Natural Language Engineering **6** (3–4): 341–362. Printed in the United Kingdom

© 2000 Cambridge University Press

Object-oriented modelling of spoken language dialogue systems

IAN M. O'NEILL and MICHAEL F. McTEAR

Faculty of Informatics, University of Ulster, Newtownabbey,
Co. Antrim BT37 0QB, Northern Ireland
e-mail: {im.oneill,mf.mctear}@ulst.ac.uk

(*Received 16 August 1999, revised 14 January 2000*)

Abstract

In this paper we show how established object modelling techniques can be used in the creation of spoken dialogue management systems. One of the motivations behind the particular approach adopted here is the observation that, in spoken human-to-human dialogues, certain skillsets and patterns of dialogue evolution are common to many different contexts; other dialogue skills and accompanying real-world knowledge are required only for more specialised transactions within particular business domains. As a starting point for modelling an automated spoken dialogue management system we recommend a use case analysis of the required functionality. The use case analysis encourages the developer to identify generic-specific relationships and interactions between different dialogue management skills. We consider some of the broad philosophies underlying current dialogue management systems and outline practical high-level dialogue behaviour based on mixed-initiative, frame-based processing, combined with a rigorously applied confirmation strategy. On the basis of the use case requirements analysis, we explore a possible design for an object-oriented dialogue management system, indicating the roles and relationships of the various classes that embody the required dialogue functionality, and showing how implemented objects within the system will interact. The manner of this interaction is such as to allow one overall system to process transactions in several business domains. We also indicate some of the advantages of a rule-based implementation: the proposed design is tailored towards such an implementation in Prolog++. An object-oriented development process places high-level, generic dialogue management functionality at the disposal of more specialised 'expert' components. Maintainability and extensibility are therefore enhanced: if the developer chooses to refine generic behaviour, it is immediately available to the more specialised components; if new domain-specific expertise is required, it can be added with minimal impact on generic behaviour.

1 Introduction

Conversational spoken language dialogue systems are intended to allow a reasonably free interaction between user and system in order to complete some kind of transaction. Until recently, however, there has been no consensus on methodologies for the design of these systems and developers have largely proceeded on a fairly *ad hoc* basis, or have developed systems that illustrate a particular theoretical position. A recent textbook documents the essential elements of best practice in the design of

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

interactive speech systems, from the stage of initial concept through analysis, design and implementation to user testing and evaluation (Bernsen, Dybkjaer and Dybkjaer 1998). Further work on the development of a best practice methodology is ongoing within the Esprit research project DISC (`http://www.disc2.dk`).

The current paper complements this work on best practice through the application of well-tested object-oriented methods to the specification, design and implementation of spoken dialogue systems. Whereas the emphasis in the 'best practice' research of Bernsen and colleagues within the DISC project is mainly on guidelines that support the principle of cooperativity in interactive systems, here the main concern is how object-oriented modelling can provide a principled account of system functionality in terms of the roles and relationships between the components of the system as well as in terms of how they interact with each other to provide for a flexible, mixed-initiative dialogue control strategy.

2 Motivations for an object-oriented approach to dialogue management

If one examines a number of different kinds of transaction, it soon becomes apparent, that regardless of the business domain of the transaction, there are certain set-pieces that are common to all such transactional dialogues. There are the introductory formalities; the process of eliciting the necessary information from the user; confirmation that the information given is accurate; completion of and confirmation of the transaction itself. If, in addition, the system supports transactions that fall into one or more of several business domains, there is the process by which the system and the user help each other orient the transaction in the appropriate business domain(s).

However, though at this high, rather abstract level, many dialogues share structure and functionality, there are significant differences when it comes to the detail of what has to be done in transactions in different business domains. The type and amount of information to be elicited from the user and provided by the system may vary greatly from domain to domain. The expertise the system needs to direct the transaction towards a successful conclusion – making best use of the information that the user has provided, knowing what meaningfully to ask for next, being able to provide suitable alternatives when a request cannot be processed – all these abilities may involve complex rules of thumb or heuristics that capture something of the reasoning and processing skills of a human domain specialist. At this level of functionality we are aiming to capture in an automated system skillsets that in the real world might distinguish a customer adviser at a travel agency, say, from a booking clerk at the theatre: in the real world, both will have many conversational and reasoning abilities in common; in the real world, only one will have the experience and the resources to book a weekend super saver fare to Boston.

