format, we will strive to maximise these qualities:

- **Expressiveness:** The ability to express advanced queries and results. We must assure that any query and result needed will be possible to formulate using the format.

- **Parsability:** How easy the format can be read by the computer system. In this criteria, we also include how easy it is to synthesise a query or a result, i.e. how easy it is for the system to write the format. We don't want to introduce too much overhead in coding and decoding information to and from our format.

- **Readability:** How easy it is for a human to read the format and understand the information contents. A readable format will be of great help for the developers of the system.

Choosing a format

We will now choose a suitable information format using the guidelines given above.

Expressiveness

As explained in section 3.2.2, a coding scheme has already been developed for coding dialogue acts in the Alparon system. Information on the query/result format will be translated from or into information codes in the coding scheme. Information used in queries originates from user utterances, which will be coded as moves with information codes. Some of the information from results will be used in system utterances, also coded as moves with information codes.

We therefore have a goal of expressiveness for the format we are about to develop: The information format should be expressive enough to express everything that can be expressed using the information codes of the coding scheme. Any expressiveness beyond this level is wasted since it will be lost in the translation to Alparon codes anyway.

Parsability

We want the translation between the information format and information codes to be as simple as possible. The simplest solution is to use the syntax from information codes also for the queries and results. Hence, if a travel plan contains a departure from Rotterdam CS, we can code this as Loc(DeP(Station(Rotterdam CS))). Translation from information format to dialogue act can now be done simply by adding a move and use the information code as given.

To maximise parsability, we therefore prefer to use the same syntax for each piece of information as used in the information codes of the coding scheme. With this choice taken, what is left is to specify how the individual codes should be put together into a list of travel plans or a query. From a parsability perspective, the simplest is to separate codes with a special character that is not used in the codes themselves.

Readability

The information codes in the coding schemes might look a bit cryptic. But readability is mainly sought to aid an Alparon developer. Since these codes are used throughout the project, the developer can be assumed to have a good knowledge of the codes and hence will read them easily.

The result of a query can be expected to contain much more information than the query. Therefore, we will here consider the case of how to make a query result consisting of a list of travel plans readable.

A travel plan is a description of how to get from one place to another. This might involve several sub travel plans, descriptions of direct connections between stations (i.e. legs of the journey). Each sub travel plan contains many information codes: the names of both stations, departure and arrival times, and usually other information like platform number, train name etc.

To maximise readability, this information should be presented in the following manner:

- With a reasonable line length, so messages can be read on paper or on the screen without adding extra line break
- Without cluttering the message with too many parenthesises, commas etc.

A travel plan is obviously too complex to fit on one line. Even a sub travel plan will be too long. Therefore, we choose to keep each information code on a separate line.

Grouping of information codes will be done by inserting blank lines between groups since this will be intuitive to read. Each sub travel plan will therefore be separated by a blank line, and each travel plan will be separated by two blank lines.

Decision

The Database Manager will use the following format for accepting queries:

- Each piece of information will be coded as an information code in the Alparon coding scheme.
- Each information code will be written on a separate line.

The results sent from the Database Manager will use this format:

- Each piece of information will be coded as an information code in the Alparon coding scheme.
- Each information code will be written on a separate line.
- One blank line separates sub travel plans.
- Two blank lines separates travel plans.

5.3 Interface

The Database Manager's job is to answer queries about travel schedules. It does this by forwarding the query to the underlying databases, translating the query and the results between the format used in Alparon and the format the specific databases use.

At present, the only message it accepts is the QueryCompleteSchedule message. This will result in a CompleteSchedule message with the resulting travel plans. Travel plans are written on this format:

```
subtravel1
subtravel2
subtravelN
```

where each sub travel has the form

```
Loc(DeP(Station(Place1)))
Tm(DeT(Time1))
Loc(DeP(Station(Place2)))
Tm(ArT(Time2))
...
```

The actual information codes will of course depend on the information contained in the sub travel. Any information code written in the Alparon coding scheme is allowed. Each sub travel is separated by a blank line. Each travel plan is separated by two blank lines.

A query will have the syntax as an under-specified travel plan - for example only departure place and arrival place plus an approximate departure time.

The database manager will have to query databases to fill in the missing details like accurate times, stations where the customer have to change trains, train names etc.

5.3.1 Making queries

Message name

```
QueryCompleteSchedule
```

Message arguments

```
N
Loc(DeP(Station(Place1)))
Loc(DeP(Station(Place2)))
Tm(DeT(Time)) {\it or} Tm(ArT(Time))
```

N is here the number of plans wanted in a reply. If no number is given, a default number will be used. At present, the default number is 6.

Other message arguments may be supported later to provide more functionality.

Answer Message name

```
CompleteSchedule
```

Answer Message arguments

```
travelplan1
travelplan2
....
travelplan6
```

A query is posed by sending a question dialogue act listing constraints to the travel plans. All locations are taken as absolute constraints (every travel plan return should include exactly the same places), whereas all time values will be considered as approximate values. The database manager will return the six travel plans that come closest to the times given in the query. Less than six plans (even zero) may be returned if more matching plans cannot be found.

Example: Trains from Delft to Rotterdam CS at around five o'clock 27 September can be queried for by sending a "QueryCompleteSchedule" message with the following argument to the database manager:

```
Loc(DeP(ns(delft)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(DeT(exact(absolute(17:00))))
Tm(GIT(Day(Date(27,September))))
```

The database manager will answer by giving six travel plans for how to go from Delft to Rotterdam CS. The six plans with departure time closest to five o'clock are chosen. The answer is given as a "CompleteSchedule" message with this argument:

```
Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(16:36))))
Loc(DeP(Platform(2)))
RI(TrT(Train(sneltrein5463)))
Loc(Dir(Station(Rotterdam CS)))
Loc(Dir(Station(Breda)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(16:46))))


Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(16:42))))
Loc(DeP(Platform(2)))
RI(TrT(Train(stoptrein 5063)))
Loc(Dir(Station(Rotterdam CS)))
Loc(Dir(Station(Rotterdam Lombardijen)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(16:56))))


Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(16:54))))
Loc(DeP(Platform(2)))
RI(TrT(Train(stoptrein 5163)))
Loc(Dir(Station(Rotterdam CS)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(17:08))))
```

```
Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(17:06))))
Loc(DeP(Platform(2)))
RI(TrT(Train(sneltrein 5465)))
Loc(Dir(Station(Rotterdam CS)))
Loc(Dir(Station(Dordrecht)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(17:17))))


Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(17:12))))
Loc(DeP(Platform(2)))
RI(TrT(Train(stoptrein 5065)))
Loc(Dir(Station(Rotterdam CS)))
Loc(Dir(Station(Rotterdam Lombardijen)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(17:26))))


Loc(DeP(ns(delft)))
Tm(DeT(exact(absolute(17:26))))
Loc(DeP(Platform(3)))
RI(TrT(Train(stoptrein 5165)))
Loc(Dir(Station(Rotterdam CS)))
Loc(ArP(ns(rotterdam\_cs)))
Tm(ArT(exact(absolute(17:40))))
```

5.4 Implementation

Currently a script for grabbing data from the web train table service is used as Database Manager. This script is written by Guillaume Lecourt. I have added code for translating between the information code syntax and the syntax this script uses.

Chapter 6

Travel Board

This chapter describes the design and implementation of the Travel Board. Also this chapter is based on a technical report that was written during the design phase and served as requirements and design specification for the implementation of the module [Oldervoll and Vark, 1997c]. I will first define the task of the travel board. Then some design issues will be discussed. At the end, I will give a detailed description of the messages the Travel board accepts and sends.

6.1 Problem Definition

The Travel Board is Alparon's blackboard for storing travel plans and presentations of travel plans. Travel plans were explained in section 5.2.2. In short, they are descriptions of how to get from one place to another using the public transport in question. A travel plan consists of one or more sub-travels. A sub-travel is a connection between two stations, with station names and departure and arrival times indicated. Additional information like platform number, transport type etc. may also be included in the travel plans.

A presentation of a travel plan is a description of how this plan should be presented to the customer. A presentation consists of one or more presentation chunks. A presentation chunk is a set of information pieces like station names and departure times. The information contained in one presentation chunk should be suitable to be presented in one turn of the system. All the presentation chunks uttered in sequence should present the most important information from the complete travel plan. Information included in the presentation will hence be a subset of the information in the complete travel plan – only the information considered essential for the customer will be included.

6.1.1 How the Travel Board will be used

To give an impression of which role the Travel Board plays in the Alparon system, a simplified description of the module interaction in a dialogue will be given here. Tasks like disambiguation and reference handling are skipped for simplicity.

When a customer speaks to the system, speech is first translated to text by the speech recogniser. The parser/coder translates this text into codes in the Alparon coding scheme. The context updating module sends new pieces of information from the customers utterance to the information board. The response module might decide to pose a follow-up question, leading to a new customer utterance. When the response decides there is enough information on the Information Board to give an answer to the customer's question, it sends a query to the Database Manager. This module answers with the set of travel plans that fits the query best. These travel plans must be stored for later use. They therefore qualify as being information of "lasting value" and should be stored on a blackboard. Hence, the Query/Travel Module passes them on to the Travel Board.

The Query/Travel Module also chooses one of the travel plans as the best and starts presenting this plan to the user. Before presenting it, the Query/Travel Module splits it up in suitable pieces not to overwhelm the customer with too much information in one utterance. These "presentation chunks" are also stored on the Travel Board. For each new utterance of the presentation, the Query/Travel Module asks the Travel Board for the next chunk of the presentation.

If the user is not satisfied with this travel plan, a new plan must be chosen. The Query/Travel Module retrieves the plans from the Travel Board and chooses a new one. If the Travel Board is running out of travel plans, the Database Manager must be queried again and the Travel Board has to be refilled.

The role of the Travel Board is therefore to act as the system's memory for how departures and arrivals make up travel connections and how these connections should be presented to the customer.

6.1.2 Tasks of the Travel Board

Now that the role of the Travel Board is explained and the information it is supposed to store is defined roughly, it is time to get down to the practical tasks the board must do. Basically, it has to store and retrieve travel plans and presentations. The actual set of messages it will understand will be adapted to the needs of the other modules.

Storing travel plans

One of the main tasks of the travel board will be to store travel plans. These plans originate from the time schedule databases. As mentioned above, it is the job of the Query/Travel Module to initiate database queries and send the results to the Travel Board.

The Query/Travel Module will also decide which of the plans fits the customers needs best. The travel board must be able to mark this plan as "current" travel plan, and to change current travel plan if the Query/Travel Module decides so.

Retrieving travel plans

The travel board must be able to retrieve the current travel plan, for example to prepare it for presentation to the user. The possibility to retrieve all travel plans on the board should also be offered since the Query/Travel Module will need this to choose a "current" travel plan.

Storing and retrieving presentation chunks

When a travel plan is to be presented to the customer, it will be chunked into "presentation chunks", i.e. chunks of information suitable for presentation in one turn. The Query/Travel Module will be responsible for the chunking, but the travel board must store and retrieve the presentations.

The presentation of a travel plan can be seen as a list of presentation chunks where each presentation chunk is a set of information like station names and departure times. Information included in the presentation will hence be a subset of the information in the complete travel plan - only the information considered essential for the customer will be included.

6.1.3 Outside the scope of the Travel Board

A blackboard is mainly a passive module (except for manipulations done to its own data). Therefore, the heaviest processing of travel plans and presentations will be done by other modules, usually the Query/Travel Module. These are them most important tasks related to the Travel Board:

- **Querying for travel plans.** This is done by the Query/Travel Module as it includes interaction with both the Information Board and the Database Manager. The Database Manager will return a set of travel plans with departure or arrival time centred around the time values in the query. If more travel plans are needed (typically in the process of choosing a new plan), the Query/Travel Module will send another query to refill the Travel Board.

- **Choosing a current travel plan.** The Query/Travel Module is also responsible for choosing which of the travel plans fits the customers wishes the best since this requires interaction with the Information board. The Travel Board marks this plan, but the decision of which to choose lies with the Query/Travel Module.

- **Chunking the current travel plan for presentation.** The Travel Board is responsible for holding the current travel plan as well as the presentation of this plan. However, the task of deciding calculating the presentation chunks is left to the Query/Travel Module.

6.2 Design

This section discusses some of the design choices made and strategies taken in the implementation of the Travel Board.

6.2.1 Intelligence vs. overhead

The Travel Board is supposed to be "self-organising" and do necessary processing on its own contents. Overdoing this will lead to a high overhead since the board needs to do a lot of work just to store a piece of information. The board should therefore not be more intelligent than what is needed. The strategy chosen is to start with a straightforward implementation.

6.2.2 To parse or not to parse

If the Travel Board is going to do any reasoning on the contents of travel plans or presentations, it will have to parse them. By parsing a travel plan, we mean splitting it up to its individual information codes. This is necessary to be able to reason on the information contents of the plan.

For instance, the Travel board might offer a RetrieveNextTravelPlan message to retrieve the next travel plan if the customer found the plan the system suggested to be too early. To do this, the Travel Board needs to read the time values from the travel plans and compare them.

Parsing has a significant overhead. It also makes the internal data structure much more complex. Unparsed, a travel plan can be considered a text string. Parsed, it must be considered a collection of dialogue acts, where each dialogue act has a name and some parameters. This will make the Travel Board much more complex. We therefore prefer to avoid parsing the travel plans unless a real need for it arises. At present, parsing is not done.

As for the RetrieveNextTravelPlan example, this functionality is now performed by the Query/Travel Module. For this task one needs to know if one should find next departure time or next arrival time. There might be other constraints that affects the choice as well, as restricting the choice to a special train type or a special route. We have therefore decided that the Query/Travel Module is best suited for this since this module performs similar reasoning when deciding which plan to choose as current travel plan.

6.2.3 Adding a time dimension

One of the uses of the History Board is to make it possible to re-examine an earlier state of the dialogue in case of a misunderstanding or when an earlier decision proves to be a mistake in light of new information. To support this, all information on the Travel Board will be marked by a turn number saying at which dialogue turn it was added. It is then possible to retrieve the state of the Travel Board at a specified turn in the past.

If a travel plan given in a StoreTravelPlans message is already on the board, this second instance will simply be ignored. The turn number indicated for this travel plan will be the turn number of the first time it was stored. This situation will often arise when the customer repeatedly asks for a later (or earlier) connection after being presented with a travel plan. The Travel Board will run out of plans and will need to be refilled with a database query with a lightly changed time value. Some of the returned plans will be the same as the plans already on the board and therefore discarded.

6.2.4 Message set

The strategy chosen when deciding which messages the Travel Board should support, is to start with a basic set of messages and add new messages as the need arises.

Here is the list of messages the Travel Board understands. Typical uses of the messages is indicated.

- **StoreTravelPlans:** Sent by the Query/Travel Module to store new travel plans after a query to the databases has been done. Note that this message can be used to store more than one travel plan at a time. This is convenient since the Database Manager usually returns multiple plan and all of them should be stored on the Travel Board.
- **SetCurrentPlan:** Sent by the Query/Travel Module to mark a travel plan as the one who looks best suited to the customers wishes.
- **RetrieveCurrentPlan:** Sent by the Query/Travel Module to get the plan previously marked as current to chunk it up in presentation chunks.
- **RetrieveAllPlans:** Sent by the Query/Travel Module to get all plans for deciding which of the travel plans to choose as current plan.
- **StorePresentation:** Sent by the Query/Travel Module to store how the current travel plan should be presented to the user.
- **RetrievePresentation:** Sent by the Query/Travel Module to get the complete description of how the current plan should be presented.
- **RetrieveNextChunk:** Sent by the Response Generation Module to get the next chunk to present. In a straightforward presentation (i.e. the presentation does not change during the presentation phase), the Response Generation Module will send successive RetrieveNextChunk messages to retrieve each piece until all the plan is presented.
- **RetrieveChunk:** Sent to retrieve an arbitrary chunk of the presentation. Which chunk to return is indicated as a parameter. Both chunks already presented and chunks to be presented in the future can be retrieved. This message is intended as a look-up service and does not affect which chunk is returned by RetrieveNextChunk.

- **Clear:** Used to reset the board when the conversion with this customer ends.
- **ClearPresentation:** Used to reset the presentation without affecting the travel plans. May be used if the presentation get interrupted and should start from the beginning next time.

For messages where a response is not needed, the Travel Board will not send a reply. Each message that does require a response has a corresponding message that the Travel Board will send as a reply. For messages requesting information, the answer will simply contain the wanted information. For other messages, some feedback may be wanted from the Travel Board indicating if the action performed was successful or not.

The messages the Travel Board sends are:

- **CurrentPlan:** Reply to RetrieveCurrentPlan.
- **AllPlans:** Reply to RetrieveAllPlans.
- **Presentation:** Reply to RetrievePresentation.
- **PrsentationStored:** Reply to StorePresentation. Used to indicate a mismatch between the presentation and the travel plan marked as "current".
- **NextChunk:** Reply to RetrieveNextChunk.
- **Chunk:** Reply to RetrieveChunk.

As of now, the message set mostly caters the needs of the Query/Travel Module. If other modules have other needs, messages to serve these can easily be added. A consequence of these new services might be that the internal representation of the travel plans and the presentations will have to change.

6.3 Interface

This section describes the interface between the travel board and the other modules in Alparon as a list of services offered by the travel board.

The messages syntax are explained fairly detailed. For lower level syntax, please refer to the explanation of the dialogue acts coding scheme in section 3.2.2.

6.3.1 Storing and retrieving travel plans

Many of these messages take travel plans as arguments. A travel plan is represented with the following syntax:

```
subtravel1
subtravel2
subtravelN
```

where each sub travel has the form

```
Loc(DeP(Station(Place1)))
Tm(DeT(Time1))
Loc(DeP(Station(Place2)))
Tm(ArT(Time2))
...
```

In addition to departure place and time and arrival place and time, information like track number, direction, travel modality etc may be included in the subtravel.

Example: To represent a travel from Venlo to Arnhem we may write:

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

6.3.2 Storing travel plans

`Receive a list of travel plans and store them.`

Message name

```
StoreTravelPlans
```

Message arguments

```
turnnumber
travelplan1
travelplan2
....
travelplanM
```

This message is used to store travel plans on the travel board. Each travel plan must have departure place and time as well as arrival place and time. Other information may also be included, such as track number, transport modality, route name, direction etc. The travel board assumes this information is correct and performs no checks on its consistency.

Example: To store two different travel suggestions for how to get from Venlo to Arnhem, we send a StoreTravelPlans message with the following body:

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))


Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(16:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6258)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(17:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(17:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3060)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(17:34))))
```

Set current plan

```
Set ‘‘current travel plan’’ to the plan given as an argument to this %%@
message.
```

Message name

```
SetCurrentPlan
```

Message arguments

```
turn(TurnNumber)
travelplan
```

The travel plan given as an argument here should be one of the travel plans already on the travel board. The travel board will have to check that this condition is met. The current travel plan is saved along with the turn number where it was selected, thus making it possible to maintain a history of which plans where chosen as current.

Example: To set current travel plan to the first of the two travel plans from the last example, we send a SetCurrentPlan message with the following arguments (given that the dialogue is now in turn number 7):

```
turn(7)
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

Retrieve current travelplan

```
Return the travel plan that was marked as current in the given
turn. Also return which turn the plan was set to current plan.
```

Message name

```
RetrieveCurrentPlan
```

Message arguments

```
turn(TurnNumber)
```

Answer Message name

```
CurrentPlan
```

Answer Message arguments

```
turn(TurnWhenSet)
travelplan
```

The travel plan in the reply is the plan that was current plan in the turn asked for. If no turn number is given, then the most recent current plan is returned, i.e. the plan that is the current plan in this turn. The reply also gives the turn where the plan was chosen as current (i.e., TurnWhenSet <= TurnNumber). If no current plan was chosen in the given turn, the reply will use a "." to indicate no travel plan is available.