The aim of the object-oriented approach to dialogue management is to 'abstract out' into implementable components those functional elements that are common to many different kinds of dialogue and those that are specific to particular business domains. Functionality that is applicable to several domains can be inherited by or used by other, more domain-specific components. Specialised components are at the

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

disposal of the broader system to help deal with enquiries as they enter areas where domain-specific expertise is required.

It would be quite possible, if rather short-sighted, to construct a monolithic dialogue management system for a particular business domain, with no immediate aim of reusing elements of the system for dialogue processing in other domains. Should implementation in a new domain be required at some later stage, it might of course be possible to identify and adapt in an ad hoc manner existing functionality. However, an object-oriented approach maximises opportunities for reuse and adaptation from the outset. In following an object-oriented methodology the developer aims to encapsulate in some components generic, inheritable functionality and in others, domain-specific functionality; the developer also creates discrete collaborating entities with clear-cut interfaces. In the case of the prototype implementation used to illustrate this paper, reported in O'Neill and McTear (1999), the object-oriented approach had the direct effect of creating an architecture that easily supported transaction processing in different, specialist business domains – travel enquiries and event-related enquiries were the areas elaborated to illustrate the principle. Processing routines and utilities common to both domains are used by the specialist domain experts, which add detailed, domain-specific processing rules of their own. Additionally, it becomes a relatively simple matter to give higher level components an opportunity to select dynamically, on the basis of identified key semantic input, the domain expert best suited to the evolving transaction. (An analogous approach is the one adopted by Wang, who uses a semantic grammar in a base class to provide high-level understanding of an utterance, and then finds a 'best match' from among the grammars of derived classes for a more detailed understanding (Wang 1998).)

3 A method for modelling the dialogue management system

It is an accepted fact of object-oriented analysis and design that there no single 'correct' solution when it comes to organising automated functionality to deal with a real-world task. This paper does not set out to establish a blueprint for an architecture against which all spoken dialogue systems must be modelled: rather it proposes a way of looking at spoken dialogue systems and of using a set of established modelling techniques so as to draw out the object-oriented nature of dialogue.

The somewhat individualistic nature of the analysis and design task is complicated by the fact that, in the real world, systems may not only have to be forward-engineered, where the developer starts with a clean sheet and needs lay out the structure and functionality of a system as clearly and concisely as possible, but they may have to be reverse-engineered also: the software engineer may have to represent in an object-oriented notation an existing implementation that has perhaps evolved rather haphazardly. Here the aim may be to facilitate the maintenance task or to provide a starting point for a new, more intuitive and maintainable implementation.

Ideally, one and the same set of basic techniques and notations will if necessary facilitate both the forward- and reverse-engineering tasks: this happily is the case with the User Modeling Language (UML) of Booch, Rumbaugh and Jacobson, the

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

distillation of more than a decade's work in developing best practice in object-oriented software engineering (Booch, Rumbaugh and Jacobson 1998). In looking specifically at how spoken dialogue management might be represented as an object-based system we will be guided by the recommendations and techniques of the UML, as well as by our own experience of developing commercial object-oriented software.

3.1 Starting the use case analysis: who will be speaking to the system and why?

It goes without saying that in designing a software system the developer has to know who is going to use the system and for what purpose. In the development of an object-oriented system, knowing the aims and objectives of users as they interact with the software takes on a new dimension, since understanding the needs of a generic user, and appreciating the requirements of more specialised users, is generally the first step in separating the system's functionality into the generic and the specialised. Knowing the generic-to-specialised relationship between different groups of people can lay the foundations for the 'uses' and 'inherits' relationships between components in the software itself. In a mixed initiative, spoken dialogue system, where the emphases are on the accuracy and the naturalness of the interaction between user and system, it becomes all the more important that developers understand how users generally can interact intuitively with the system, and, once the generic modus operandi is established, how dialogue can be specialised to deal with specialised business and reasoning skillsets.

A 'use case' approach to requirements analysis is now widely accepted as well suited to object-oriented development. The notion here is that the system to be implemented is described by sets of sequences of interaction between the user and the system. These 'sets of sequences of interaction' constitute the individual use cases, which are described by the simple graphical notation shown here, as well as by brief textual descriptions. These textual descriptions of the sorts of interaction that take place between user and system will be important in the more detailed stages of system development: a textual analysis of the descriptions is an important step in a forward-engineered system for determining the main objects in the system, along with their methods, attributes and relationship to other objects.

How does the developer establish the scope of the functionality described in each use case? Each use case serves some useful purpose for the user – it accomplishes something that the user perceives as useful. Each use case describes typical interaction, as well as exceptions and error handling – thus it can rightly be described as a 'set of sequences'.

The human 'actor' (a person acting in a particular business capacity or role), accesses the functionality represented by the use case. To illustrate the object-oriented approach in action, let us say the overall dialogue management system under development is primarily concerned with providing information. The highly generic user, let us simply call him or her the Customer, might access the functionality of a use case that is simply called Process enquiry (see Figure 1).

According to such a scheme, Process enquiry deals with very high-level in-

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

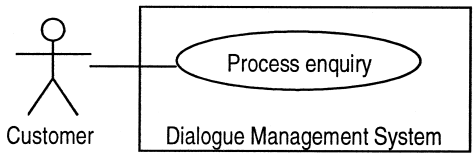


Fig. 1. A basic enquiry system.

teractions between the user and the system, indicating the manner in which the system responds to very general service requests, and outlining the broad pattern of interaction between user and system.

The written description to accompany the use case Process enquiry will include descriptions of very high level requests that the user will typically make, and the manner in which the system responds, encouraging the user whenever possible to narrow the scope of those requests. Perhaps more significantly the description will probably indicate which mechanisms in the system are at play to enable information to be input to and output from Process enquiry's decision making processes.

If the system is only ever intended to deal with one real-world application area, then conceivably all the expertise needed to process enquiries in that area might be included in the use case Process enquiry. However, if we want to exploit the dialogue management system's potential to deal with enquiries in a number of application areas, in as efficient and maintainable a manner as possible, then already our representation of the automated dialogue system is oversimplified. Let us investigate the situation where the system is intended not just to 'provide information' in a single business area, but is intended, for example, to meet the needs of customers that want entertainment information as well as travel information. Not only do we now have two subclasses of Customer, let us call them Event-goer and Traveller, but – and this is where our understanding of the types of user finds its direct counterpart in the system implementation – we may also assume that the system has at its disposal more specialised functionality in order to process, on the one hand, the Event-goer's event enquiry, and on the other hand the Traveller's travel enquiry. From the system developer's perspective, then, we can immediately add two new use cases, Process event enquiry and Process travel enquiry (see Figure 2), each inheriting the generalised or generic processing capabilities described by Process enquiry, but adding domain specific know-how. Building on the generalised behaviour, each of Process enquiry's specialisations describe how the system guides the user through detailed transactions within a specific real-world business domain.

This idea that a core 'way of dealing with dialogues' lies at the heart of the majority of interactions with the user is fundamental to this particular object-oriented approach to the design of spoken language dialogue systems. So far we have suggested one level of genericity, represented by the Process enquiry use case, the use case that sets the tone for the manner in which the system prompts the user for information and utilises that information to service a user's enquiry. Having decided to 'abstract out' generic enquiry processing capabilities from the more domain-specific, the developer might reasonably choose to describe all the generic behaviour needed