Example: To ask for the current travel plan, we send a RetrieveCurrentPlan message with a simple body to the travel board (we assume now that the dialogue is in turn 9):

```
turn(9)
```

The travel board replies with a CurrentPlan message with a slightly longer body. We asked for the travel plan at turn 9. We assume now that current plan did not change after we set it in turn 7. The travel board answers with this travel plan and indicates that it was set in turn 7, implicating that it was still the current plan in turn 9. The body of the message is actually exactly the same as in the corresponding SetCurrentPlan message:

```
turn(7)
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
```

```
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

Retrieve all plans

`Return all travel plans that were on the travel board on the specified turn.`

Message name

`RetrieveAllPlans`

Message arguments
turn(TurnNumber)
Answer Message name

`AllPlans`

Answer Message arguments

```
travelplan1
travelplan2
....
travelplanM
```

The travel plans are here returned using the same syntax as when storing plans with StoreTravelPlans messages.

Example: We assume now that the only travel plans on the board are the two travels from Venlo to Arnhem. We can retrieve them by sending a RetrieveAllPlans message with an empty body to the travel board. The travel board will then reply with a AllPlans message with a the following body (exactly the same as in the StoreTravelPlans message we used to fill the board):

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))
```

```
Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))


Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(16:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6258)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(17:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(17:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3060)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(17:34))))
```

6.3.3 Storing and retrieving presentation chunks

A presentation chunk is a list of information codes that is to be presented by the system in one turn. The presentation of a travel plan is chunked up in pieces to make it easier to comprehend by the user. A presentation is stored as a list of presentation chunks. A presentation chunk is usually a subset of the codes making up a subtravel, where only the information that is to be presented to the user is included. An exception is the representation of a place where one has to change train. This is represented as Via(ChangePlace). If a departure time for the connecting train (i.e. departure from the via-place) is to be presented, this is done by adding DeT(ChangeTime) and DeP(ChangePlace). Similar for arrival at the place of changing trains.

The syntax of presentations is similar to the syntax of travel plans:

```
presentationchunk1
presentationchunk2
...
```

```
presentationchunkN]
```

The presentation chunks are separated with a blank line. Each presentation chunk has the form:

```
Loc(DeP(Station(Place)))
Tm(DeT(Time))
...
```

Any information codes may be present in a presentation chunk. However, each chunk will usually not contain more than three or four codes since there otherwise would be too much information to present in a single utterance.

Example: We consider the first of the travel plans from Venlo to Arnhem. The system may decide to presented the travel in three chunks:

1. Leave from Venlo at 14:33
2. Change train in Nijmegen
3. You will arrive at Arnhem at 15:34

This presentation may be represented like this:

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))

Via(ns(nymegen))

Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

When storing a presentation, a set of chunks making up a presentation is expected. For retrieval, there is a choice between retrieving the complete presentation or a single chunk.

Storing a presentation

```
Store a list of presentation chunks
```

Message name

```
StorePresentation
```

Message arguments

```
travel plan
presentation
```

Answer Message name

`PresentationStored`

Answer Message arguments

`OK`

or

`currenttravelplan`

We are only interested in presentations of the current plan, so the travel board will compare the travel given as an argument in this message to its current travel plan. If they are identical, the presentation will be stored and the travel board answers with the word "OK". Otherwise the presentation is not stored, and the travel board answers by sending the current travel plan.

Example: To store the presentation of the Venlo - Arnhem trip, we send a StorePresentation message with this body:

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))
Loc(DeP(Platform(1b)))
RI(TrT(Train(sneltrein 6250)))
Loc(Dir(Station(Nijmegen)))
Loc(ArP(ns(nymegen)))
Tm(ArT(exact(absolute(15:15))))

Loc(DeP(ns(nymegen)))
Tm(DeT(exact(absolute(15:21))))
Loc(DeP(Platform(1a)))
RI(TrT(Train(Intercity 3052)))
Loc(Dir(Station(Arnhem)))
Loc(Dir(Station(Den Helder)))
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))


Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))

Via(ns(nymegen))

Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

This presentation has three chunks. One possible way to realise this presentation as utterances is:

- Leave from Arnhem at 14:33
- Change train in Nijmegen
- You will arrive in Arnhem at 15:34

Assuming that this is the correct current travel plan, the travel board answers by sending a PresentationStored message with this simple body:

```
OK
```

Retrieving the presentation

```
Return the list of presentation chunks (i.e. the
complete presentation) previously stored on the board.
```

Message name

```
RetrievePresentation
```

Message arguments
none
Answer Message name

```
Presentation
```

Answer Message arguments

```
presentation
```

The presentation is returned using the same syntax as when storing with StorePresentation messages.

Example: Assuming we have stored the presentation of the Venlo - Arnhem trip on the board, we can retrieve this presentation by sending a RetrievePresentation message with an empty body. The travel board will answer with a Presentation message with the following body:

```
Loc(DeP(ns(venlo)))
Tm(DeT(exact(absolute(14:33))))

Via(ns(nymegen))

Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

Retrieving the next chunk of a presentation

`Retrieve the next chunk of the presentation, relative to the last chunk retrieved with this message. If none of the chunks are presented yet, retrieve the first chunk.`

Message name

`RetrieveNextChunk`

Message arguments

`none`

Answer Message name

`NextChunk`

Answer Message arguments

`presentationchunk`

This is the message to be used when presenting the travel plan with the presentation. Note that successive RetrieveNextChunk messages will retrieve different chunks since the chunk retrieved is the next relative to the chunk already returned. RetrieveChunk is therefore non-re-entrant.

Example: Assume the first chunk of the Venlo - Arnhem presentation has been presented to the customer. To get the next chunk, the Response Generation Module sends a RetrieveNextChunk message with an empty body. The Travel Board then answers with a Chunk message with this body:

```
Via(ns(nymegen))
```

Retrieving an arbitrary chunk of the presentation

`Retrieve the presentation chunk with the indicated number`

Message name

`RetrieveChunk`

Message arguments

`chunknumber`

Answer Message name

`Chunk`

Answer Message arguments

`presentationchunk`

This message can be used to look up any chunk of the presentation. The chunks are numbered in the sequence they are meant to be presented, starting with 1 for the first chunk. Chunk numbers can be given as absolute numbers (e.g. 2 for chunk number 2), or relative to the next chunk (e.g. +1 for the chunk that will be retrieved with the next RetrieveNextChunk message, -1 for the one last retrieved by a RetrieveNextChunk message). 0 is an illegal chunk number. Note that successive messages with the same chunk number argument will retrieve the same chunk, so RetrieveChunk is re-entrant.

Example: To retrieve chunk number 3 of the Venlo - Arnhem presentation (i.e. the last chunk), some module may send a RetrieveChunk message with this body:

```
3
```

The Travel Board will answer with a Chunk message with this body:

```
Loc(ArP(ns(arnhem)))
Tm(ArT(exact(absolute(15:34))))
```

6.3.4 Administration messages

These messages are used for clearing parts or all of the board.

Reseting the board

`Reset the board by clearing all information on it`

Message name

`Clear`

Message arguments

`none`

This message is used when the customer hangs up the phone and the conversation is over. The system will now start from fresh with a new customer, so all information on the board is now irrelevant. The Clear message deletes all information, and the board will again become empty.

Example: If our Venlo - Arnhem customer is satisfied with the 14:33 train from Venlo, she might decide to end the conversation. A Clear message will then be sent to the Travel Board. The body is empty (except for the message name), and there is no reply message.

6.3.5 Reseting the presentation

`Reset the presentation by removing all presentation chunks.`

Message name

`ClearPresentation`

Message arguments

`none`

This message can be used when a presentation is interrupted. This would signal that the presentation should not continue but rather be started from the beginning, possibly with a completely new presentation scheme.

Example: If our Venlo - Arnhem customer interrupts the presentation and starts talking about something else, the system might decide that the presentation should be restarted. A ClearPresentation will then be sent to the Travel Board. A new StorePresentation message will have to be sent before the presentation can start again.

6.4 Implementation

The Travel Board is implemented in Perl. The implementation is heavily based on an early implementation of the Information Board written by Guillaume Lecourt. Source documentation can be found in appendix A. Code for the message handling functions (i.e. the part of the code written by me) is listed in appendix D.

Chapter 7

Query Travel Module

This chapter describes the design and implementation of the Query Travel module. As with the Database Manager and Travel Board this chapter is based on a technical report that was written during the design phase and served as requirements and design specification for the implementation of the module [Oldervoll and Vark, 1997b]. I will first define the tasks of the Query Travel module. Then I will discuss some design issues, with a special emphasis on the preparation of the presentations and how this relates to Rats' work discussed in 4.2.2. At the end, the messages sent and accepted by the Query Travel module will be explained.

7.1 Problem Definition

The Query Travel module should execute new queries to the Database manager when needed, store the results at the Travel Board, choose the best travel plan and decide how this plan should be presented to the customer.

When capabilities for allowing the customer to interrupt presentations are added to the system, the Query Travel module must also be able to replan a presentation after it is started. Replanning is not part of this version of the module.

7.1.1 Role in the Dialogue

The Query Travel module is responsible for the problem solving phase of the dialogue (see section 4.1 for an explanation of the phases of the dialogue). The Response Generation Module decides when it is time to enter the problem solving phase. It then sends a message to the Query Travel module. This module retrieves the information the customer is interested in and prepares it for presentation. Upon completion, the Query Travel module sends a message to the Response Generation module. The Response Module can now take the dialogue into the presentation phase by starting the presentation.

Here is a more detailed explanation of how this happens. The end of the information phase is decided by the Response Generation Module. It checks the Information Board to see whether the customer has given enough information for the system to provide an answer. If so, it leads the dialogue into the problem solving phase by sending a message to the Query Travel module. The Query Travel modules then queries the Database Manager, typically for travel plans. To do this it first has to formulate a query that corresponds to the customer's wishes. This information will be retrieved from the Information Board. When travel plans are returned by the Database Manager, the Query/Travel module stores them on the Travel Board. One of the plans is chosen for presentation to the customer. The Query/Travel module prepares this plan for presentation by chunking it up in suitable pieces so the presentation will be easy to comprehend by the customer.

In the rest of this chapter, the tasks of the Query/Travel module will be discussed further.

7.1.2 Tasks

The tasks to be performed by the Query/Travel module can be summarised as: Formulate query, execute query, store travel plans, choose msot appropriate travel plan, and prepare presentation. This will be explained in this section.

Formulate query

A query should be formulated using information from the Information Board. This involves reading slots from the Information Board and synthesising a query from the values.

Execute query

The query should be executed by sending it as a message to the Database Manager. An answer message, normally containing travel plans, will be received.

Store travel plans

Store the received travel plans on the Travel Board. This is done by sending a message.

Choose one travel plan

The travel plan that best fits the query should be chosen. Both newly arrived plans and plans already on the Travel Board must be considered. Criteria for this choice must be obtained from the info board. The algorithm can be very simple (like choosing the plan with arrival or departure time closest to the query criteria), or complex (making predictions about unstated user preferences, like minimising travel time, minimising the number of stopover etc.).

Prepare presentation

It should be decided how the chosen travel plan should be presented to the customer. This is done by chunking it up in suitable presentation chunks. Each chunk should contain a suitable amount of information to be uttered by the system in one turn. This chunking should aim at making it as easy as possible for the customer to understand and remember the information given. Only the information assumed to be important for the customer is included to avoid drowning her in details. In other words, the Grice maxim of informatioveness should be followed. The chunks, as well as information within chunks, should be ordered so as to make it as easy as possible to grasp the contents. Store the presentation on the Travel Board by sending a message.

7.2 Design

This section discusses the design of the Query Travel module. Emphasis in design is made on having a rapid prototype up and running while making the design extendible so more advanced features can be added later. strategies to be refined later.

7.2.1 Formulate query

Query formulation is done by retrieving all slot values from the Information Board, translating them into information codes, and putting these codes into a query message. Only positive slot values will be considered (i.e. the negative slot value “arrival place is not Amsterdam” will not be considered).

7.2.2 Execute query

Executing the query is done by sending a the query message to the Database Manager and receiving a reply message. By doing this in a separate thread the Query Travel module could utilise the waiting time. There are no pressing tasks to be done, however, so we will let the module just be idle while waiting for the reply.

7.2.3 Store travel plans

Storing the travel plans is a straightforward task, done by sending a message to the Travel Board. We choose to first store the plans just returned from the database, then retrieve all plans (including the ones just stored) when deciding which plan to choose. In this way also plans present on the Travel Board are considered when choosing the best plan.

7.2.4 Choose one travel plan

Choosing the best travel plan is a non-trivial task and will be discussed in some length here. The best travel plan is the one that fits best with the query given. All plans will have the departure place and arrival place given in the queries. This is one of the constraints of the Database Manager, so we can assume this to be true. For now, the choice of which travel plan to present to the customer will be done by comparing the time values in the query with the resulting travel plans. Each plan will be given a penalty score indicating how much it differs from the perfect plan. Others could have been used in addition, like preferring express trains to local trains, but for now we will only consider time values.

The calculation of the penalty score will be done using an heuristic function simulating average users preferences. This heuristics will be set using the developers common sense, so it should be easy to change. For good optimisation it will be useful to log all queries as formulated by the customer together with the travel plan the customer finally accepted. A sophisticated solution would be to automatically adjust the heuristics according to this log. This will not be attempted at this stage.

The heuristic rules used are:

- For departure times, too late is preferred to too early.
- For arrival times, too early is preferred to too late.
- For time intervals ("between 7 and 8"), the centre of the interval is preferred.
- If the customer specifies that the arrival or departure should be before a certain time ("before 7"), time values before this time have strong priority to time values after.
- Equivalently for "after": If the customer specifies that the arrival or departure should be after a certain time ("after 7"), time values after this time have strong priority to time values after
- Travel plans with shorter duration are preferred.
- Departure place and arrival place have to match exactly, or the plan will not be considered.

The score will be calculated by giving each time value discrepancy with a certain number of penalty per minute. The weight currently used are given in table 7.1. They are somewhat arbitrary assigned but is easy to change.

This strategy might be made more sophisticated later. For example, the customer might be allowed to choose a special route by giving a city that she wants to travel via.

Table 7.1: Weights used when calculating penalty score

When “before” used:	
Too late	100
Too early	1
When “after” used:	
Too late	1
Too early	100
When “between” used:	
Times off centre of interval	1
Departure time after interval	10
Departure time before interval	100
Arrival time after interval	100
Arrival time before interval	10
When none of “before”, “after” and “between” used:	
Departure time too late	10
Departure time too early	100
Arrival time too late	100
Arrival time too early	10
Long travel duration	20

7.2.5 Prepare presentation

The Query travel module should prepare presentations of travel plans by chunking them up in pieces suitable for the system to present as one utterance. The presentation will be based on the work in [Rats, 1997b].

The presentations produced by the Query Travel module will be in the form of lists of information codes, as defined in section 6.1. The code lists will be transformed into dialogue acts by the Dialogue Act Generation module. These dialogue acts will be translated further into written sentences by the Text Generation module, and eventually into speech by the Text-To-Speech module.

Rats divides presentations into a number of situations. Each situation is characterised by the kind of information given in the query phase and the kind of information returning from the database. She proposes templates for the presentations for the different situations. For example, for the situation where the query gave departure place and arrival place and a time value (either departure or arrival time), and the resulting travel plan is direct travel, the template is:

```
You may depart at [DeT] from [DeP],
then you are at [ArT] in [ArP].
```

Following this template, the Query/Travel module should produce two chunks: Chunk 1 consists of DeT and DeP, chunk 2 consists of ArT and ArP. The actual values for DeT, DeP, ArT and ArP will of course be taken from the travel plan to be presented.

The Query Travel module should be able to use such templates for its presentation preparation. It should be easy to add new templates. If no template match the situation at hand, it should use default rules to produce a presentation. What is needed is then:

- A template format that makes it easy to write new templates
- A mechanism for adding such templates with having to reprogram the system
- A mechanism for selecting an appropriate template
- A mechanism for applying default rules if no template is found

Template format

I have chosen a solution where each template is written in a separate file. The format of this file has taken the template descriptions in [Oldervoll and Vark, 1997c] as starting point to be as natural as possible. A template file consists of three parts:

- Constraints on the query
- Constraints on the results
- The description of how to present the plan

The parts are separated by blank lines. Pieces of information are referred to with slot names. Any slot names may be used, depending on what data is available. The ones currently used are as listed in table 7.3. The constraints parts consist of comma-separated lists of constraints, where each single constraint has either of these forms:

- **"Slot = Value":** satisfied if the given slot has the given value
- **"Slot1 = Slot2":** satisfied if the two given slots have equal, non-blank values
- **"Slot":** satisfied if the given slot has any non-blank value

Values can be numbers (unquoted) or quoted strings. Strings without quotes are interpreted as slot names.

The description of how to present the plan are given as strings containing slot names in square brackets. Each utterance is written on a separate line. The example already given follows this syntax:

```
You may depart at [DeT] from [DeP],
then you are at [ArT] in [ArP].
```

The only significant parts are "[DeT]", "[DeP]", "[ArT]" and "[ArP]". This is an equivalent way of written the example presentation description:

7.3 Slots used

Slot name	Description
DeP	Departure place
ArP	Arrival place
DeT	Departure time
ArT	Arrival time
Day	Date of travel
Dir	Destination of train
Changes	Number of train changes in travel

```
[DeT] [DeP]
[ArT] [ArP]
```

The words in between makes the template more readable, but will not be used by the Query Travel module. However, by specifying full sentences in the template files the Text Generation module might use the same files for deciding which words to use in a presentation.

Lines starting with "#äre treated as comments and hence ignored.

Reading templates

Template files are read by the Query Travel module on start-up. File names are now specified in the code. More flexibility can easily be added:

- Template file names can be specified in a set-up file or specified with a graphical user interface
- A mechanism for adding templates while the system is running can be implemented

Selecting a template

A template is selected by testing the query and results against each template. Templates are tried in a given order and the first matching template will be chosen. More general templates should therefore be put after more specific ones on the template list.

Testing against a template are done by checking the query and the results returned from the database against the constraints given in the template. For a template to match, all constraints must be fulfilled. A presentation will then be generated by using the slot names in the presentation description part of the template, inserting slot values from the results.

Which templates to use

Rats lists a total of 11 templates based on studies of a corpus of 100 human-human OVR dialogues. For some of the templates the data material is very thin (often only one dialogue). As I mentioned in section 4.2.2, I only find the data material sufficient for three of the situations. One of these situations is a case where two travel plans are returned from the database. The Query Travel module will always choose one travel plan as being the best suited, so this situation does not apply. Hence only two of the situations, situation 1 and 3 (see figure 4.1) will be used.

However, there is one problem. For situation 3 Rats gives two different templates, depending on the value for direction (the "Dir" slot) in the travel plan. One travel plan might have more than one value for direction in our database, and then the look-up mechanism used in the current Query Travel implementation dose not work since slot name is used as key in a hash table. The look-up mechanism could be changed, but this could seriously affect effectivity. The solution chosen now is to avoid using constraints on the "Dir" slot. Hence only one template will be used for situation 3.

Hence we currently use 4 templates: Situation 1 and situation 3, with arrival time or departure time given, respectively.

Default rules when no template match

For each presentation the system will check whether any of the available templates are suitable. If not, default rules will be used to produce a presentation. These rules will be based on general principles for information presentation taken from [Rats, 1997b], mainly:

- The presentation should follow temporal order
- Each utterance should contain at least one new information element
- Two information elements per utterance is preferred
- For utterances with two information elements, the new element should come first
- For utterances with three information elements, a given element should come first

If the system is to follow the Grice maxim of informativeness, the presentation should contain all the information the customer wants and nothing more. However, this can be difficult to achieve since it is not easy to foresee what the customer wants. Rats does not emphasis this issue. My suggestions for default rules are:

- Include all time values and station names mentioned in the plan
- Skip all other information (platform numbers, directions etc.).

7.4 Interface

In this section I will first specify the syntax for the messages the Query Travel module accepts, as I did for the Database manager and the Travel Board. However, since this module is an active Discourse Manager module, it also takes initiative to interaction with other modules. Therefore a list of messages sent from the Query Travel module will also be given, grouped by the tasks performed. The syntax of these messages are not specified.

7.4.1 Doing a new query

`Formulate and execute a new query, store resulting plans, choose the best plan and prepare a presentation of it.`

Message name

`NewQuery`

Message arguments

`none`

Answer Message name

`QueryDone`

Answer Message arguments

`failed or succeeded`

This is the only message the module accepts[1]. It starts up the problem solving of the Query Travel. On completion the Query Travel module replies with a QueryDone message with argument succeeded (if a plan is ready for presentation) or failed (if no plan could be found).

7.4.2 Messages Sent to Other Modules

Table 7.4.2 summarises the messages the Query Travel modules send to other modules. When a reply is given, the name of the reply message is given in the table.

[1] All modules using the Java framework explained in chapter 8 also accepts messages "Stop" and "AddMessage"

Table 7.2: Messages sent to other modules, with replies indicated

Module	Message name	Reply name
Information Board	RetrieveValue	Value
Database Manager	QueryCompleteSchedule	CompleteSchedule
Travel Board	StoreTravelPlans	
Travel Board	RetrieveAllPlans	AllPlans
Travel Board	SetCurrentPlan	
Travel Board	StorePresentation	PresentationStored
Response Generation	QueryDone	

7.5 Implementation

The Query Travel module is implemented in Java, using the framework described in chapter 8. Code style follows the conventions given in [CodeConventions, 1997]. Source documenation is done according to guidelines in [JavaDoc, 1998].

Some features are not yet implemented:

- The default rules for presentations are very crude: Information codes for time values and station names are chunked together in groups of three.
- When comparing two travel plans to decide which should be chosen, it is not checked whether the same query was used when calculating scores.
- There is no mechanism for checking if a new query s the same as the previous one but with modified time values (so travel plans on the Travel Board should be kept) or whether it is a completely new query (so the Travel Board should be emptied).

For details on the implementation, see source documentation in appendix B and source code in E.

Chapter 8

The Java Framework

For implementing the Query Travel module I decided to use the Java programming language[K. Arnold, 1996, Campione and Walrath, 1995]. The Alparon Dialogue Manager was specifically designed to allow modules to be implemented in any language that can read text from and write text to sockets. Java was chosen because it has good sockets support, and because I believe it will be a very important language in the years to come. The JavaSpeech API, to be released in the spring or summer of 1998, might also be an interesting alternative for controlling the speech recogniser and text-to-speech system.

8.1 Porting lib from Perl

The existing modules were all implemented in Perl. They all used a Perl library called lib developed as part of Alparon. This library implements messages, servers to send and receive messages, and settings objects to configure the servers. This library was ported fairly easily to Java. Some features were added to provide good Java programming style. For example, all classes got a toString() method so object of the type could be printed using System.out.println(myObject).

8.2 A framework for modules using messages

Every Alparon module will have to deal with setting up sockets for receiving and sending messages, reading and writing messages to and from those sockets etc. There is no reason why programmers should be bothered with these low-level technical details for every new module. The Perl library hides some of these details, but I wanted to achieve a higher level of abstraction. I therefore developed a class MessagingModule to be inherited by any Alparon Java module. This MessagingModule as super class will cause all messaging task to be handled invisibly. Also, all modules will have to specify a set of messages that the module can handle. Using the Java framework, for each message a class inheriting the MessageHandler class should be implemented. Two exceptions UnknownMessageException and MalformedMessageException can be used if things go wrong.

8.3 An example: A simple board

Using this framework makes it much easier to implement new dialogue manager modules. As an example I will show the implementation of SimpleBoard, a simple black board that can store one value. It accepts two messages Store and Retrieve. The complete code is given in figure 8.1. Understanding the code requires knowledge of Java, of course, but the point here is its simplicity. The constructor calls a initialising and registers handlers for two messages. Both the initialiser and the method for adding handlers are inherited from MessagingModule. The two message handler classes calls public methods of the SimpleBoard class, hereby allowing interaction with the module to be done with messages. Lots of the gory details SimpleBoard has been hidden from the programmer:

- The setting up of a server using sockets for reading and writing messages
- The choice of which MessageHandler to use for a given incoming message
- Giving appropriate error messages for malformed messages or unknown message names

8.4 Implementation

Code style follows the conventions given in [CodeConventions, 1997]. Source documenation is done according to guidelines in [JavaDoc, 1998].

One missing feature is to set signal handlers, defining what to do when the module receives a signal like Control-C (kill). The Alparon portmappers use signals to shut down modules, This should be added.

For more details, see source documentation in appendix B and source code in E.

```
import alparon.lib.*;

public class SimpleBoard extends MessagingModule {
    protected static final String NAME = "SIMPLEBOARD";
    protected String value;

    public QueryTravel(int pid) {
            initialise(this, pid);
            addHandler("Store", new StoreHandler(this));
            addHandler("Retrieve", new RetriveHandler(this));
    }

    public void setValue(String newValue) {
        value = newValue;
    }

    public String getValue() {
        return value;
    }

    public static void main(String[] args) throws Exception {
        int pid = Integer.parseInt(args[0]));
        SimpleBoard sb = new SimpleBoard(pid);
            while (sb.server() != null) {
                    sb.listen();
                }
    }
}


public class StoreHandler extends MessageHandler {

    public StoreHandler(MessagingModule module) {
        super(module);
    }

    public void handleMessage(Message msg) throws MalformedMessageException {
        checkName(msg, "Store");
        module.setValue(msg.getBody()[1]);
    }
}


public class RetrieveHandler extends MessageHandler {

    public RetriveHandler(MessagingModule module) {
        super(module);
    }

    public void handleMessage(Message msg) throws MalformedMessageException {
        checkName(msg, "Retrieve");
        module.reply(msg, "Value", module.getValue());
    }
}
```

Figure 8.1: Java code for a simple black board module

Chapter 9

Future work

This chapter will give an outline of future work on the Alparon system. I start with tasks directly related to what I have done, before I briefly note on some work to be done on other parts.

9.1 Information Presentation

Some parts of my work were not fully finished because of time restrictions. In particular, the default rules for preparing presentations should be implemented according what was specified in the design. See notes on the implementation of Query Travel (section 7.5 for a some other tasks to be done.

9.1.1 Usability Testing of the Presentation Strategy

The strategy for information strategy was guided by a wish to improve the presentation done by VIOS, the ASP system currently used for train table information. To check whether this goal has been reached, usability testing should be performed. A possible testing strategy is:

1. **Heuristic evaluation:** A small-scale evaluation should be performed to get a first indication of the effectiveness of the presentation strategy. The design can then be adapted according to results. Nielsen offers a method to perform such evaluations quick and with relative small effort [Nielsen, 1994].

2. **User testing:** Next step would then be to test the presentation strategy on real users. A direct comparison against VIOS should be done to check the effect of our changes in strategy.

To be able to hold such tests, some kind of interface is needed. Ideally we would want speech input and input, but this might require too much work before it is operative. A text-based solution might be the best for testing purposes.

Wizard-of-Oz techniques (a human simulating the machine secretly) might also be used. This might be especially useful for simulating speech input and output.

9.1.2 Extensions of Work Done

More templates should be added. It might be necessary to make the syntax of the template file more powerful. As already mentioned the default rules should be improved.

To follow Rats' recommended approach, a next step would be to allow the user to allow the user interrupt the presentation [Rats, 1997b]. This requires adding functionality for dynamically re-planning presentations to the Query Travel module.

We could also try to use a user model to adapt the presentations to different needs. For example, an experienced user might want a briefer presentations, whereas a user that doesn't know the area in question would need more details on where to change trains etc.

9.2 Other Parts of Alparon

The implementation of the Dialogue Manager is in progress, but need to be finished. To get a demonstrable prototype, speech input and output should be added.

9.2.1 Query Phase

Much of the work to be done on the Dialogue Manager concerns the query phase. An important module will be the Response Generation module, since this module is imperative in deciding the dialogue strategy. Another exciting module will be the Control Board, which is meant to give the opportunity of experimenting with different dialogue strategies.

9.2.2 Speech Input

To add speech input, we obviously need a speech recogniser. It might also be necessary to improve the parser/coder to allow a richer language to be used as input.

9.2.3 Speech Output

Output is currently just indicated by specifying the information each utterance of a presentation should contain. A Dialogue Act Generation module is needed to generate actual dialogue acts to say. From this a Text Generation module would generate a sequence of words. The last step would then be to add a Text-To-Speech module that generated speech from those sentences. Several general Text-To-Speech system are available, for example Festival [Festival, 1998].

Summary

I have in this report described my work with designing and implementing information presentation in Alparon, an automatic speech processing system for offering train time table information by telephone. Tasks covered include formulating and executing queries form the information given by the customer, translating between data formats used in databases and the format chosen in Alparon, storing travel plans, choosing the best plan and preparing a presentation of this plan. Emphasis was put on the strategy for presenting travel plans, which was based on a study of how human operators perform this task [Rats, 1997b]. Three modules are implemented:

- **Database Manager:** Responsible for translating between format used by time table databases and a format I chose for travel information in the Alparon system. This format uses the information codes from Alparon coding scheme as building blocks. As more databases become available, this module will be responsible for coordinating them. The current version uses only one database, namely an electronic timetable service available on the Internet. The current implementation is very simple but allows testing to be done on real data.
- **Travel Board:** Black board module for storing travel plans and presentations of travel plans. Offers services through a set of messages that can easily be expanded. The current implementation does not perform much reasoning on its contents, and the data representation might have to change if more intelligence is needed.
- **Query Travel module:** Performs the tasks of formulating a query, choosing the best travel plan and preparing the presentation of the chosen travel plan. Executes query and stores results through sending messages. The choice of the best travel plan is implemented quite generic so the strategy can be optimised later. Presentation preparation tries to find a presentation template that matches the query and travel plan given. If no template can be found, default rules are applied. The implementation of default rules is crude. New templates can be added easily.

The Database Manager and the Travel Board were implemented in Perl. The Query Travel module was implemented in Java. A Java framework was developed to allow reuse of messaging functionality through inheritance.

No conclusions can be made on the usefulness of the presentation strategy since no usability testing is done. This is because the Dialogue Manager is not fully implemented yet.

Appendix A

TravelBoard Source Doucumentation

This appendix documents the Perl source code of the Travel Board. The text is generated from the source code by the program perldoc which is part of the standard Perl distribution.

Large parts of the Travel Board uses code from an early version of the Inforamtion Board, written by Guillaume Lecourt [Lecourt and Vark, 1997]. Since this appendix is generated from the source code, much of the text is written by Guillaume Lecourt.

Appendix B

Query Travel Source Documentation

This appendix documents the Java source code of the Query Travel Module. The text is generated from the source code by the program JavaDoc from Sun Microsystems, Inc.

Appendix C

Java Framework Source Documentation

This appendix documents the Java source code of the Java framework `alparon.lib`. The text is generated from the source code by the program JavaDoc from Sun Microsystems, Inc.

Appendix D

Travel Board Code Listing

Thei appendix lists the Perl source code of the Travel Board. Only message handler functions are listed here, since they are the only part I have written.[1]

```
##############################

=head1 TRAVEL BOARD MODULE TOOLS

=head2 FUNCTIONS THAT PROSSESS MESSAGES

    The following functions are called by HandleMessage().

=item StoreTravelPlans ()

I<Arguments>: a turn number, a list of travel plans

I<Returns>:

=cut

##############################

sub StoreTravelPlans {
    my $msg = shift;
    my @body = $msg->body();
    shift @body;    ### remove message name
    my $turnnumber = shift @body;
    my $planlist = shift @body;
```

[1]The rest of the Travel Board uses code written by Guillaume Lecourt.

```
    ### &splitcomma expects a comma after the last element in the list
    $planlist .= ",";
    my @newplans = splitcomma($planlist);

    ### @newplans now holds the new travel plans - add them to @travelplans
    ### All plans end with a comma which has to be chopped of

    my $plan;
    foreach $plan (@newplans) {
        chop($plan);
        push @{$travelplans{$plan}}, $turnnumber;
    }

    ### everything ok ###
    return undef;
}

##############################

=item SetCurrentPlan ()

I<Arguments>: a turn number, a travel plan

I<Returns>:

=cut

##############################

sub SetCurrentPlan {
    my $msg = shift;
    my @body = $msg->body();
    shift @body;    ### remove message name
    my $turnnumber = shift @body;
    my $newcurrentplan = shift @body;

    if (not ($travelplans{$newcurrentplan})) {
        die "Current travel plan #$newcurrentplan# is unknown to the travel board";
    }

    push(@currentplans,([$newcurrentplan, $turnnumber]));

    ### Check if the new plan ruined the sorting of @currentplans
    my $new = @currentplans - 1;
    print "new=$new\n";
```

```
    my @tmp;
    ### 1 is the entry for turn number

    while (($new!=0) and
           ($currentplans[$new]->[1] < $currentplans[$new-1]->[1])) {
        ### sorting is ruined - swap elements to do bubble-sort
        print "sorting...\n";
        @tmp = @{$currentplans[$new]};
        @{$currentplans[$new]} = @{$currentplans[$new-1]};
        @{$currentplans[$new-1]} = @tmp;
        $new--;
    }

    ### everything ok ###
    return undef;
}

##############################

=item RetrieveCurrentPlan ()

I<Arguments>: a turn number

I<Returns>: a turn number, a travel plan

=cut

##############################

sub RetrieveCurrentPlan {
    my $msg = shift;
    my @body = $msg->body();
    shift @body;    ### remove message name

    my $index = @currentplans - 1;
    my @response = ();

    if ($body[0] =~ /^turn/i) {
        ## turn number given
        my $turnnumber = shift @body;
        ## reading @currentplans from the end
        ## until an entry with low enough turn number is found
        while (($index>=0) and ($currentplans[$index][1] > $turnnumber)) {
            $index--;
        }
    }
```

```
    else {
        ## no turn number given - using last current plan
        ## $index allready correct - do nothing
    }

    if ($index < 0) {
        ### No currentplan was chosen at the indicated turn
        push @response, (".");
    }
    else {
        ### Returning currentplan
        my $planentry = $currentplans[$index];
        ### setting @response to turnnumber+\n+currentplan
        push @response, ($planentry->[1], $planentry->[0]);
    }

    ### everything ok ###
    reply($msg, @response);
    return undef;
}

##############################

=item RetrieveAllPlans ()

I<Arguments>: a turn number

I<Returns>: a list of travel plans

=cut

##############################

sub RetrieveAllPlans {
    my $msg = shift;
    my @body = $msg->body();
    shift @body;    ### remove message name
    my $turnnumber = shift @body;
    my $plan;
    my @turnlist;
    my $turn;
    my $found;
    my $answer = "";

    ### loop through all plans on the board
    foreach $plan (keys(%travelplans)) {
```

```
        ### loop through the list of turns to check if the
        ### plan was on the board at the specified turn
        print "plan=", $plan, "\n";
        print "turns: #", @{$travelplans{$plan}}, "#\n";
        @turnlist = @{$travelplans{$plan}};
        $found = 0;

        foreach $turn (@turnlist) {
            if ($turn <= $turnnumber) {
                $found = 1;
                ###jump out of the foreach $turn loop
                last;
            }
        }

        if ($found) {
            ### add this travel plan to the answer list
            $answer .= "$plan,";
        }
    }

    ### remove comma after the last plan and give the answer as response
    chop($answer);
    reply($msg, ($answer));

    ### everything ok ###
    return undef;
}

##############################

=item StorePresentation ()

I<Arguments>: a travel plan, a presentation

I<Returns>: "ok" or a travelplan or  "."

=cut

##############################

sub StorePresentation {
    my $msg = shift;
    my @body = $msg->body();
    shift @body;    ### remove message name
    my $receivedplan = shift @body;
```

```
    my $receivedpresentation = shift @body;
    my @response;

    if (@currentplans > 0) {
        my $realcurrentplan = $currentplans[$#currentplans][0];

        if ($receivedplan eq $realcurrentplan) {
            $presentation = $receivedpresentation;
            @response = ("ok");
        }
        else {
            ### presented plan is not current travel plan
            @response = ($realcurrentplan);
        }
    }
    else {
        ### no current travel plan chosen
        @response = (".");
    }

    ### everything ok ###
    reply($msg, @response);
    return undef;
}

##############################

=item RetrievePresentation ()

I<Arguments>:

I<Returns>: a presentation

=cut

##############################

sub RetrievePresentation {
    my $msg = shift;
    reply($msg,($presentation));
    ### everything ok ###
    return undef;
}
```

Appendix E

Query Travel Code Listing

```
/*
 * QueryTravel
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import COM.stevesoft.pat.Regex;

/**
 * Alparon Discourse Manager module for doing queries, choosing the
 * best travel plan and planning the presentation of this plan.
 *
 * This module listens for messages. The only message currently
 * supported is NewQuery. On reception of a NewQuery message, it
 * does this:
 * <ul>
 * <li>Formulates a database query, using information from Inforamtion
 *     Board
 * <li>Executes the query by sending a message to the Database Manager
 * <li>Stores the results from Database manager on Travel Board
 * <li>Chooses the best of the travel plans currently on the Travel
 *     Board
 * <li>Chunks up the chosen travel plan is pieces suitable for
 *     presentation in one system utterance.
 * Infoarmtion Board, Database Manager, Travel Board and Information
 * Board are other Alparon Dialogue manager modules.
 * Travel plans are description of how to get from one place to
 * another using the public transport in question (that is, trains).
 *
 * @author Thomas Oldervoll
```

```
 * @version $Id$
 */
public class QueryTravel extends MessagingModule {
    // Constants
    protected static final String NAME = "QUERYTRAVEL";
    protected static final String PACKAGE = "alparon.querytravel";

    // Variables
    protected Query query;
    protected TravelPlans travelPlans;
    protected TravelPlan currentPlan;
    protected Presentation presentation;

    // Constructor

    /**
     * @param pid       the integer process id of the process
     *                  that created the object
     */
    public QueryTravel(int pid) {
        initialise(this, pid);
        addHandler("NewQuery", new NewQueryHandler(this));
    }

    /**
     * Queries for new travel plans, stores them, chooses one plan
     * and prepares it for presentaytion. Called by the message handler
     * for "NewQuery" messages.
     */
    public void newQuery() {
        // Formulate
        Debug.println("Formulating...", 7);
        query = new Query(this);
        // Send and receive
        Debug.println("Sending and receiving...", 7);
        travelPlans = query.run();

        if (travelPlans == null) {
            Debug.println("No plans retrieved...", 7);
            presentation = null;
        }
        else {
            // Store new plans at Travelboard
            travelPlans.store(this);

        // Read plans from Travel Board to make sure plans already there
        // are taken in consideration
        send("TRAVELBOARD", "RetrieveAllPlans", "");
        Message allPlansMsg = receive("AllPlans");
        intr codesLength = allPlansMsg.body().length - 1;
```

```
    String[] allPlansCodes = new String[codesLength];
    System.arraycopy(allPlansMsg.body(), 1,
                     allPlansCodes, 0, codesLength);
    travelPlans = newTravelPlans(allPlansCodes);

        // Choose best plan
        Debug.println("Choosing...", 7);
        currentPlan = travelPlans.choose(query.slots());
        Debug.println("Travelplans:\n" + travelPlans.toString(), 9);
        Debug.println("\nCurrent plan:\n" + currentPlan.toString(), 9);

        // Chunk presentation of chosen plan
        Debug.println("Chunking...", 7);
        presentation = new Presentation(query, currentPlan);
        presentation.store(this);
    }