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

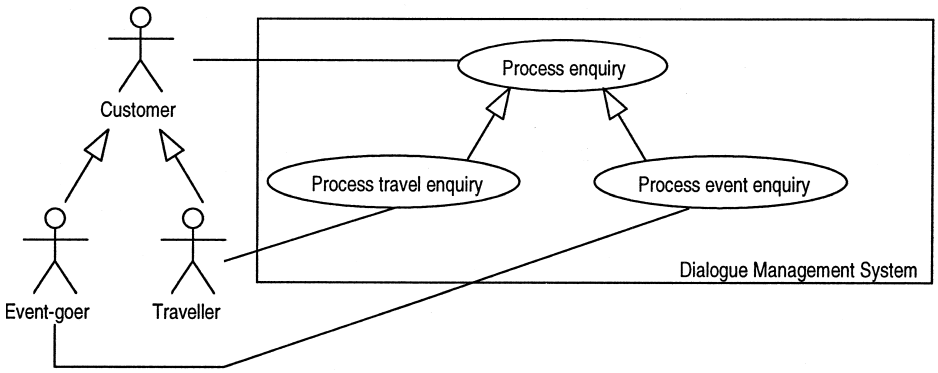


Fig. 2. A system able to handle different types of enquiry.

to manage the dialogue in the generalised use case. Alternatively, from the point of view of creating as adaptable and maintainable an architecture as possible, it might also be useful to identify dialogue behaviour that is still more generic than that needed to support an enquiry-based information service alone. For example, whatever the purpose of the information exchange between system and user, the system will need a confirmation strategy if it is to compensate adequately for its inevitably imperfect artificial hearing and ensure that it has heard the user's utterances correctly. In this illustration of the requirements analysis process, let us then add another level of abstraction – Manage discourse – that will describe the largely mechanistic process, and the decision-making criteria, by which the system confirms new information supplied by the user and queries information that has been changed or negated. This highly generic processing will be inherited by Process enquiry and will be passed on to be incorporated in the domain-specific processing strategies described by Process travel enquiry and Process event enquiry (see Figure 3).

4 System functionality that supports the core transaction

The use case approach is concerned with more than just the inheritance relationships by means of which different functional areas within the system share characteristics and add characteristics of their own. Along with the inheritance relationship (i.e. generalised – specialised), use cases can have include relationships, in which one piece of commonly used functionality is 'slotted into' the processing routine of another, and extend relationships, where optional or alternative functionality is separated out from mainstream interaction. The use of such relationships in the requirements analysis gives the developer the opportunity of documenting some of the dialogue interactions that support the core interaction of information request and information provision.

Let us look at the situation in the opening stages of a dialogue between the user and a system that is capable of handling transactions in a number of real-world domains. Once the semantic content of the user-utterance is passed to the dialogue management system, the latter has to decide whether one of its areas of

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

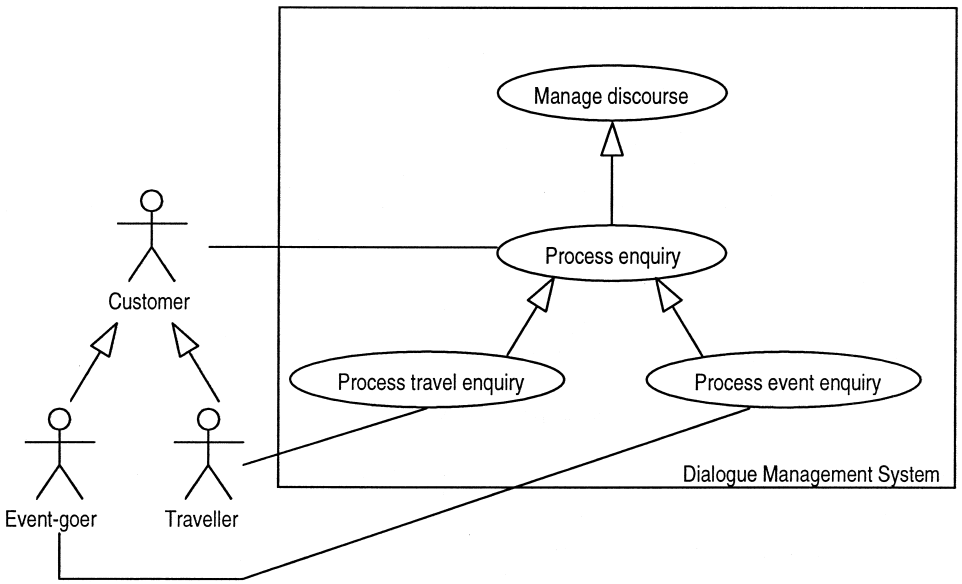


Fig. 3. Discourse management identified as highly generic behaviour.

expertise can handle the enquiry, or whether the user must be prompted to provide more information and set the context of the transaction. We shall call the area of functionality that has the necessary 'overview' of the system's capabilities Identify domain. This use case will describe the broad rules of thumb used by the system to associate key semantic concepts with particular processing domains.