    // Report to RESPONSE module
    Debug.println("Reporting...", 7);
    report();
}

/**
 * Sends a message to the Response Generation module, saying
 * if the query just finished was a success or not.
 * A query is a success if it leads to a travel plan beeing ready
 * for presentation.
 */
private void report() {
    String status = (travelPlans == null)? "failed" : "succeded";
    String[] contents = {status};
    send("RESPONSE", "QueryDone", contents);
}

/**
  * Unit testing routine.
  * Run this by running <code>java Settings</code>.
  *
  * @param args             a string giving the process ID (optional)
  * @exception Exception    if some unexpected happens, like a
  *                         failure in one of the other moduels
  */
public static void main(String[] args) throws Exception {
    int pid = ((args.length == 0) ? 42 : Integer.parseInt(args[0]));
    Debug.level(9);
    Debug.println(NAME + " started with pid " + pid, 2);

    QueryTravel qt = new QueryTravel(pid);
    Debug.println("Running: " + (qt.server() != null), 9);
```

```
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");

        try {
            // resume listening after catching exceptions
            while (qt.server() != null) {
                Debug.println("Running: " + (qt.server() != null), 9);
                try {
                    qt.listen();
                    Debug.println(NAME + " finished.", 2);
                }
                catch (UnknownMessageException e) {
                    System.out.println(e.getMessage());
                }
                catch (MalformedMessageException e) {
                    System.out.println(e.getMessage());
                }
            }
        }
        finally {
            // clean up before exiting
            qt.stop();
        }
    }

}
```

```
/*
 * NewQueryHandler
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;

/**
 * Handler for "NewQuery" messages to the Query Travel module.
 * Simply calls <code>QueryTravel.newQuery</code>.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */
public class NewQueryHandler extends MessageHandler {

    /**
     * Constructs a handler connected to a given module.
     *
     * @param module    a messaging module this handler should serve
     */
    public NewQueryHandler(MessagingModule module) {
        super(module);
    }

    /**
     * Handles "NewQuery" messages by calling <code>newQuery</code>
     * of the connected Query Travel module.
     *
     * @param msg       the message to handle
     * @exception ClassCastException if the connected module is not
     *                  a <code>QueryTravel</code> module
     * @exception MalformedMessageException if the message is not a
     *                  "NewQuery" message
     */
    public void handleMessage(Message msg) throws MalformedMessageException {
        checkName(msg, "NewQuery");
        ((QueryTravel)module).newQuery();
    }

}
```

```
/*
 * Query
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;

/**
 * A representation of a query to the Database Manager.
 * A query has a list of inforamtion codes that are given information.
 * Functionality a query offers:
 * <ul>
 * <li>Can compose itself by asking the Information Board
 *     about which information is given.
 * <li>Can execute itself by sending a message to the Databse Manager.
 * </ul>
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */
public class Query {

    // Variables
    protected Connection connection;
    protected Slots slots;
    private int numberOfReplies = 0;     // "0" means use default

    // Constructors

    /**
     * Constructs a query that expects a default number of travel
     * plans in the reply.
     *
     * @param connection    the connection used for sending and
     *                      receiving messages
     */
    public Query(Connection connection) {
        this(connection, 0);
    }

    /**
     * Constructs a query that expects a specified number of travel
     * plans in the reply.
     *
     * @param connection    the connection used for sending and
     *                      receiving messages
     * @param replies       the number of replies requested
```

```
 */
public Query(Connection connection, int replies) {
    this.connection = connection;
    Debug.println("new Query", 9);
    numberOfReplies = replies;
    compose();
}

/*
 * @return slots        the information slots the query consits of
 */
public Slots slots() {
    return slots;
}

/*
 * Constructs a query, executes it and returns the travel plans
 * received.
 *
 * @return              a collection of travel plans
 */
public TravelPlans run() {
    Message inMsg;
    TravelPlans returnedPlans;
    String[] codeList;

    // compose message
    InfoCode[] codes = slots.toInfoCodes();
    if (codes.length == 0) {
        // INFOBOARD returned no slot values
        return null;
    }
    else {
        // Send query to DATABASEMANAGER
        String[] body = new String[codes.length + 2];
        body[0] = "QueryCompleteSchedule";
        body[1] = String.valueOf(numberOfReplies);
        for (int i = 2; i < codes.length + 2; i++) {
            body[i] = codes[i-2].toString();
        }

        // Send and wait for reply
        connection.send("DATABASEMANAGER", body);
        inMsg = connection.receive("CompleteSchedule");

        // Extract travel plans from reply
        codeList = new String[inMsg.body().length - 1];
        System.arraycopy(inMsg.body(), 1, codeList, 0, codeList.length);
        returnedPlans = new TravelPlans(codeList);
        return returnedPlans;
```

```java
        }
    }

    /**
     * Composes a new query by reading inforamtion from the
     * Info Board.
     */
    private void compose() {
        Message inMsg;
        String[] body = {"RetrieveValue", "ArP", "DeP", "ArT", "DeT", "Day"};
        String[] slotStrings;

        connection.send("INFOBOARD", body);
        inMsg = connection.receive("Value");

        // extract slot information from the message
        slotStrings = new String[inMsg.body().length - 1];
        System.arraycopy
            (inMsg.body(), 1, slotStrings, 0, inMsg.body().length - 1);

        // convert to InfoCode format
        slots = new Slots(slotStrings);
        Debug.println(slots.toString(), 7);
    }

    private InfoCode[] slotsToCodes(String[] slotStrings) {
        InfoCode[] codes = new InfoCode[slotStrings.length];
        StringTokenizer tok;

        for (int i = 0; i < slotStrings.length; i++) {
            tok = new StringTokenizer(slotStrings[i]);
            codes[i] = new InfoCode(tok.nextToken(), tok.nextToken());
        }

        return codes;
    }

}
```

```
/*
 * TravelPlans
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import COM.stevesoft.pat.Regex;

/**
 * A collection of travel plans. Each travel plan is a
 * <code>TravelPlan</code> object. Offers this functionality:
 * <ul>
 * <li> Read travel plans from the code list format returned by
 *      the Database Manager
 * <li>Choose the travel plan in the collection that best fits
 *     a given query
 *<li>Store all travel plans on the Travel Board
 *</ul>
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see InfoCode
 * @see Match
 * @see Slots
 * @see SubTravelPlan
 * @see TravelPlan
 */
public class TravelPlans {
    // Variable
    public Vector plans = new Vector();

    // Constructors

    /**
     * Constructs empty travel plan collection.
     */
    public TravelPlans() {
    }

    /**
     * Constructs a travel plan collection and fills it with travel
     * plans given as a list of information codes.
     *
     * @param codeList  the string array list of information codes
     */
    public TravelPlans(String[] codeList) {
```

```
        CodeParser parser = new CodeParser();
        plans = parser.parse(codeList);
        //      Debug.println("new plans:\n" + plans.elementAt(0).toString(), 9);
    }

    /**
     * Adds a set of travel plans to the collection.
     * The plans are given as a list of inforamtion codes.
     *
     * @param codeList  the string array list of information codes
     */
    public void addPlans(String[] codeList) {
        CodeParser parser = new CodeParser();
            Vector newPlans = parser.parse(codeList);
        for (int i = 0; i < newPlans.size(); i++) {
            plans.addElement(newPlans.elementAt(i));
        }
    }

    /**
     * Adds one travel plan to the collection.
     *
     * @param plan      the travel plan to add
     */
    public void addPlan(TravelPlan plan) {
        plans.addElement(plan);
    }

    /**
     * Stores all travel plans in the collection on the Travel Board.
     *
     * @param connection    the connection to use for sending the
     *                      message
     */
    public void store(Connection connection) {
        //      String[] body = new String[plans.size() + 1];
        //      body[0] = "StoreTravelPlans";
        //      System.arraycopy(toStringArray(), 0, body, 1, plans.size());
        //      connection.send("TRAVELBOARD", body);
        connection.send("TRAVELBOARD", "StoreTravelPlans", toStringArray());
    }

    /**
     * Returns the travel plan of the collections that fits best
     * with the given query. The comparison of two plans is
     * done by instantiating a <code>Match</code> object.
     *
     * @param querySlots    slots the containing the information
     *                      given in the query
     */
```

```java
public TravelPlan choose(Slots querySlots) {
    Enumeration planEnum = plans.elements();

    if (!planEnum.hasMoreElements()) {
        // No plans available
        return null;
    }

    // Find the plan that matches the query best

    TravelPlan thisPlan;
    Match thisMatch;
    // Initialise bestPlan to the first plan
    TravelPlan bestPlan = (TravelPlan) planEnum.nextElement();
    Match bestMatch = new Match(bestPlan.global(), querySlots);

    while (planEnum.hasMoreElements()) {
        thisPlan = (TravelPlan) planEnum.nextElement();
        thisMatch = new Match(thisPlan.global(), querySlots);
        if (thisMatch.better(bestMatch)) {
            bestMatch = thisMatch;
            bestPlan = thisPlan;
        }
    }
    return bestPlan;
}

/**
 * Returns all travel plans in collection, represented with
 * as a list of information codes. This is the format Travel
 * Board expects in messages.
 *
 * @return          a string array with the information codes
 */
public String[] toStringArray() {
    Vector strings = new Vector();
    String[] planStrings;
    int elements = 0;
    for (int i=0; i < plans.size(); i++) {
        planStrings =
            ((TravelPlan) plans.elementAt(i)).toStringArray();
        for (int j=0; j < planStrings.length; j++) {
            strings.addElement(planStrings[j]);
        }
        // Add one extra blank line to separate travelplans
        strings.addElement("");
    }
    String[] stringArray = new String[strings.size()];
    strings.copyInto(stringArray);
    return stringArray;
```

```
    }

    /**
     * Returns all travel plans represented with a single,
     * multi-line string giving the information codes.
     *
     * @return          a string with the information codes
     */
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        for (int i=0; i < plans.size(); i++) {
            buffer.append((plans.elementAt(i)).toString() + "\n");
        }
        return buffer.toString();
    }

}
```

```
/*
 * TravelPlan
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import COM.stevesoft.pat.Regex;

/**
 * A description of how to travel by public transport from one place
 * to another. A travel plan consists of a number of one or more
 * sub travel plans. Each sub travel plan consists of a number of
 * information codes from the Alparon coding scheme.
 * <p>
 * A <code>TravelPlan</code> object has soem functionality beyond
 * being a collection of SubTravelPlan objects:
 * <ul>
 * <li>Global slots, that is the departure time and place,
 *     arrival time and place and number of train changes
 *     (assuming data as about trains) for the complete travel.
 * <li>Look-up possibilities, where "DeP2" can be used as a
 *     slot name meaning "departure place for sub travel 2".
 * </ul>
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see InfoCode
 * @see Slots
 * @see SubTravelPlan
 * @see TravelPlans
 */
public class TravelPlan implements Constrainable {

    // Variable
    private Vector subTravels = new Vector();

    // Public methods

    /**
     * Adds a sub travel plan to the end of this travel plan.
     *
     * @param subTravel     the sub travel to add
     */
    public void addSubTravel(SubTravelPlan subTravel) {
        subTravels.addElement(subTravel);
    }
```

```
/**
 * Return an array of the sub travels in this travel plan.
 *
 * @return      the sub travel plan array
 */
public SubTravelPlan[] subTravels() {
    SubTravelPlan[] subTravelArray = new SubTravelPlan[subTravels.size()];
    subTravels.copyInto(subTravelArray);
    return subTravelArray;
}

/**
 * Returns global informtion for this travel plan.
 * That is, the departure time and place,
 * arrival time and place and number of train changes
 * (assuming data as about trains) for the complete travel.
 */
public Slots global() {
    /*
     * All information is recalculated each time global is called
     * to assure consistency.
     * Using regular expression from COM.stevesoft.pat.Regex
     * Escape sequences in the search patterns are preceeded by \\
     */
    Slots slots = new Slots();
    Regex placeExp;
    Regex timeExp;
    Regex dateExp;
    String[] codes;
    boolean searchForDay;

    // Search for date, departure place and -time in the first subtravel
    codes = ((SubTravelPlan) subTravels.elementAt(0)).toStringArray();
    placeExp = new Regex("Loc\\(DeP\\(Station\\((.*)\\)\\)\\)");
    timeExp = new Regex("Tm\\(DeT\\(Exact\\(absolute\\((.*)\\)\\)\\)\\)");
    dateExp = new Regex("Tm\\(GIT\\(Day\\(Date\\((.*)\\)\\)\\)\\)");
    for (int i = 0; i < codes.length; i ++) {
        if (placeExp.search(codes[i])) {
            slots.add("DeP", placeExp.stringMatched(1));
        }
        if (timeExp.search(codes[i])) {
            slots.add("DeT", timeExp.stringMatched(1));
        }
        if (dateExp.search(codes[i])) {
            slots.add("Day", dateExp.stringMatched(1));
        }
    }

    // Search for arrival place and time in the last subtravel
```

```
    // If no DeT found, also search for date
    // If DeT found but not date, also search for date
    codes = ((SubTravelPlan)
             subTravels.elementAt(subTravels.size() - 1)).toStringArray();
    searchForDay =
        (slots.value("DeT") == null) || (slots.value("Day") == null);
    placeExp = new Regex("Loc\\(ArP\\(Station\\((.*)\\)\\)\\)");
    timeExp = new Regex("Tm\\(ArT\\(Exact\\(absolute\\((.*)\\)\\)\\)\\)");
    dateExp = new Regex("Tm\\(GIT\\(Day\\(Date\\((.*)\\)\\)\\)\\)");
    for (int i = 0; i < codes.length; i ++) {
        if (placeExp.search(codes[i])) {
            slots.add("ArP", placeExp.stringMatched(1));
        }
        if (timeExp.search(codes[i])) {
            slots.add("ArT", timeExp.stringMatched(1));
        }
        if (searchForDay && dateExp.search(codes[i])) {
            slots.add("Day", dateExp.stringMatched(1));
        }
    }

    // Add number of changes
    slots.add("Changes", (new Integer(subTravels.size() - 1)).toString());

    return slots;
}

/**
 * Return the string information code that corresponds to a given
 * slot name, where "DeP2" can be used as a
 * slot name meaning "departure place for sub travel 2".
 *
 * @param slotName  the string slot name to look up
 * @return          the string information code corresponding to
 *                  the slot name
 */
public String lookUp(String slotName) {
    Regex numbered = new Regex("/(\\w+)(\\d+)");
    if (numbered.search(slotName)) {
        // Slot name ends in a number.
        // Refers to a slot of a certain subtravel (1 is first subtravel).
        int subTravelNo = Integer.parseInt(numbered.stringMatched(2));
        slotName =  numbered.stringMatched(1);
        SubTravelPlan subTravel = subTravels()[subTravelNo - 1];
        return (String) subTravel.codes().get(slotName);
    }
    else {
        // Slot name without number. Refer to a global value for the plan.
        return global().value(slotName);
    }
```

```
    }

    /**
     * Return the <code>InfoCode</code> object that corresponds to a
     * given slot name, where "DeP2" can be used as a
     * slot name meaning "departure place for sub travel 2".
     *
     * @param slotName  the string slot name to look up
     * @return          the info code corresponding to
     *                  the slot name
     */
    public InfoCode lookUpCode(String slotName) {
        Regex numbered = new Regex("(\\w+)(\\d+)$");
        Debug.println("Looking up #" + slotName + "#", 9);
        if (numbered.search(slotName)) {
            // Slot name ends in a number.
            // Refers to a slot of a certain subtravel (1 is first subtravel).
            int subTravelNo = Integer.parseInt(numbered.stringMatched(2));
            slotName =  numbered.stringMatched(1);
            Debug.println("Slot #" + slotName + "#, subtravel "
                          + subTravelNo, 9);
            SubTravelPlan subTravel = subTravels()[subTravelNo - 1];
            return (InfoCode) subTravel.codes().get(slotName);
        }
        else {
            // Slot name without number. Refer to a global value for the plan.
            return new InfoCode(slotName, global().value(slotName));
        }
    }

    /**
     * Returns all information codes in this travel plan as a
     * array of strings.
     *
     * @return      a string array of information codes
     */
    public String[] toStringArray() {
        Vector strings = new Vector();
        String[] subStrings;
        int elements = 0;
        for (int i=0; i < subTravels.size(); i++) {
            subStrings =
                ((SubTravelPlan) subTravels.elementAt(i)).toStringArray();
            for (int j=0; j < subStrings.length; j++) {
                strings.addElement(subStrings[j]);
            }
            // Add blank line to separate subtravels
            strings.addElement("");
        }
        String[] stringArray = new String[strings.size()];
```

```
        strings.copyInto(stringArray);
        return stringArray;
    }

    /**
     * Gives a short description of this travel plan.
     *
     * @return      a string describing the travel plan.
     */
    public String toString() {
        Slots global = global();
        StringBuffer buffer = new StringBuffer();

        buffer.append("From " + global.value("DeP"));
        if (global.value("DeT") != null) {
            buffer.append(" at " + global.value("DeT"));
            if (global.value("Day") != null) {
                buffer.append(" " + global.value("Day"));
            }
        }
        buffer.append(", to " + global.value("ArP"));
        if (global.value("ArT") != null) {
            buffer.append(" at " + global.value("ArT"));
            if ((global.value("Day") != null)
                && (global.value("DeT") == null)) {
                // There is a date, and it is not mentioned yet
                buffer.append(" " + global.value("Day"));
            }
        }
        return buffer.toString();
    }

}
```

```
/*
 * SubTravelPlan
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;

/**
 * A description of a direct connection between two stations.
 * Consists of a number of information codes from the Alparon
 * coding scheme.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see InfoCode
 * @see Slots
 * @see TravelPlan
 * @see TravelPlans
 */
public class SubTravelPlan {
    // Variables
    private Hashtable codes = new Hashtable();

    // Accessor

    /**
     * @return codes    the hash table of information codes
     *                  of this sub travel
     */
    public Hashtable codes() {
        return codes;
    }

    // Public methods

    /**
     * Adds an information code, given as an <code>InfoCode</code>
     * objcet, to the end of this sub travel.
     *
     * @param code      the info code to add
     */
    public void addCode(InfoCode code) {
        codes.put(code.slotName(), code);
    }

    /**
```

```
     * Adds an information code, given as a string, to the end
     * of this sub travel.
     *
     * @param code      the string information code to add
     */
    public void addCode(String codeString) {
        InfoCode newCode = new InfoCode(codeString);
        Debug.println("adding code " + codeString, 9);
        addCode(newCode);
    }

    /**
     * @return      a string array with all information codes of
     *              this sub travel
     */
    public String[] toStringArray() {
        int size = codes.size();
        String[] strings = new String[size];
        Enumeration enum = codes.elements();
        for (int i=0; i < size; i++) {
            strings[i] = enum.nextElement().toString();
        }
        return strings;
    }

    /**
     * @return      a multi-line string with all information codes of
     *              this sub travel
     */
    public String toString() {
        StringBuffer str = new StringBuffer();
        InfoCode code;
        Enumeration enum = codes.elements();
        while (enum.hasMoreElements()) {
            code = (InfoCode) enum.nextElement();
            str.append(code.toString() + "\n");
        }
        return str.toString();
    }

}
```

```
/*
 * InfoCode
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import COM.stevesoft.pat.Regex;

/**
 * An information code, as specified in the Alparon coding scheme.
 * InfoCode can between two formats for this information:
 * <ul>
 * <li>A string giving the complete format. Example:
 *     "Loc(DeP(Station(Delft)))"
 * <li>A pair of strings giving a slot name and a slot value.
 *     Example: name "DeP", value "Delft"
 * </ul>
 *
 * The name/value pairs are handy since the names acan be used
 * as hash table keys (see SubTravelPlan). However,
 * not all information codes can be written as name/value pairs.
 * For those codes, both name and value will be set to
 * the code itself making hashing possible also for these.
 * It is also possible to invnet new slots for convenience, even
 * though they are not in the Alparon coding scheme. An example
 * of such a slot is Changes, used in <code>TravelPlan.global</code>.
 * Such "fake codes" will get their code value set to <code>null</code>.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Slots
 * @see SubTravelPlan
 */
public class InfoCode {
    // Variables
    private String code;
    private String slotName;
    private String slotValue;

    // Constructors

    /**
     * Constructs an info code from the complete code string.
     * Tries to convert the code into a name/value pair.
     *
     * @param codeString    the string infoarmtion code
     */
    public InfoCode(String codeString) {
```

```
        code = codeString;
        parseCode(code);
    }

    /**
     * Constructs an info code from a slot name/value pair.
     * Converts the name/value pair into the full code.
     *
     * @param name      the string slot name
     * @param value     the string slot value
     */
    public InfoCode(String name, String value) {
        slotName = name;
        slotValue = value;
        makeCode(name, value);
    }

    // Accessors

    /**
     * @return      the string slot name
     */
    public String slotName() {
        return slotName;
    }

    /**
     * @return      the string slot value
     */
    public String slotValue() {
        return slotValue;
    }

    /**
     * @return      the string full information code
     */
    public String code() {
        return code;
    }

    /**
     * Returns the code string. For objects that have no
     * <code>null</code> strings, return a combination of
     * slot name and slot value.
     *
     * @return      a string representation of this code
     */
    public String toString() {
        if (code == null) {
            return slotName + " " + slotValue;
```

```
        }
        else {
            return code;
        }
    }


    /**
     * Generates a code string given a slot name and slot value.
     * The reverse transormation of parseCode.
     *
     * @param name      the string slot name
     * @param value     the string slot value
     */
    private void makeCode(String name, String value) {
        if (name.equals("DeP")) {
            code = "Loc(DeP(Station(" + value + ")))";
        }
        else if (name.equals("DeT")) {
            code = "Tm(DeT(" + value + "))";
        }
        else if (name.equals("ArP")) {
            code = "Loc(ArP(Station(" + value + ")))";
        }
        else if (name.equals("ArT")) {
            code = "Tm(ArT(" + value + "))";
        }
        else if (name.equals("Day")) {
            code = "Tm(GIT(Day(Date(" + value + "))))";
        }
        else if (name.equals("Dir")) {
            code = "Loc(Dir(Station(" + value + ")))";
        }
        else {
            code = null;
            System.err.println("Warning: Slot " + slotName +
                               " has no code equivalent.");
        }
    }

    /**
     * Generates a slot name and slot value given a code string.
     * The reverse transformation of makeCode.
     *
     * @param code      the string information code
     */
    private void parseCode(String code) {

        // Define regular expression for code strings for each slot
        Regex dePCode = new Regex("Loc\\(DeP\\(Station\\((.*)\\)\\)\\)");
```

```
        Regex deTCode = new Regex("Tm\\(DeT\\((.*)\\)\\)");
        Regex arPCode = new Regex("Loc\\(ArP\\(Station\\((.*)\\)\\)\\)");
        Regex arTCode = new Regex("Tm\\(ArT\\((.*)\\)\\)");
        Regex dayCode = new Regex("Tm\\(GIT\\(Day\\(Date\\((.*)\\)\\)\\)\\)");

        // Check code for match against each of the slot pattern
        if (dePCode.search(code)) {
            slotName = "DeP";
            slotValue = dePCode.stringMatched(1);
        }
        else if (deTCode.search(code)) {
            slotName = "DeT";
            slotValue = deTCode.stringMatched(1);
        }
        else if (arPCode.search(code)) {
            slotName = "ArP";
            slotValue = arPCode.stringMatched(1);
        }
        else if (arTCode.search(code)) {
            slotName = "ArT";
            slotValue = arTCode.stringMatched(1);
        }
        else if (dayCode.search(code)) {
            slotName = "Day";
            slotValue = dayCode.stringMatched(1);
        }
        else {
            /* Code matches none of the slot patterns.
             * Set slotName = slotValue = code,
             * so the InfoCode can be hashed anyway.
             */
            slotName = code;
            slotValue = code;
        }
    }

}
```

```
/*
 * CodeParser
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import java.io.*;
import java.util.*;

/**
 * A parser that reads a list of information codes on the format
 * used in Alparon messages and returns a vector of
 * <code>TravelPlan</code> objects.
 * Each line in the input is one information code.
 * Sub travel plans are divided by a blank line.
 * Travel plans are divided by two blank lines.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see InfoCode
 */
class CodeParser {
    /**
     * Parses a list of inforamtion codes into a collection
     * of travel plans.
     *
     * @param codeList      the string array of infoarmtion codes
     * @return              the vector of travel plans
     */
    public Vector parse(String[] codeList) {
        Vector travels = new Vector();
        TravelPlan travel = new TravelPlan();
        SubTravelPlan subTravel = new SubTravelPlan();

        for (int i = 0; i < codeList.length; i++) {
            if (codeList[i].trim().equals("")) {
                // Blank line: end subtravel
                travel.addSubTravel(subTravel);
                subTravel = new SubTravelPlan();
                if ((i == (codeList.length - 1)) ||
                    (codeList[i+1].trim().equals(""))) {
                    // Two blank lines or at last line: end travel
                    travels.addElement(travel);
                    travel = new TravelPlan();
                    i++;         // next line already read
                    continue;    // skip to line after next line
                }
            }
```

```
            else {
                // Nonblank line: add to subtravel
                subTravel.addCode(codeList[i].trim());
            }
        }

        return travels;
    }
}
```

```
/*
 * Slots
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;

/**
 * A collection of information slots, where each slot is an
 * information code written as a name/value pair.
 * See class InfoCode for details on conversions
 * between information codes and slots.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see InfoCode
 */
 public class Slots implements Constrainable {

    // Variable
    private Hashtable hash = new Hashtable();

    // Constructors

    /**
     * Constructs an empty slot collection.
     */
    public Slots() {}

    /**
     * Constructs a slot collection initialised from an array of
     * strings, where each string is a name/value pair.
     *
     * @param slotStrings   the string array with name/value pairs
     */
    public Slots(String[] slotStrings) {
        StringTokenizer tok;
        String slotName;

        Debug.println("no of slots: " + slotStrings.length, 9);
        for (int i = 1; i < slotStrings.length; i++) {
            Debug.println("slot: #" + slotStrings[i] + "#", 9);
            tok = new StringTokenizer(slotStrings[i]);
            slotName = tok.nextToken();
            Debug.println("slotname: #" + slotName + "#", 9);
            //      if (!slotName.substring(0, 4).equals("turn")) {
```

```
            // Discard turn info.
        add(slotName, readSlot(tok));
            //          }
    }
}

/**
 * Adds a slot to the hash table of slots, indexed by name.
 *
 * @param key       the slot name, to be used as key
 * @param value     the slot value
 */
public void add(String key, String value) {
    if (nonblank(key) && nonblank(value)) {
        hash.put(key, value);
    }
}

/**
 * Given a slot name, returns the corresponding slot value.
 *
 * @param key       the string slot name
 * @return          the string slot value, or null if no entry
                    matches the given key
 */
public String value(String key) {
    return (String) hash.get(key);
}

/**
 * Given a slot name, returns the corresponding slot value.
 * Same as <code>value</code>, included to implement
 * <code>Constrainable</code>.
 *
 * @param slotName  the string slot name
 * @return          the string slot value, or null if no entry
                    matches the given key
 */
public String lookUp(String slotName) {
    return value(slotName);
}

/**
 * @return          the array of info codes representing the
 *                  slots
 */
public InfoCode[] toInfoCodes() {
    Vector codeVector = new Vector();
    String name;
    InfoCode code;
```

```
        Enumeration enum = hash.keys();
        while (enum.hasMoreElements()) {
            name = (String) enum.nextElement();
            code = new InfoCode(name, value(name));
            if (code.code() != null) {
                codeVector.addElement(code);
            }
        }

        InfoCode[] codes = new InfoCode[codeVector.size()];
        codeVector.copyInto(codes);
        return codes;
    }

    /**
     * @return          a multi-line string representing
     *                  the slots as name/value pairs
     */
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        Enumeration enum = hash.keys();
        String name;
        while (enum.hasMoreElements()) {
            name = (String) enum.nextElement();
            buffer.append(name + " " + value(name) + "\n");
        }
        return buffer.toString();
    }

    /**
     * @param str       a string to be tested
     * @return          false if a string consists of only
     *                  white-space, true otherwise
     */
    private boolean nonblank(String str) {
        return (str != null) && !str.trim().equals("");
    }

    /**
     * @param tok       the strink tokeizer to read from
     * @return          the next token, stripped for trailing
     *                  white-space
     */
    private String readSlot(StringTokenizer tok) {
        if (tok.hasMoreTokens()) {
            return tok.nextToken().trim();
        }
        else {
            return null;
```

```
        }
    }

}
```

```
/*
 * Match
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import COM.stevesoft.pat.Regex;

/**
 * A representation of how good two Slots objects match each other.
 * The constructor takes the two Slots objects to compare.
 * Subscores are calculated for different match criterias.
 * The importance of each subscore  can be adjusted by changing
 * the values of the weight variables. The matching may be made more
 * sophisticated by changing the findScore method.
 *
 * The methods better and worse can be used to compare to Match objects
 * to find the best match.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */
public class Match {
    // Change the initial valuesof these weights to change default behavior
    // High number means high penalty per minute difference
    private int deTAfterWeight  =  10;
    private int deTBeforeWeight = 100;
    private int arTAfterWeight  = 100;
    private int arTBeforeWeight =  10;
    private int durationWeight  =  20;

    private int deTScore;
    private int arTScore;
    private int durationScore;

    private int score;

    private Slots ideal;
    private Slots actual;
    private boolean idealDeTInterval = false;
    private boolean idealArTInterval = false;

    /**
     * Constructs a match.
     *
     * @param actualSlots       the slots to check for match
     *                          against the ideal
```

```
 *                          specifying the ideal to
 *                          match actual Slots against
 * @param idealSlots        the slots formiding the ideal
 */
public Match (Slots actualSlots, Slots idealSlots) {
    ideal = idealSlots;
    actual = actualSlots;
    score = findScore();
    Debug.println("Score: " + score, 9);
}

/**
 * Checks if another match is better than this.
 *
 * @param otherMatch        the match to comapre against
 * @return                  true if the match is better than this one,
 */
public boolean better(Match otherMatch) {
    return this.score < otherMatch.score;
}

/**
 * Checks if  another match is worse than this.
 *
 * @param otherMatch        the match to comapre against
 * @return                  true if the match is worse  than this one,
 */
public boolean worse(Match otherMatch) {
    return this.score > otherMatch.score;
}

/**
 * Calculate a score number for this match.
 *
 * @return                  an integer score
 */
private int findScore() {
    checkRestrictions();

    Debug.println("\n\nScoring plan...", 9);
    Debug.println("  QUERY:", 9);
    Debug.println(ideal.toString(), 9);
    Debug.println("  PLAN:", 9);
    Debug.println(actual.toString(), 9);

    if (!compatible()) {
        return Integer.MAX_VALUE;   // the worst possible score
    }
    findSubScores();
    return deTScore + arTScore + durationScore;
```

```
}

/**
 * Calculate the individual sub scores.
 * There is sub score based on match in departure time,
 * on arrival time, and on duartion of the travel.
 */
private void findSubScores() {
    deTScore = findSubScore(ideal.value("DeT"), actual.value("DeT"),
                            deTBeforeWeight, deTAfterWeight,
                            idealDeTInterval);
    arTScore = findSubScore(ideal.value("ArT"), actual.value("ArT"),
                            arTBeforeWeight, arTAfterWeight,
                            idealArTInterval);
    durationScore =
        timeDifference(actual.value("DeT"), actual.value("ArT"))
        * durationWeight;
    Debug.println("deTScore = " + deTScore, 9);
    Debug.println("arTScore = " + arTScore, 9);
    Debug.println("durScore = " + durationScore, 9);
}

/**
 * Calculate one sub score.
 *
 * @param idealTime     the time value given in query
 * @param actualTime    the time value given in result
 * @param beforeWeight  the weight to use if actualTime is before
 *                      idealtime
 * @param afterWeight   the weight to use if actualTime is after
 *                      idealtime
 * @param interval      true if the query specified a time interval
 * @return              an integer score
 */
private int findSubScore(String idealTime, String actualTime,
                         int beforeWeight, int afterWeight,
                         boolean interval) {
    if (idealTime == null) {
        return 0;
    }
    else {
        if (interval) {
            // idealTime is an interval, i.e. a "Between" time
            String[] idealTimes = splitTime(idealTime);
            int diffDown = timeDifference(idealTimes[0], actualTime);
            int diffUp = timeDifference(actualTime, idealTimes[1]);
            int mid = Math.abs(diffUp - (Math.abs(diffUp - diffDown) / 2));
            Debug.println("Interval. Mid=" + mid +
                          " Down=" + diffDown + " Up=" + diffUp, 9);
            if ((diffDown >= 0) && (diffUp >= 0)) {
```

```
                // Actual time is in the interval
                // Score = difference from middle of interval
                return Math.abs(diffUp - mid);
            }
            else if (diffDown < 0) {
                // Actual time is before the interval
                // Score calculated with beforeWeight
                // Adding mid makes score continous at the interval boundry
                return mid + (-diffDown * beforeWeight);
            }
            else if (diffUp < 0) {
                // Actual time is after the interval
                // Score calculated with afterWeight
                // Adding mid makes score continous at the interval boundry
                return mid + (-diffUp * afterWeight);
            }
        }
        else {
            // idealTime is not an interval
            int diff = timeDifference(idealTime, actualTime);
            if (diff < 0) {
                // Actual time is before ideal time
                return -diff * beforeWeight;
            }
            else {
                // Actual time is after ideal time
                return diff * afterWeight;
            }
        }
    }
    return 0;     // if not returned yet
}

/**
 * Checks which restrictions were given in the query and adjust
 * weigths.
 */
private void checkRestrictions() {
    // Change weight assignments here to change behavior with
    // "Before", "After" and "Between" time values
    String deTSubCode;
    String arTSubCode;

    // Search for a word before first (
    Regex r = new Regex("(\\w+)\\(.*");

    if ((ideal.value("DeT") != null) && r.search(ideal.value("DeT"))) {
        deTSubCode = r.stringMatched(1);
        if (deTSubCode.equals("Before")) {
            deTBeforeWeight =   1;
```

```
            deTAfterWeight  = 100;
        }
        else if (deTSubCode.equals("After")) {
            deTBeforeWeight = 100;
            deTAfterWeight  =   1;
        }
        else if (deTSubCode.equals("Between")) {
            idealDeTInterval = true;
        }
        // else keep default values
    }

    if ((ideal.value("ArT") != null) && r.search(ideal.value("ArT"))) {
        arTSubCode = r.stringMatched(1);
        if (arTSubCode.equals("Before")) {
            arTBeforeWeight =   1;
            arTAfterWeight  = 100;
        }
        else if (arTSubCode.equals("After")) {
            arTBeforeWeight = 100;
            arTAfterWeight  =   1;
        }
        else if (arTSubCode.equals("Between")) {
            idealArTInterval = true;
        }
        // else keep default values
    }
}

/**
 * Checks if the time string given contain two time values.
 *
 * return       string array with two strings, with modifiers like
 *              "absolute" removed, or null, of the string did not
 *              contain two time values
 */
private String[] splitTime (String time) {
    String[] times;
    // Search for two time values. Example: "18:50, 19:10"
    // Return array of two time strings, with modifiers like
    // "absolute" removed.
    Regex r = new Regex
        (".*?\\((\\d+)\\:(\\d+)\\)\\s*\\,.*?\\((\\d+)\\:(\\d+)\\).*");

    if (r.search(time)) {
        times = new String[2];
        times[0] = r.stringMatched(1) + ":" + r.stringMatched(2);
        times[1] = r.stringMatched(3) + ":" + r.stringMatched(4);
        return times;
    }
```

```
    else {
        Debug.println("Warning: Illegal time format " + time, 9);
        return null; // not found
    }
}

/**
 * Checks whether the ideal and the actual values are compatible,
 * i.e. have the same departure place, arrival place and day.
 *
 * @return      true if ideal and actual are compatible
 */
private boolean compatible() {
    boolean rightDeP = actual.value("DeP").equals(ideal.value("DeP"));
    boolean rightArP = actual.value("ArP").equals(ideal.value("ArP"));
    boolean rightDay =
        ((actual.value("Day") == null) ||
         actual.value("DeP").equals(ideal.value("DeP")));
    Debug.println("Right? " + rightDeP + rightArP + rightDay, 9);
    return rightDeP && rightArP && rightDay;
}

/**
 * @param time1     a string time value
 * @param time2     a string time value
 * @return          the differnece between time and time2, in minutes
 */
private int timeDifference(String time1, String time2) {
    return minutes(time2) - minutes(time1);
}

/**
 * Calculate number of minutes from midnight for a given time value.
 *
 * @param time      the time value
 * @return          an integer number of minutes since midnight
 */

private int minutes(String time) {
    if (time != null) {
        Regex r = new Regex("(\\d+)\\:(\\d+)");
        if (r.search(time)) {
            int hours = Integer.parseInt(r.stringMatched(1));
            int mins  = Integer.parseInt(r.stringMatched(2));
            return hours*60 + mins;
        }
    }
    Debug.println("Warning: Illegal time expression " + time, 5);
    return 0;
}
```

```
}
```

```
/*
 * Constraints
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;

/**
 * A collection of constraints that can be checked against
 * Constrainable objects. The constraints are based on slots
 * (name/value pairs). The Constrainable interface specifies
 * a look-up mechanism for slots.
 *
 * The collection consists of 3 parts:
 * <ol> Slots that must exist, i.e. have non-blank values
 * <li> Slots that must have a certain value
 * <li> Slots that must have the same value as some givenmother slots
 * </ol>
 *
 * The class offer functionality to construct new Constraints from
 * a string represenation (typically part of a template file), and
 * for checking if Constrainable objects like travel plans
 * fullfills all constraints.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Conttrainable
 */
public class Constraints {

    // Variables
    Vector mustExist = new Vector();
    Hashtable mustHaveValue = new Hashtable();
    Hashtable mustEqual = new Hashtable();

    /**
     * Reads a string and adds any contstraints found to the
     * collection.
     *
     * @param constraintString  the string to read constraints from
     */
    public void parseAndAdd(String constraintString) {
        StringTokenizer tok = new StringTokenizer(constraintString, ",");
        Regex eq = new Regex("(.*)=(.*)");
```

```
        Regex number = new Regex("(\\d+)");
        Regex string = new Regex("\"(.*)\"");
        Regex slotName = new Regex("(\\w+)");
        String constraint;
        String slot;
        String value;

        while (tok.hasMoreTokens()) {
            constraint = tok.nextToken().trim();
            if (eq.search(constraint)) {
                // Constraint of form "Slot = Value"
                Debug.println("Eq constr: " + constraint, 9);
                slot = eq.stringMatched(1).trim();
                value = eq.stringMatched(2).trim();
                if (number.matchAt(value, 0)) {
                    // Value is a number - use as is
                    addValue(slot, value);
                } else if (string.matchAt(value, 0)) {
                    // Value is a quoted string - remove quotes
                    addValue(slot, string.stringMatched(1));
                } else if (slotName.matchAt(value, 0)) {
                    // Value is a slot name
                    addEqual(slot, slotName.stringMatched(1));
                } else {
                    System.err.println("Syntax error in template: "
                                       + constraintString);
                }
            }
            else {
                // Constraint of form "Slot"
                Debug.println("Non-eq constr: " + constraint, 9);
                addExist(constraint);
            }
        }
    }