When it describes how semantic content is passed to the main system expertise for further processing, the use case Identify domain can 'include' the functionality described in the use case Process enquiry. In the UML, this include relationship is formally represented in a written use case description in the following manner:

"If the utterance is too ambiguous to be placed in a single business domain, the system will check what expertise is available to it. include(Process enquiry)."

In the context of an overall systems analysis this is just one of several potential include relationships that might appear in the use case analysis of system functionality. A use case Manage I/O that arbitrates input to and output from the overall dialogue control system – in other words enables the to and fro of semantic material between user and system, regardless of the nature of that material – would include the functionality of Identify domain. Likewise, for the sake of maintainability, it might be decided to separate out part of the functionality of Manage discourse. While the core functionality described in Manage discourse aims to react in predictable ways to new and changed input from the user, the system also has a facility for storing and recalling, for its own and the user's benefit, the semantic content of its own and the user's utterance. Here again there is an opportunity to include functionality of a Log discourse use case within the functionality described in the Manage discourse use case. Figure 4 shows the overall use case model, with the include relationships.

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

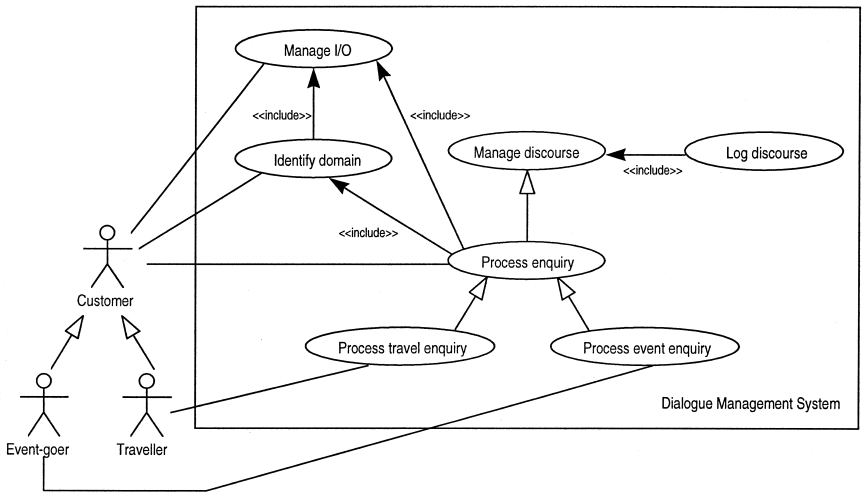


Fig. 4. Identifying functionality that supports the main transactions.

Already, then, an object-oriented approach to requirements analysis has given us a useful set of inter-related functional areas – and at a glance it is also clear to see that in some sense we have already established a very high level system architecture that will translate well into actual implemented components. Let us, though, at this point attempt to put some detail on the sort of generic and specialised functionality that might exist within the system, and in particular the relationship between the system's generic confirmation strategy and its domain specific expertise.

5 More fundamental questions regarding dialogue management

5.1 Candidates for a generic management strategy

Even a quick comparison of current, implemented dialogue management systems reveals fundamentally different approaches to the dialogue management task. Current solutions span a continuum from highly constrained and computationally rudimentary to naturalistic and computationally demanding. Designing a broad, domain-independent dialogue management strategy, whose influence permeates an entire system, may be regarded as a delicate balancing act between, on the one hand, academic discourse theory – with its notions of intentions and obligations in dialogue – and, on the other hand, practical considerations of how relatively natural spoken interaction with the human user can be achieved with reasonable implementational and computational effort.