    /**
     * Test if a contstarinable object fullfills these constraints.
     *
     * @param candidate     a constrainble object to check
     * @return              true if the object passed, false if not
     */
    public boolean test(Constrainable candidate) {
        // candidate is either a travel plan or a query
        Enumeration enum;
        String name;
        String value;
        String slot1, slot2;
        String value1, value2;
```

```
// Check that all slots specified in mustExist has values in candidate
enum = mustExist.elements();
while (enum.hasMoreElements()) {
    name = (String) enum.nextElement();
    if (candidate.lookUp(name) == null) {
        Debug.println("template mismatch: no " + name, 9);
        return false;
    }
}

// Check that all pairs of slots in mustEqual have equal values
enum = mustEqual.keys();
while (enum.hasMoreElements()) {
    slot1 = (String) enum.nextElement();
    value1 = candidate.lookUp(slot1);
    slot2 = (String) mustEqual.get(slot1);
    value2 = candidate.lookUp(slot2);
    if (value1 == null) {
        // No such slot in candidate
        Debug.println("template mismatch: no " + slot1
                      + " to equal " + slot2, 9);
        return false;
    }
    else if (value2 == null) {
        // No such slot in candidate
        Debug.println("template mismatch: no " + slot2
                      + " to equal " + slot1, 9);
        return false;
    }
    else {
        boolean equal = value1.equals(value2);
        if (!equal) {
            Debug.println("template mismatch: " + slot1
                          + " and " + slot2 + " not equal", 9);
        }
        return equal;
    }
}

// Check the values of all slots specified in mustHaveValue
enum = mustHaveValue.keys();
while (enum.hasMoreElements()) {
    name = (String) enum.nextElement();
    value = (String) mustHaveValue.get(name);
    if (candidate.lookUp(name) == null) {
        // No such slot in candidate
        Debug.println("template mismatch: no " + name
                      + " to have value " + value, 9);
        return false;
    }
```

```
            else if (!candidate.lookUp(name).equals(value)) {
                // Slot has wrong value in candidate
                Debug.println("template mismatch: " + name + " is not "
                              + value, 9);
                return false;
            }
        }

        // If not returned by now, the candidate passed all tests
        Debug.println("template match", 9);
        return true;
    }

    private void addExist(String slotName) {
        mustExist.addElement(slotName);
    }

    private void addValue(String slotName, String value) {
        mustHaveValue.put(slotName, value);
    }

    private void addEqual(String slot1, String slot2) {
        mustEqual.put(slot1, slot2);
    }

}
```

```
/*
 * Constrainable
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;

/**
 * Any object that can be constrained using
 * <code>Constraints</code> objects.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Constraint
 */
public interface Constrainable {
    String lookUp(String slotName);
}
```

```
/*
 * Presentation
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import java.io.*;

/**
 * A represenation of how to present a travel plan to a customer.
 * A presentation consists of a collection of presentation chunks.
 * It keeps a reference to the travel plan it is presentating and
 * the query that lead to the presentation. The class keeps a static
 * vector of presentation templates, initialised by reading template
 * files atat start-up. When a new object is created,
 * this vector is searched for a matching template. If one is found,
 * the object keeps a reference to that template and use it for
 * calculating the presentation chunks.
 *
 * If no template is found,
 * presenatation chunks are calculated with
 * <code>generateDefault</code>. This method separates given information
 * from new information makes a choice of which information is
 * important enough to be included in the presenation. However,
 * the current implementation of <code>generateDefault</code> is crude
 * and exploites this functionality badly.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see PresentationChunk
 * @see PresentationTemplate
 */
public class Presentation {

    // Class variables
    private static final String TEMPLATE_DIR =
        "/home/thomaso/java/alparon/querytravel/templates";
    private static Vector templates;

    // Member variables
    private Query query;
    private TravelPlan plan;
    private PresentationTemplate template;
    private Vector chunks;

    // Constructor
```

```
/**
 * Constructs a new presenation of a plan.
 *
 * @param query     the query that the presenation will answer
 * @param plan      the travel plan to be presented
 */
public Presentation(Query query, TravelPlan plan) {
    this.query = query;
    this.plan = plan;
    template = findTemplate();
    if (template == null) {
        Debug.println("generating default presentation...", 9);
        chunks = generateDefault();
    }
    else {
        Debug.println("\napplying template...", 9);
        chunks = template.apply(plan);
    }
}

/**
 * Store this presentation on the Travel Board. A connection
 * is needed for sending the message. Ususally the Query
 * Travel module will give a reference to itself as
 * connection, so the Query Travel socket can be used.
 *
 * @param connection    a connection to send the message on
 */
public void store(Connection connection) {
    connection.send("TRAVELBOARD", "StorePresentation", toStringArray());
}

/**
 * Return a representation of the presenation as a string array.
 * Each string represents one utterance and contains a
 * list of infoarmtion codes to be included in the utternce.
 *
 * @return              the array of strings listing the inforamtion
 *                      to be included in each utterance
 */
public String[] toStringArray() {
    Enumeration enum = chunks.elements();
    Vector strings = new Vector();
    String [] chunkStrings;

    while (enum.hasMoreElements()) {
        chunkStrings =
            ((PresentationChunk) enum.nextElement()).toStringArray();
        for (int i = 0; i < chunkStrings.length; i++) {
```

```
            strings.addElement(chunkStrings[i]);
        }
        // Add blank line to separate chunks
        strings.addElement("");
    }

    String[] stringArray = new String[strings.size()];
    strings.copyInto(stringArray);
    return stringArray;
}

/** Returns a string represenation of the list. This is done by
 * listing all information to be presented, without indicating
 * borders betwwen utterances.
 *
 * @return      a string containing all information
 *              conatined in this presenttion
 */
public String toString() {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < chunks.size(); i++) {
        buf.append(chunks.elementAt(i).toString() + "\n");
    }
    return buf.toString();
}

/**
 * Finds a presnetation template that matches the travel plan
 * and query associated with this presentation.
 *
 * @return      a presentation template that matches, or null if
 *               no template was found
 */
private PresentationTemplate findTemplate() {
    PresentationTemplate thisTemplate;
    Enumeration enum = templates.elements();
    while (enum.hasMoreElements()) {
        thisTemplate = (PresentationTemplate) enum.nextElement();
        if (thisTemplate.match(query, plan)) {
            // Found a matching template - stop looking and return it
            return thisTemplate;
        }
    }
    // Looped through all templates without finding a match
    return null;
}

/**
 * Generates a defualt presentation. This should be done
 * for the cases where no template was found.
```

```
 *
 * @return      a vector of presentation chunks
 */
private Vector generateDefault() {
    Hashtable codes = new Hashtable();
    Vector chunks = new Vector();
    InfoCode[] givenCodes = query.slots().toInfoCodes();

    // Collect all important codes, initialised as "new"
    for (int i = 0; i < plan.subTravels().length; i++) {
        addSubTravelCodes(codes, plan.subTravels()[i]);
    }
    // Mark codes allready given in the query as "given"
    for (int i = 0; i < givenCodes.length; i++) {
        if (codes.contains(givenCodes[i])) {
            codes.put(givenCodes[i], "given");
        }
    }

    // Noe choose a crude strategy:
    // Chunk 3 and 3 codes together, and add them to chunks
    PresentationChunk chunk = new PresentationChunk();
    Enumeration enum = codes.keys();
    int n = 0;
    while (enum.hasMoreElements()) {
        if (n == 3) {
            chunks.addElement(chunk);
            chunk = new PresentationChunk();
            n = 0;
        }
        chunk.addCode((InfoCode) enum.nextElement());
        n++;
    }
    chunks.addElement(chunk);
    return chunks;
}

/**
 * Adds the information codes from one sub travel  to the hash
 * table of codes. Only codes considered important by
 * <code>importantCode</code> are added. All codes
 * are marked as being "new".
 *
 * @param codes     the hash table to add codes to
 * @param subTravel the sub travel plan to read codes from
 */
private void addSubTravelCodes(Hashtable codes, SubTravelPlan subTravel) {
    InfoCode code;
    Enumeration enum = subTravel.codes().elements();
    while (enum.hasMoreElements()) {
```

```
            code = (InfoCode) enum.nextElement();
            if (importantCode(code)) {
                codes.put(code, "new");
            }
        }
        return;
    }

    /**
     * Checks whether an information code is important,
     * that is if it is a code for location or time.
     *
     * @param code      the info code to be checked
     * @return          true if the code contains information
     *                  on location or time, false otherwise
     */
    private boolean importantCode(InfoCode code) {
        String codeString = code.toString();
        return ((codeString.substring(0, 4).equals("Loc(")) ||
                (codeString.substring(0, 3).equals("Tm(")));
    }

    {
        // Initialising templates (shared by all objects)
        Debug.println("Initialising templates", 9);
        File dir = new File(TEMPLATE_DIR);
        File templateFile;
        Vector fileNames = new Vector();

        /* Add template files to file names in prioritised order.
         * First matching template will be used, so keep specific
         * templates before general ones.
         */
        fileNames.addElement("direct_DeT.tpl");
        fileNames.addElement("direct_ArT.tpl");
        fileNames.addElement("1change_DeT.tpl");
        fileNames.addElement("1change_ArT.tpl");
        Debug.println(fileNames.size() + " template files", 9);

        // Fill templates vector by reading the specified file
        templates = new Vector();
        for (int i = 0; i < fileNames.size(); i++) {
            templateFile = new File(dir, (String) fileNames.elementAt(i));
            templates.addElement(new PresentationTemplate(templateFile));
            Debug.println(fileNames.elementAt(i) + " added\n", 9);
        }
    }

}
```

```
/*
 * PresentationChunk
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;

/**
 * A list of information codes suitabkle for giving contents to
 * one system utterance.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Presentation
 */
public class PresentationChunk {
    // Vector containing InfoCodes for this chunk
    private Vector codes = new Vector();

    /**
     * Adds an information code to the end of this chunk.
     *
     * @param code      the inforamtion code to add
     */
    public void addCode(InfoCode code) {
        codes.addElement(code);
    }

    /**
     * @return          the array of information codes of this chunk
     */
    public InfoCode[] codes() {
        InfoCode[] codeArray = new InfoCode[codes.size()];
        codes.copyInto(codeArray);
        return codeArray;
    }

    /**
     * Returns a representation of the chunk as an array of strings.
     * Each string gives one information code.
     *
     * @return      the string array with the inforamtion codes
     */
    public String[] toStringArray() {
        if (codes == null) {
            Debug.println("Warning: null PresentationChunk", 5);
```

```
            return null;
        }
        else {
            String[] strings = new String[codes.size()];
            for (int i=0; i < codes.size(); i++) {
                strings[i] = codes.elementAt(i).toString();
            }
            return strings;
        }
    }

}
```

```
/*
 * PresentationTemplate
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.querytravel;
import alparon.lib.*;
import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;

/**
 * A description of how a travel plan should be presented in
 * a given situation, where the situation ios specified as
 * lists of constraints on the query to be answered and on
 * the travel plan.
 *
 * The constructor expects a file handle and will read the
 * template from file. The template has functionality of
 * checking whether it fits in a certain situation and
 * of generating a list of presentation chunks for a
 * travel plan by "applying itself".
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Presentation
 * @see PresentationChunk
 */
public class PresentationTemplate {

    // Variables
    private String name;
    private Constraints queryConstraints;
    private Constraints resultsConstraints;
    private String[] utterances;
    private PresentationChunk[] pattern;

    // Constructor

    /**
     * Constructs a presentation template by reading it from file
     *
     * @param file      the file to read from
     */
    public PresentationTemplate(File file) {
        name = file.getName();
        readFromFile(file);
    }
```

```
/**
 * Check whether this template is applicable for the
 * given situation.
 *
 * @param query     the query to be answered
 * @param plan      the travel plan to present
 * @return          true if this template can be used,
 *                  false otherwise
 */
public boolean match(Query query, TravelPlan plan) {
    Debug.println("Checking query against template " + name, 9);
    boolean queryOK = queryConstraints.test(query.slots());
    Debug.println("Checking results against template...", 9);
    boolean resultsOK = resultsConstraints.test(plan);
    boolean templateOK = queryOK && resultsOK;
    if (templateOK) {
        Debug.println("Using template " + name, 9);
    }
    return templateOK;
}

/**
 * Generates a list of presentation chunks that follows the
 * pattern specified by this template.
 *
 * @param plan      the travel plan to be presented
 * @return          a vector of presentation chunks
 */
public Vector apply(TravelPlan plan) {
    Vector appliedChunks = new Vector();
    PresentationChunk chunk;
    InfoCode[] patternCodes;
    InfoCode actualCode;

    for (int i = 0; i < pattern.length; i++) {
        // loop for each chunk in the pattern, i.e. each utterance
        chunk = new PresentationChunk();
        patternCodes = pattern[i].codes();
        Debug.println("patternCodes length: " + patternCodes.length, 9);

        for (int j = 0; j < patternCodes.length; j++) {
            // loop for each InfoCode in the chunk
            actualCode = plan.lookUpCode(patternCodes[j].slotName());
            Debug.println("Using " + actualCode.code() + " for "
                          + patternCodes[j].slotName(), 9);
            chunk.addCode(actualCode);
        }
        appliedChunks.addElement(chunk);
    }
```

```
    return appliedChunks;
}

/**
 * Reads a template from file and handles exceptions that may
 * arise.
 *
 * @param file      the file to read from
 */
private void readFromFile(File file) {
    try {
        BufferedReader in = new BufferedReader(new FileReader(file));
        String line;
        StringTokenizer tok;
        String constraint;
        Vector utteranceVector = new Vector();

        line = nextSignificantLine(in);

        // Read constraints on query
        queryConstraints = new Constraints();
        while (!blank(line)) {
            if (!comment(line)) {
                Debug.println("Query: #" + line + "#", 9);
                queryConstraints.parseAndAdd(line);
            }
            line = in.readLine();
        }

        // Discard separator line(s)
        line = nextSignificantLine(in);

        // Read constraints on results
        resultsConstraints = new Constraints();
        while (!blank(line)) {
            if (!comment(line)) {
                Debug.println("Results: #" + line + "#", 9);
                resultsConstraints.parseAndAdd(line);
            }
            line = in.readLine();
        }

        // Discard separator line(s)
        line = nextSignificantLine(in);

        // Read template
        while ((line != null) && !blank(line)) {
            if (!comment(line)) {
                Debug.println("Template: #" + line + "#", 9);
```

```
                utteranceVector.addElement(line);
            }
            line = in.readLine();
        }
        utterances = new String[utteranceVector.size()];
        utteranceVector.copyInto(utterances);

        in.close();

        pattern = findPattern(utterances);
    }
    catch (FileNotFoundException e) {
        System.err.println("Warning: Presentation template file "
                           + file.getName() + " not found.");
    }
    catch (IOException e) {
        System.err.println("Warning: Error reading template file "
                           + file.getName() + ".");
    }
}

/**
 * Tests if a string consist of only white-space.
 *
 * @param line      the striong to be tested
 * @return          true if the string is only white-space
 */
private boolean blank(String line) {
    return line.trim().equals("");
}

/**
 * Tests if a string is a comment, that is, starts with "#".
 *
 * @param line      the striong to be tested
 * @return          true if the string is a comment
 */
private boolean comment(String line) {
    Regex r = new Regex("^#");
    return r.search(line);
}

/**
 * Returns teh next non-blank, non-comment line.
 *
 * @param in       a buffered reader to read from
 * @exception IOException if end of file is reached or any
 *                 other problem arises when reading.
 */
private String nextSignificantLine(BufferedReader in) throws IOException {
```

```
    // Discard blank lines and comments
    String line = in.readLine();
    while (blank(line) || comment(line)) {
        line = in.readLine();
    }
    return line;
}

/**
 * Generates presentation chunks by extracting slot names from
 * a list of strings. Each string corresponds to one
 * presentation chunk. Slot names are any string enclosed in
 * square brackets. Example: "[DeP]".
 *
 * @param strings   the array of strings to examine
 * @return          an array of presentation chunks
 */
private PresentationChunk[] findPattern(String[] strings) {
    PresentationChunk[] foundChunks =
        new PresentationChunk[strings.length];;
    for (int i = 0; i < strings.length; i++) {
        foundChunks[i] = findAllCodes(strings[i]);
    }
    return foundChunks;
}

/**
 * Generates one presentation chunks by extracting slot names from
 * a string. Slot names are any string enclosed in
 * square brackets. Example: "[DeP]".
 *
 * @param string    the string to examine
 */
private PresentationChunk findAllCodes(String string) {
    PresentationChunk chunk = new PresentationChunk();
    Regex code = new Regex("\\[(.*?)\\]");
    int pos = 0;
    Debug.println("Checking for codes in " + string, 9);
    while ((pos >= 0) && code.searchFrom(string, pos)) {
        Debug.println("Found " + code.stringMatched(1), 9);
        chunk.addCode(new InfoCode(code.stringMatched(1)));
        pos = code.matchedTo();
    }
    return chunk;
}

/**
 * Unit testing routine.
 * Run this by running <code>java PresentationTemplate</code>.
 *
```

```
     * @param args              not used
     * @exception Exception     if problems reading files etc
     */
    public static void main(String[] args) throws Exception {
        Debug.level(9);
        File file = new File("template1.txt");
        PresentationTemplate templ = new PresentationTemplate(file);
    }

}
```

Appendix F

Java Framework Code Listing

```
/*
 * MessagingModule
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.util.*;
import java.io.*;

/**
 * An Alparon module that can send and receive messages to other
 * Alparon modules. This class should be inherited by all modules.
 * It then offers these services:
 * <ul>
 * <li>Sets up sockets for sending and receiving messages
 * <li>Hands incoming messages over to the right message handler
 *     (see class MessageHandler)
 * <li>Sends messages, given message name, content and receiver
 * <li>Sends replies, given message name and content
 * <li>Loops waiting for incoming messages
 * <p>
 *
 * How to make a new Alparon module:
 * <ul>
 * <li>Implement module functionlity in a class that extends
 *     <code>MessagingModule</code>.
 * <li>For each message the module should be able to receive, i
 *     implement a class that extends MessageHandler.
 *     These message handlers should call methods of the module,
 *     hereby offering the module functionality through the
```

```
 *     messages.
 * <li>The module's constructor should first call
 *     <code>initialise</code>, then <code>addHandler</code>
 *     for ech message handler.
 * </ul>
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */

public class MessagingModule implements Connection {

    // Class variable
    static protected Hashtable msgHandlers = new Hashtable();

    // Instance variables
    protected Server server;
    protected Settings settings;

    // Accesors

    /**
     * @return      the settings of this module, as generated on
     *              construction
     */
    public Settings settings() {
        return settings;
    }

    /**
     * @return      the server this module uses for sending and
     *              receiving messages
     */
    public Server server() {
        return server;
    }

    // Public methods

    /**
     * Initialises the module by generating settings, setting up
     * a server and registrering standard messages "stop" and
     * "addMessage". This makes every messaging module capable
     * of handling "stop" and "addMessage" messages.
     * Needs the module object to decide which child
     * class it is a member of, to set the name of the module.
     *
     * @param module    the messaging module object of some type
     *                  derived from MessagingModule;
     * @param pid       the integer process id of the process that
```

```
 *                  created the object
 */
public void initialise(MessagingModule module, int pid) {
    settings = generateSettings(module, pid);
    server = new Server(settings);
    addHandler("AddMessage", new AddMessageHandler(this));
    addHandler("Stop", new StopHandler(this));
}

/**
 * Registrers a message handler so allow this module to accept
 * a new type of message.
 *
 * @param msgName   the string name of the message to be accepted
 * @param handler   the message handler that messages of the new
 *                  type should be sent to
 */
public void addHandler(String msgName, MessageHandler handler) {
    msgHandlers.put(msgName, handler);
}

/**
 * Sends a message, given the <code>Message</code> object.
 *
 * @param msg       the message to send
 */
public void send(Message msg) {
    Debug.println(msg.toString(), 5);
    server.sendMessage(msg);
}

/**
 * Sends a message, with message name as part of the body
 * parameter. First line in the body is the message name.
 *
 * @param receiver  the string name (GMN) of the receiver
 * @param body      the string array body of the message
 */
public void send(String receiver, String[] body) {
    Message msg =
        new Message(settings.imn(), receiver, body, settings.pid());
    send(msg);
}

/**
 * Sends a message, with message name as separate parameter.
 *
 * @param receiver  the string name (GMN) of the receiver
 * @param msgName   the string message name
 * @param body      the string array contents of the message
```

```
     */
    public void send(String receiver, String msgName, String[] contents) {
        String[] body = new String[contents.length + 1];
        body[0] = msgName;
        System.arraycopy(contents, 0, body, 1, contents.length);
        send(receiver, body);
    }

    /**
     * Receives a message.
     *
     * @return          the message object
     */
    public Message receive() {
        Message msg = server.receiveMessage();
        Debug.println(msg.toString(), 4);
        return msg;
    }

    /**
     * Receives a message with a given name. If another message
     * arrive, discard it and wait for next message.
     *
     * @param name      the string name of the message to receive
     * @return          the message object
     */
    public Message receive(String name) {
        Message msg = null;
        boolean rightMessage = false;
        while (!rightMessage) {
            msg = receive();
            // If message has wrong name, discard it
            rightMessage = msg.body()[0].equals(name);
            if (!rightMessage) {
                System.err.println("Warning: Discarded " + msg.body()[0]
                                   + " message since " + name
                                   + " message was expected.");
            }
        }
        return msg;
    }

    /**
     * Sends a reply, with reply name as separate parameter and
     * contents as a single string.
     *
     * @param msg       the message to reply to
     * @param replyName the string name of the reply message
     * @param contents  the string contents of the reply
     *                  message
```

```
 */
public void reply(Message msg, String replyName, String contents) {
    String[] body = new String[] {replyName, contents};
    reply(msg, body);
}

/**
 * Sends a reply, with reply name as separate parameter and
 * contents as an array of strings.
 *
 * @param msg       the message to reply to
 * @param replyName the string name of the reply message
 * @param contents  the string array contents of the reply
 *                  message
 *
 */
public void reply(Message msg, String replyName, String[] contents) {
    String[] body = new String[contents.length + 1];
    body[0] = replyName;
    System.arraycopy(contents, 0, body, 1, contents.length);
    reply(msg, body);
}

/**
 * Sends a reply, with reply name as part of the body parameter.
 *
 * @param msg       the message to reply to
 * @param body      the string array body the reply message
 */
public void reply(Message msg, String[] body) {
    Message replyMsg =
        new Message(settings.imn(), msg.source(), body, settings.pid());
    replyMsg.messageID(msg.messageID());
    send(replyMsg);
}

/**
 * Listens to incoming messages, sending them to the
 * appropriate message handler. Loops until the server
 * shuts down and hence blocks the thread.
 *
 * @exception UnknownMessageException   if a message for which
 *                  this module has no handler is received
 * @exception MalformedMessageException if a message with incorrect
 *                  syntax is received
 */
public void listen()
    throws UnknownMessageException, MalformedMessageException {

    Message receivedMsg;
```

```
        String msgName;
        String className;
        MessageHandler msgHandler;

        // Loops until the server is shut down
        while (server != null) {
            receivedMsg = receive();
            receivedMsg.display();
            msgName = (receivedMsg.body())[0];
            //      if (msgName.equals("Stop")) {
            //          break;
            //      }
            System.out.println("Received " + msgName);

            msgHandler = (MessageHandler) msgHandlers.get(msgName);
            if (msgHandler == null) {
                throw new UnknownMessageException(msgName);
            }
            msgHandler.handleMessage(receivedMsg);
        }
    }

    /**
     * Shuts down the server and releases the socket.
     */
    public void stop() {
        try {
            if (server != null) {
                server.stop();
                Debug.println(settings.gmn() + " stopped.", 2);
            }
            server = null;
        }
        catch (IOException e) {
            System.err.println("Error stopping " + settings.gmn());
            System.err.println(e.getMessage());
        }
    }

    /**
     * Shuts down the server once the object is no longer
     * referenced.
     *
     * @exception Throwable     if problem shutting down sever or
     *                          in super.finalize()
     */
    protected void finalize() throws Throwable {
        server.stop();
        super.finalize();
    }
```

```
    /**
     * Generates settings for this module by analysing its full
     * class name.
     *
     * @param module    the messaging module object to generate
     *                  settings for
     * @param pid       the process id of the process that created
     *                  the object
     * @return          the settings object generated
     */
    private Settings generateSettings(MessagingModule module, int pid) {
        String className = module.getClass().getName();
        String pkg;
        String name;
        int pkgEnd = className.lastIndexOf(".");
        if (pkgEnd == -1) {
            // no package name
            pkg = "";
        }
        else {
            pkg = className.substring(0, pkgEnd);
        }
        name = className.substring(pkgEnd+1).toUpperCase();
        return new Settings(name, pkg, pid);
    }

}
```

```
/*
 * Message
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.lang.*;
import java.util.*;
import java.io.*;

/**
  * A message sent between any modules that are registered with a Alparon
  * portmapper.
  * The text package transferred consists of the following
  * fields:
  * <ul>
  * <li> destination module name (GMN or IMN)
  * <li> source module name (IMN since sender always knows this)
  * <li> message ID (an integer that is unique except for reply messages
  *                  who have the same ID as the message they are replying to)
  * <li> body (first line is usually message name, the rest could be anything)
  * </ul>
  * GMN means "Generic Module Name" and refers to a type of module.
  * IMN means "Instantiated Module Name" and refer to a unique instance of
  * a generic module type (see class Settings for details).
  * <p>
  * A message object contains all this data and has operations to
  * send and receive it.
  * To send a message, initialize it with all its data with
  * <code>Message(String, String, String[], int)</code> and invoke its
  * <code>send</code> method.
  * To receive a message, initialize it to an empty
  * message with <code>Message(int)</code> and invoke its <code>receive</code>
  * method. If the message contains any data when the
  * <code>receive</code> method is invoked, this is overwritten by the received
  * data. The constructors need the process ID number of the process creating
  * the object to make a unique message ID.
  * <p>
  * The pid is unfortunately nesessary as a parameter to the constructors
  * since Java does not have a method to retrieve the current pid
  * (this is considered to be too plattform specific to be supported by
  * Java). In UNIX, give the pid as command-line parameter by typing
  * 'java MyClass $$'.
  *
  * @author Thomas Oldervoll
  * @version $Id$
  *
  * @see Settings
  */
```

```
public class Message {
    /*
     * Ported from the Perl module Message.pm
     */

    // Constants
    private static final int INITIAL_BODY_LENGTH = 10;
    private static final int MAX_MESSAGE_ID = 1000000;

    // Class variables
    private static int nextMessageID;
    private static boolean initialized = false;

    // Instance variables
    private int messageID;
    private String source;
    private String destination;
    private String[] body;
    private Date timestamp;

    // Constructors

    /**
     * @param pid           the integer process ID for the process creating the
     *                          the object;
     */
    public Message(int pid) {
        messageID = newMessageID(pid);
        return;
    }

    /**
     * @param source    the name of the sending module (IMN)
     * @param destination  the name of the receiving module (GMN or IMN)
     * @param body      the text contents of the message
     * @param pid       the integer process ID of the process creating the
     *                  object
     */
    public Message(String source, String destination, String[] body, int pid) {
        messageID = newMessageID(pid);
        this.source = source;
        this.destination = destination;
        this.body = body;
        return;
    }

    // Accessors

    /**
     * @param messageID an integer to identify the messages and connect replies
```

```
     */
    public void messageID(int messageID) {
        this.messageID = messageID;
        return;
    }

    /**
     * @param source    the string name of the sending module (IMN)
     */
    public void source(String source) {
        this.source = source;
        return;
    }

    /**
     * @param destination  the string name of the receiving module (GMN or IMN)
     */
    public void destination(String destination) {
        this.destination = destination;
        return;
    }

    /**
     * @param body      the text contents of the message
     */
    public void body(String[] body) {
        this.body = body;
        return;
    }

    /**
     * @param time      the integer time the message was sent or reiceived
     *                  (milliseconds since January 1, 1970, 00:00:00 GMT)
     */
    public void timestamp(int time) {
        timestamp.setTime(time);
        return;
    }

    /**
     * @return          an integer to identify the messages and connect replies
     */
    public int messageID() {
        return messageID;
    }

    /**
     * @return          the string name of the sending module (IMN)
     */
    public String source() {
```

```
    return source;
}

/**
 * @return          the string name of the receiving module (GMN or IMN)
 */
public String destination() {
    return destination;
}

/**
 * @return          the text contents of the message
 */
public String[] body() {
    return body;
}

/**
 * @return          the integer time the message was sent or reiceived
 *                  (milliseconds since January 1, 1970, 00:00:00 GMT)
 */
public long timestamp() {
    return timestamp.getTime();
}

// Public methods

/**
 * Sends message data to System.out
 */
public void send() {
    // No OutputStream given - use System.out as default
    this.send(System.out);
}

/**
 * Sends message data to the specified stream
 *
 * @param outstream an outputstream to send to
 */
public void send(OutputStream outstream) {
    // Transform OutputStream into PrintWriter for conveniency
    PrintWriter out =
        new PrintWriter(new BufferedOutputStream(outstream), true);

    // create timestamp
    timestamp = new Date();

    // write message fields to the created stream
    out.println(destination);
```

```
        out.println(source);
        out.println(messageID);
        for (int i = 0; i < body.length; i++) {
            out.println(body[i]);
        }
    }

    /**
     * Reads message data from System.in
     */
    public void receive() {
        // No InputStream given - use System.in as default
        this.receive(System.in);
    }

    /**
     * Reads message data from the specified stream
     *
     * @param instream  an inputstream to read from
     */
    public void receive(InputStream instream) {
        // Transform InputStream into BufferedReader for conveniency
        BufferedReader in =
            new BufferedReader(new InputStreamReader(instream));
        Vector bodyVector = new Vector(INITIAL_BODY_LENGTH);
        String nextLine;

        // reset message
        this.clear();

        // create timestamp
        timestamp = new Date();

        try {
            // read destination, source, messageID
            destination = in.readLine();
            source = in.readLine();
            try {
                messageID = Integer.parseInt(in.readLine());
            }
            catch (NumberFormatException e) {
                System.err.println("Illegal message ID: " + e.getMessage());
            }

            // read body, i.e. the rest of the message
            while ((nextLine = in.readLine()) != null) {
                bodyVector.addElement(nextLine);
            }
            body = new String[bodyVector.size()];
            bodyVector.copyInto(body);
```

```
    }
    catch (IOException e) {
        System.err.println("Error reading message: " + e.getMessage());
    }
}

/**
 * Write message data to System.out
 */
public void display() {
    // No OutputStream given - use System.out as default
    this.display(System.out);
}

/**
 * Write message data to the specified stream
 *
 * @param outstream an outputstream to write to
 */
public void display(OutputStream outstream) {
    // Transform OutputStream into PrintWriter for conveniency
    PrintWriter out =
        new PrintWriter(new BufferedOutputStream(outstream),true);
    out.println(this);
    return;
}

/**
 * @return          a string representation of the message data
 */
public String toString() {
    StringBuffer text = new StringBuffer("To:\t" + destination + "\n" +
                                         "From:\t" + source + "\n" +
                                         "MID:\t" + messageID + "\n");
    if (timestamp != null) {
        text.append("Time:\t" + timestamp.getTime() + "\n");
    }
    // Printing body with each line tab'ed
    text.append("Body:\t" + body[0] + "\n");
    for (int i=1; i < body.length; i++) {
        text.append("\t" + body[i] + "\n");
    }
    return text.toString();
}

// Private mehods
/**
 * Resets all message data
 */
private void clear() {
```

```
        messageID = -1;
        source = null;
        destination = null;
        body = null;
        timestamp = null;
        return;
    }

    /**
     * Produces an integer that the instance can use as an unique message ID.
     * If this is the first instance of this class, a random value based
     * on the process ID is created. Otherwise, the message ID used by the last
     * instance increased with one is produced.
     *
     * @param pid       the integer process ID of the process creating
     * @return          an integer not used as message ID by any instance

     */
    private int newMessageID(int pid) {
        if (!initialized) {
            // make random initial value of nextMessageID based on pid
            int randomPart = (int)(Math.floor(Math.random()*1000));
            int pidPart = pid % 1000;
            nextMessageID = (pidPart*1000) + randomPart;
            initialized = true;
        }

        int mid = nextMessageID;
        nextMessageID++;
        if (nextMessageID > MAX_MESSAGE_ID) {
            nextMessageID =- MAX_MESSAGE_ID;
        }
        return mid;
    }

    /**
     * Unit testing routine.
     * Run this by running <code>java Message</code>.
     *
     * @param args      an integer process ID (optional)
     */
    public static void main(String[] args) {
        int pid = ((args.length == 0) ? 42 : Integer.parseInt(args[0]));
        System.out.println("pid = " + pid);
        Message mes1;
        String[] body = {"body"};
        mes1 = new Message("from", "to", body, pid);
        mes1.display();
        mes1.send();
    }
```

```
}
```

```
/*
 * Server
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.net.*;
import java.io.*;

/**
  * Server for sending and receiving messages between Alparon modules via
  * sockets. Every module creates its own instance of Server. To send a
  * message, invoke the <code>sendMessage</code> with a message as an
  * argument. To receive a message, invoke the <code>receiveMessage</code>
  * method, which returns a message. If something goes wrong, the error
  * message is available through the <code>errorMessage</code> method.
  * <p>
  * The server opens up a socket on a specific port. The port number can
  * either chosen randomly or be specified in the constructor. For instance
  * can a server for a portmapper be started like this:
  * <p><code>
  * Server pmServer = new Server(pmSettings, Server.PORTMAPPER_PORT)
  * </code>
  *
  * @author Thomas Oldervoll
  * @version $Id$
  */
public class Server {
    /*
     * Ported from the Perl module Server.pm
     */

    // Constants
    private static final int PORTMAPPER_PORT = 21000;
    private static final int PORT_BASE = 21001;
    private static final int PORT_RANGE = 1000;
    private static final int MAX_QUEUE = 50;
    private static final String PORTMAPPER_NAME = "PM";
    private static final String START_MSG = "IMNstart";
    private static final String STOP_MSG = "IMNstop";

    // Variables
    protected int port;
    protected Settings settings;
    private ServerSocket serverSocket = null;
    private String errorMessage = "";

    // Constructors
```

```
/**
 * @param settings  the settings of the module starting this server
 */
public Server(Settings settings) {
    this.settings = settings;
    do {
        // port =   a random number between PORT_BASE
        //          and PORT_BASE + PORT_RANGE
        port = PORT_BASE + (int)(Math.floor(Math.random() * PORT_RANGE));
    } while (!start());
    register();
}

/**
 * @param settings  the settings of the module starting this server
 * @param port      an integer port number the Server should bind to
 * @exception BindException  if the port is already taken
 */
public Server(Settings settings, int port) throws BindException {
    this.settings = settings;
    this.port = port;
    if (!start()) {
        // Port already taken
        throw new BindException("Port " + port + " seems to be in use already.");
    }
}

// Accesors

/**
 * @return          a string reporting the last error that occured with
 *                  this server object
 */
public String errorMessage() {
    return errorMessage;
}

/**
 * @return          an integer port number that this server has a
 *                  socket bound to
 */
public int port() {
    return port;
}

// Public methods

/**
 * Writes a message to the socket this server is bound to
 *
```

```
     * @param message   a message to be sent
     * @return          a boolean value stating whether the message was
     *                  sent or not
     */
    public boolean sendMessage(Message message) {
        try {
            Socket sendSocket =
                new Socket(InetAddress.getLocalHost(), PORTMAPPER_PORT);
            OutputStream outstream = sendSocket.getOutputStream();
            message.send(outstream);
            outstream.close();
            sendSocket.close();
        }
        catch (UnknownHostException e) {
            errorMessage = "Unknown host: " + e.getMessage();
            return false;
        }
        catch (IOException e) {
            errorMessage = "Can't open connection: " + e.getMessage();
            return false;
        }
        return true;
    }

    /**
     * Reads a message from the socket this server is bound to
     *
     * @return          the message received
     */
    public Message receiveMessage() {
        try {
            Socket client = serverSocket.accept();
            InputStream instream = client.getInputStream();
            Message receivedMsg = new Message(settings.pid());
            receivedMsg.receive(instream);
            instream.close();
            client.close();
            return receivedMsg;
        }
        catch (IOException e) {
            errorMessage = "Could not receive message: " + e.getMessage();
            return null;
        }
    }

    /**
     * Shuts down the server by notifying the portmapper and closing the socket
     *
     * @exception IOException    if an error occurs when closing the socket
     */
```

```
public void stop() throws IOException {
    unregister();
    serverSocket.close();
}

// Private methods

/**
 * Binds the servers socket to port number <code>port</code>
 *
 * @return          a boolean value stating whether the socket was
 *                  successfully bound or not
 */
private boolean start() {
    try {
        serverSocket = new ServerSocket(port, MAX_QUEUE);
    }
    catch (IOException e) {
        // Do nothing - another port number will be attempted
        return false;
    }
    return true;
}

/**
 * Sends a start message to the local portmapper to notify the
 * portmapper of this server's existence
 */
private void register() {
    String[] body = {START_MSG, settings.imn(),
                     String.valueOf(port)};
    Message sendMsg = new Message
        (settings.imn(), PORTMAPPER_NAME, body, settings.pid());
    sendMsg.display();
    sendMessage(sendMsg);
}

/**
 * Sends a stop message to the local portmapper to notify the
 * portmapper that this server will shut down
 */
private void unregister() {
    String[] body = {STOP_MSG, settings.imn()};
    Message sendMsg = new Message
        (settings.imn(), PORTMAPPER_NAME, body, settings.pid());
    sendMsg.display();
    sendMessage(sendMsg);
}

/**
```

```
     * Sets the error message to be reported by <code>errorMessage()</code>.
     *
     * @param errorMessage      a string reporting describing the error
     *                          that has occured
     */
    private void errorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
        return;
    }

    /**
     * Releases the socket once the object is no longer referenced.
     *
     * @exception Throwable     if problem shutting down socket or
     *                          in super.finalize()
     */
    public void finalize() throws Throwable {
        stop();
        super.finalize();
    }

}
```

```
/*
 * Settings
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.io.*;
import java.net.*;

/**
  * A collection of settings for an Alparon module.
  * A module uses a Settings object to keep track of its settings for:
  * <dl>
  * <dt>Generic Module Name (gmn)</dt>
  * <dd>A name which all instances
  *     of the same module have in common. Examples are "PM" and
  *     "INFOBOARD". Generic module names are in caps.</dd>
  * <dt>Instantiated Module Name (imn)</dt>
  * <dd>A name that uniquely identify a
  *     instantiated module. Two instances of the same module
  *     have the same generic module name, but a different
  *     instantiated module name. The instantiated module name
  *     consists of the concatenation of the generic module
  *     name, the host on which the instance is running, and the
  *     process id of the instance. Examples are "PMmercedes3973"
  *     and "INFOBOARDcitroen21874". </dd>
  * <dt>Package name</dt>
  * <dd>The name of the Java package that this Module's class belong to.
  *     Example: "alparon.lib" </dd>
  * <dt>pid</dt>
  *     The process id for the process that created this object.
  *     Example: 21874. </dd>
  * </dl>
  *
  * @author Thomas Oldervoll
  * @version $Id$
  */
public class Settings {
    /*
     * Ported from the Perl module Settings.pm
     */

    // Variables
    private String gmn;
    private String imn;
    private String packageName;
    private int pid;

    // Constructor
```

```
/**
 * @param name      a string Generic Module Name
 * @param pkg       a string package name
 * @param pid        the integer process ID of the process creating the
 *                   object
 */
public Settings(String name, String pkg, int pid) {
    gmn = name;
    packageName = pkg;
    this.pid = pid;
    try {
        String host = InetAddress.getLocalHost().getHostName();
        imn = gmn + host + pid;
        return;
    }
    catch (UnknownHostException e) {
        System.err.println("Local host is unknown: " + e.getMessage());
    }
    return;
}

// Accessors

/**
 * @return          a string Generic Module Name
 */
public String gmn() {
    return gmn;
}

/**
 * @return          a string Instantiated  Module Name
 */
public String imn() {
    return imn;
}

/**
 * @return          the integer process ID of the process creating the
 *                   object
 */
public int pid() {
    return pid;
}

/**
 * @return          a string package name
 */
public String packageName() {
```

```
    return packageName;
}

// Public methods

/**
 * Write settings to System.out.
 */
public void display() {
    // No OutputStream given - use System.out as default
    this.display(System.out);
}

/**
 * Write settings to the given stream.
 *
 * @param outstream an outputstream to write to
 */
public void display(OutputStream outstream) {
    // Transform OutputStream into PrintWriter for conveniency
    PrintWriter out =
        new PrintWriter(new BufferedOutputStream(outstream),true);

    // Print gmn and imn
    out.println();
    out.println(this);
    out.println();

    return;
}

/**
 * @return          a string representation of the message data
 */
public String toString() {
    return "Generic Module name (GMN)\t: " + gmn + "\n" +
        "Instantiated Module name (IMN)\t: " + imn;
}


/**
 * Unit testing routine.
 * Run this by running <code>java Settings</code>.
 *
 * @param args      a string giving the process ID (optional)
 */
public static void main(String[] args) {
    int pid = ((args.length == 0) ? 42 : Integer.parseInt(args[0]));
    Settings test = new Settings("TEST", "alparon.lib", pid);
    System.out.println(test);
```

```
        System.out.println("gmn=" + test.gmn() + ", imn=" + test.imn());
        test.display();
    }

}
```

```
/*
 * Connection
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.util.*;

/** An Alparon messaging connection.
 * Specifies methods for sending and receiving messages
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */

public interface Connection {
    public void send(Message msg);
    public void send(String receiver, String[] body);
    public void send(String receiver, String msgName, String[] contents);
    public Message receive();
    public Message receive(String msgName);
}
```

```
/*
 * MessageHandler
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;

/**
 * A handler for a special type of messages.
 * All classes handling messages for modules should extend this.
 * The <code>handleMessage</code> method should be implemented to
 * receive a the message and read it. Actions should be taken
 * depending on the contents of the message, and these actions
 * will usually involve calling methods of the module the handler
 * is connected to.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see MessagingModule
 */

public abstract class MessageHandler {
    // Variable
    protected MessagingModule module;

    // Constructors

    /**
     * Constructs a handler not connected to any module.
     */
    public MessageHandler() {
    }

    /**
     * Constructs a handler connected to a given module.
     *
     * @param module    a messaging module this handler should serve
     */
    public MessageHandler(MessagingModule mod) {
        module = mod;
    }

    /**
     * Connects this handler to a messaging module.
     *
     * @param mod       the messaging module to connect to
     */
    public void module(MessagingModule mod) {
```

```
        module = mod;
    }

    abstract public void handleMessage(Message msg)
        throws MalformedMessageException;

    /**
     * Checks if the given message has the given message name.
     *
     * @param msg       the message to check
     * @param name      the string message name to check against
     * @exception MalformedMessageException if the message name does
     *                  not match
     */
    protected void checkName(Message msg, String name)
        throws MalformedMessageException {

        if (!msg.body()[0].equals(name)) {
            throw new MalformedMessageException();
        }
    }
}
```

```
/*
 * AddMessageHandler
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;

/**
 * Handler for "AddMessage" messages.
 * A messagingModule can be made capable of handling new messages
 * while running by sending it a message named "AddMessage".
 * The first line of the body is taken as the name of the new message.
 * The next line is taken as the class name for the handler for the
 * message, e.g. <code>alparon.lib.addMessageHandler</code>
 * for this handler.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */
public class AddMessageHandler extends MessageHandler {

    /**
     * @param module    the messaging module this handler should
     *                  serve
     */
    public AddMessageHandler(MessagingModule module) {
        super(module);
    }

    /**
    * @param msg        the message this handler is asked to handle
    * @exception MalformedMessageException if the message had
    *                   bad syntax
    */
    public void handleMessage(Message msg) throws MalformedMessageException {
        checkName(msg, "addMessage");
        String msgName = msg.body()[1];
        String handlerClass = msg.body()[2];
        try {
            //MessageHandler handler = (MessageHandler)o;
            MessageHandler handler =
                (MessageHandler)Class.forName(handlerClass).newInstance();
            // Set "module" of the new object to "module" of addMessageHandler
            handler.module(module);
            // Registre the new handler
            module.addHandler(msgName, handler);
            module.reply(msg, "messageAdded", "ok");
        }
        catch (Exception e) {
```

```
            // Handler not available
            Debug.println(e.getClass().getName() +  e.getMessage(), 9);
            module.reply(msg, "messageAdded", "failed");
        }
    }

}
```

```
/*
 * StopHandler
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;

/**
 * Handler for "Stop" messages.
 * Calls the <code>stop</code> method of the module
 * this handler is connected to.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */

public class StopHandler extends MessageHandler {

    /**
     * Constructs a handler connected to a specified module.
     *
     * @param module    the messaging module to connect to
     */
    public StopHandler(MessagingModule module) {
        super(module);
    }

    /**
     * Calls the connected module's <code>stop</code> method if
     * the given message is a Stop message.
     *
     * @param msg       the message that should be a Stop message
     * @exception MalformedMessageException if the message is
     *                  not a Stop message
     */
    public void handleMessage(Message msg) throws MalformedMessageException {
        checkName(msg, "Stop");
        module.stop();
    }

}
```

```
/*
 * Debug
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;
import java.util.*;

/**
 * A class to print debug messages. All methods are static, so instances
 * should be created. Allows programmers to put debug messages with a
 * specified level in their code. Verbosity of debug output can then
 * be adjusted by setting the debug level with <code>Debug.level</code>.
 * <p>
 * Each debug statement will have a specified level. The string will
 * be printed to System.out if current debug level is higher or equal
 * to the statement's associated level.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */

public class Debug {
    private static final int standardLevel = 2;
    private static int level;  // 0 is no debug info, 9 is maximum

    /**
     * @return          the integer current debug level
     */
    static int level() {
        return level;
    }

    /**
     * @param newLevel  the integer level to set debug level to
     */
    public static void level(int newLevel) {
        level = newLevel;
    }

    /**
     * Print debug string if debug level is higher than standard level.
     *
     * @param text      the string to print
     */
    public static void println(String text) {
        println(text, 2);
    }
```

```
    /**
     * Print debug string if debug level is higher than the given
     * level.
     *
     * @param text      the string to print
     */
    public static void println(String text, int minLevel) {
        if (level >= minLevel) {
            System.err.println(text);
        }
    }

    /**
     * @return      true if debug level is higher or equal to
     *              standard level
     */
    public static boolean on() {
        return (level >= standardLevel);
    }
}
```

```java
/*
 * MalformedMessageException
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;

/**
 * Thrown when a message breaks the rules of syntax for Alparon messages,
 * specified in the documentation of <code>Message</code>.
 *
 *
 * @author Thomas Oldervoll
 * @version $Id$
 *
 * @see Message
 */

public class MalformedMessageException extends Exception {

    /**
     * Constructs exception.
     */
    public MalformedMessageException() {
        super("Malformed message");
    }

    /**
     * Constructs exception with a specified message.
     */
    public MalformedMessageException(String msgString) {
        super("Malformed message:\n" + msgString);
    }

}
```

```
/*
 * UnknownMessageException
 *
 * $Id$ Thomas Oldervoll
 */

package alparon.lib;

/**
 * Thrown when module receives a message that is has no message
 * handler for.
 *
 * @author Thomas Oldervoll
 * @version $Id$
 */

public class UnknownMessageException extends Exception {

    /**
     * Constructs exception.
     */
    public UnknownMessageException() {
        super("Unknown message");
    }

    /**
     * Constructs exception with a specified message.
     */
    public UnknownMessageException(String msgName) {
        super("Unknown message " + msgName);
    }

}
```

Bibliography

[Alexandersson and Reithinger, 1995] J. Alexandersson and N. Reithinger (1995), Designing the Dialogue Component in a Speech Translation System, A Corpus Based Approach, In [Twente95, 1995], pp. 35–43.

[Allen, 1995] J. Allen (1995), *Natural Language Understanding*, The Benjamin/Cummings Publings Company, Inc., 2 edition.

[Allen, 1997] J. Allen (1997), Tutorial on Dialogue Managment, ACL Workshop.

[Aust et al., 1994] H. Aust, M. Oerder, F. Seide, and V. Steinbiss (1994), Experience with the Philips Automatic Train Timetable Information System, in *Proceedings of the IVTTSA '94*, pp. 67–72, Kyoto, Japan.

[Austin, 1962] J.L. Austin (1962), *How to Do Things with Words*, Oxford University Press.

[Baecker et al., 1995] R.M. Baecker, J. Grudin, and W.A.S. Buxton, editors (1995), *Readings in Human-Computer Interaction: Toward the Year 2000*, Morgan Kaufmann Publishres, Inc., 2 edition.

[Baker, 1976] J. Baker (1976), *Stochastic Modeling for Speech Recognition*, PhD thesis, Carnegie Mellon University.

[Binsted et al., 1995] K. Binsted, A. Cawsey, and R. Jones (1995), Generating Personalised Patient Information Using the Medical Record, in *Proceedings of AI in Medicine Europe*.

[Blasband, 1997] M. Blasband (1997), Initial Evaluation of the Dutch Environment, Technical report, ARISE Deliverable D 3.10.

[Bogen, 1972] J. Bogen (1972), *Wittgenstein's Philosophy of Language*, Routledge & Kegan Paul.

[Bratseth, 1997] J.S. Bratseth (1997), Bustuc - A Natural Language Bus Traffic Information System, Master's thesis, Norwegian University of Science and technology (NTNU).

[Button et al., 1995] G. Button, J. Coulter, J.R.E. Lee, and W. Sharrock (1995), *Computers, Minds and Conduct*, Polity Press.

[Campione and Walrath, 1995] M. Campione and K. Walrath (1995), *The Java Tutorial: Object-Oriented Programming for the Internet*, The Java Series, Addison Wesley, Hypertext edition available at http://java.sun.com/docs/books/tutorial/.

[Carletta et al., 1995] J. Carletta, A. Isard, S. Isard, J. Kowtko, G. Doherty-Sneddon, and A. Anderson (1995), The Coding of Dialogue Structure in a Corpus, In [Twente95, 1995], pp. 25–34.

[Cawsey, 1991] A. Cawsey (1991), Generating Interactive Explanations, in *Proceedings of the ninth National Conference on Artificial Intelligence (AAAI-91)*.

[CodeConventions, 1997] CodeConventions (1997), *Java Code Conventions*, Sun Microsystems, Inc., Available at the World Wide Web at http://java.sun.com/docs/codeconv/.

[Cohen and Oviatt, 1994] P.R. Cohen and S.L. Oviatt (1994), The Role of Voice in Human-Machine Communication, in D. Roe and J. Wilpon, editors, *Voice Communication between Huamns and Machines*, pp. 34–75, National Academy Press.

[Dybkjær et al., 1995] Dybkjær, L. Dybkjær, and N.O. Berntsen (1995), Design, Formalization and Evaluation of Spoken Language Dialogue, in *Proceedings of the 9th Twente Workshop on Language Technology*, University of Twente.

[EUROSPEECH'97, 1997] EUROSPEECH'97 (1997), *Proceedings of EUROSPEECH '97*.

[Festival, 1998] Festival (1998), The Festival Speech Synthesis System, http://www.cstr.ed.ac.uk/projects/festival.html.

[Grice, 1975] H.P. Grice (1975), Logic and Conversation, in P. Cole and J.L. Morgan, editors, *Synatx and Semantics*, Academic Press.

[Helander et al., 1997] M.G. Helander, T.K. Landauer, and P.V. Prabhu, editors (1997), *Handbook of Human-Computer Interaction*, Elsevier, 2 edition.

[HLTSurvey, 1996] HLTSurvey (1996), Survey of the State of the Art in Human Language Technology, http://www.cse.ogi.edu/CSLU/HLTsurvey/HLTsurvey.html.

[JavaDoc, 1998] JavaDoc (1998), *How to Write Doc Comments for Javadoc*, Sun Microsystems, Inc.

[K. Arnold, 1996] J. Gosling K. Arnold (1996), *The Java Programming Language*, The Java Series, Addison Wesley.

[Kling, 1995] R. Kling (1995), Controveries About Computerization and the Organization of White Collar Work, In [Baecker et al., 1995], pp. 254–272.

[Larsen, 1997] L.B. Larsen (1997), A Strategy for Mixed-Initiative Dilaogue Control, In [EUROSPEECH'97, 1997].

[Lecourt and Vark, 1997] G. Lecourt and R.J. van Vark (1997), Considering Information Management in Alparon, Technical Report 97–10, Alparon, Delft University of Technology.

[Lieske et al., 1997] C. Lieske, J. Bos, M. Emele, B. Gamback, and C.J. Rupp (1997), Giving Prosody a Meaning, In [EUROSPEECH'97, 1997].

[Manintveld and Rothkrantz, 1997] W.A.Th. Manintveld and L.J.M. Rothkrantz (1997), The OVR–WOz experiment: Setup and Analysis, Technical Report 97–04, Alparon, Delft University of Technology.

[Nielsen, 1994] J. Nielsen (1994), Heuristic Evaluation, in *Usability Inspection Methods*, John Wiley & Sons.

[N.J. Nilsson, 1995] N.J. Nilsson (1995), Eye on the Prize, *AI Magazine*, 16(2):9–17.

[Oldervoll, 1997] T. Oldervoll (1997), Traffic Information Dialogue Systems, Project report, Norwegian University of Science and Technology (NTNU).

[Oldervoll and Vark, 1997a] T. Oldervoll and R.J. van Vark (1997a), Database Manager, version 1.0, Technical Report 97–15, Alparon, Delft University of Technology, under construction.

[Oldervoll and Vark, 1997b] T. Oldervoll and R.J. van Vark (1997b), Query Travel Module, version 1.0, Technical Report 97–19, Alparon, Delft University of Technology, under construction.

[Oldervoll and Vark, 1997c] T. Oldervoll and R.J. van Vark (1997c), Travel Board, version 1.0, Technical Report 97–16, Alparon, Delft University of Technology, under construction.

[Peters and Kock, 1997] A. Peters and J. Kock (1997), The digital phone operator, User appreciation and evaluation of the VIOS speech recognition system, Technical Report 97–05, Alparon, Delft University of Technology.

[Qu, 1997] Y. Qu (1997), Planning Transition Relevance Points in Speech-Based Information Systems, in *Technical Notes of AAAI Spring Symposium on Computational Models for Mixed Initiative Interaction*.

[Rats, 1997a] M.M.M. Rats (1997a), The Suitability of the Alparon Coding Scheme for Dialogue Management, Technical Report 97–02, Alparon, Delft University of Technology.

[Rats, 1997b] M.M.M. Rats (1997b), Towards a better information presentation for VIOS, Technical Report 97–06, Alparon, Delft University of Technology.

[Rats et al., 1997] M.M.M. Rats, R.J. van Vark, and J.P.M. de Vreught (1997), Corpus–Based Information Presentation for a Spoken Public Transport Information System, in *ACL/EACL Workshop on Interactive Spoken Dialog Systems: Bringing Speech and NLP Together in Real Applications*, pp. 106–113.

[Rats and Vreught, 1997] M.M.M. Rats and J.P.M. de Vreught (1997), Parser–Coder, version 1.0, Technical Report 97–21, Alparon, Delft University of Technology.

[Sacks et al., 1974] H. Sacks, E.A. Schegloff, and G. Jefferson (1974), A Simplest Systematics for the Organization of Turn-taking for Conversation, *Language*, 50:696–735.

[Searle, 1969] J.R. Searle (1969), *Speech Acts*, Cambridge University Press.

[Seide and Kellner, 1997] F. Seide and A. Kellner (1997), Towards an Automatd Directory Information, In [EUROSPEECH'97, 1997].

[SpeechMania, 1998] SpeechMania (1998), *SpeechMania 2.0 Developer;s Guide*, Philips Speech Processing.

[SPODIS, 1997] SPODIS (1997), Spoken Dialog Systems for Telephone Services, http://teleipcn.tele.ntnu.no/SPODIS/, Project description on the World Wide Web.

[Twente95, 1995] Twente95 (1995), *Corpus–based Approaches to Dialogue Modelling*, 9th Twente Workshop on Language Technology, University of Twente.

[Vagle et al., 1993] W. Vagle, M. Sandvik, and J. Svennevig (1993), *Tekst og Kontekst – En innføring i tekstlingvistikk og og pragmatikk*, Landslaget for norskundervisning (LNU) / J.W. Cappelens Forlag, In Norwegian (Text and Context – An Introduction to Text Linguistics and Pragmatics).

[Vark, 1997a] R.J. van Vark (1997a), Designing Dialogue Management in the Alparon Project, Technical Report 97–01, Alparon, Delft University of Technology.

[Vark, 1997b] R.J. van Vark (1997b), Dialogue Manager Framework & Test Environment, Source documentation, Technical Report 97–14, Alparon, Delft University of Technology.

[Vark, 1997c] R.J. van Vark (1997c), DiMaCS 1.0, Technical Report 97–18, Alparon, Delft University of Technology, under construction.

[Vark, 1997d] R.J. van Vark (1997d), Module Library, Source documentation, Technical Report 97–13, Alparon, Delft University of Technology.

[Vark et al., 1997a] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz (1997a), Classification of Public Transport Information Dialogues using an Information Based Coding Scheme, *Dialogue Processing in Spoken Language Systems, Lecture Notes in Artificial Intelligence (1236)*, pp. 55–69.

[Vark et al., 1997b] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz (1997b), Analysing OVR Dialogues, Coding Scheme 1.0, Technical Report 96–137, Alparon, Delft University of Technology.

[Vark et al., 1997c] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz (1997c), Analysis of the OVR log files, Progress Report, Technical Report 97–03, Alparon, Delft University of Technology.

[Vark et al., 1997d] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz (1997d), An Automated Speech Processing System for Public Transport Information Services, in *Third International Congress on Information Engineering*, pp. 212–221.

[Veldhuijzen van Zanten, 1996] G. Veldhuijzen van Zanten (1996), Pragmatic Interpretation and Dialogue Management in Spoken–Language Systems, in *Dialogue Management in Natural Language Systems*, 11th Twente Workshop on Language Technology, pp. 81–88, University of Twente.

[Vreught and Rothkrantz, 1997] J.P.M. de Vreught and L.J.M. Rothkrantz (1997), Datamining OVR Log Files, Technical Report 97–22, Alparon, Delft University of Technology.

[White, 1990] G.M. White (1990), Natural Language Understanding and Speech Recognition, *Communications of the ACM*, 33(8).

[Wittgenstein, 1958] L. Wittgenstein (1958), *Preliminary Studies for the 'Philosophical Investiagtions': The Blue and Brown Books*, Basil Blackwell.

[Yankelovich et al., 1995] N. Yankelovich, G.A. Levow, and M. Marx (1995), Designing SpeechActs: Issues in Speech User Interfaces, in *Human factors in computing systems*, ACM Press.