Let us briefly consider the two extremes of the continuum just mentioned. For the sake of achieving a viable, if closely constrained, implemented version of human dialogue some systems simply present the user with a series of pre-determined choices at different junctures (Novick and Sutton, 1996). These 'finite state machines' are perfectly adequate for relatively simple transactions – ordering a pizza with a range of toppings, for instance. Such an approach tends also to yield reusable 'set pieces'

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

that can be integrated into dialogue when particular kinds of information are needed – credit card or address information, for example. Alternatively, the system developer might pursue the more rigorous approach based on discourse theory, attempting to model in the implemented system an understanding of the purpose behind the user's utterance – whether it is a query, a confirmation and so on – and then getting the system to respond appropriately (Allen, Miller, Ringger and Sikorski 1996). However, attempting to endow the automated system with a sense of dialogue understanding based on principles of discourse theory will probably remain the preserve of dedicated academic research teams for some years to come.

Of particular interest to us is the middle ground between the highly constrained finite state machines and philosophically rich dialogue management based on discourse theory. Some of the most successful currently implemented systems occupy this middle ground, adopting a mixed-initiative, frame-based approach (Aust and Oerder 1995; Aust, Oerder, Seide and Steinbiss 1995). Here the system accepts whatever relevant information is offered by the user – even if the system had not explicitly prompted for it – and uses the information to populate a request template. This request template contains all the data needed to complete a particular kind of transaction – and a complex system may contain a number of templates for different kinds of transaction. While pragmatically setting out to word-spot whatever information it needs to complete its domain-specific transaction, most frame-based applications will methodically confirm the relevant semantic content of the user's utterances – often using a mixture of implicit and explicit confirmation strategies. For implicit confirmation, the system might incorporate information supplied by the user into the wording of its next question; in confirming explicitly the system might simply ask the user if it has understood one or more values correctly. In developing an actual prototype based on the OO architecture described here, we have favoured the practical strengths of the frame-based approaches and have combined a methodical confirmation strategy with mixed initiative dialogue control. The OO approach has the added advantage that the generic dialogue control strategy is combined, through the inheritance mechanism of object-orientation, with domain-specific and ultimately organisation-specific transaction management.

So, what kind of criteria might a practical system use to determine the confirmedness and therefore the usability of the information supplied by the user? This is the sort of processing that will be described in the Manage discourse use case. The strategy can of course be as subtle as the developer chooses to make it. However, in each of its dialogue turns a fairly basic system might aim to confirm, either implicitly or explicitly, up to a maximum number of semantic values that are new to it; and query the user rather more pointedly – perhaps dealing with a single value at a time – when it comes to values that have been changed or negated. A basic system might also incorporate into its generic confirmation strategy the use of discourse pegs – counters that correspond to the key attributes of the current query and that are incremented as semantic values for the attributes are confirmed or repeated by the user, and reset or decremented as the values are changed or negated. Adopting a system of discourse pegs allows the developer to set objective thresholds, above which information supplied by the user can be used by the system with relative safety

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

to help further the transaction. In particular, confirmed information can be used with the domain-specific processing rules in specialised business-related use cases.

5.2 Getting down to business: domain-specific strategies

What kind of processing might be described by the use cases for the specific business domains? In short, these must attempt as far as possible to spell out the heuristics or rules of thumb that a human expert would use to progress a transaction, given that certain details have been supplied by the customer, and recognised with reasonable certainty by the expert. It might, for example, be reasonable to indicate in the Process for travel domain use case that, if the customer has stated that he or she wants to travel to Belfast, the system should respond by asking the customer when they want to travel, and so on.

However, a dialogue system intended to answer enquiries about real-world services will probably need to identify a real-world organisation that can act as service provider. Once a good candidate service provider has been identified, the system will attempt to process the customer's inquiry to a conclusion in terms of that provider's service offering.

These last points have important implications for the implemented architecture – for already there is the suggestion that at different stages of its processing the system will have to investigate which component or type of component is best suited to moving the transaction towards a conclusion. Moreover, a system based on the emerging architecture will be able to poll its components at different levels of specialisation, first checking quite generically which kinds of processing are available to handle the incoming enquiry – i.e. looking for the implemented counterparts of Process for travel domain or Process for event domain – and ultimately interrogating specific implemented instances, each representing a single real-world service provider, to see which one's service offering best matches the confirmed details supplied by the customer.

6 Design: finding practical solutions

So far we have considered a relatively high-level technique for ordering and describing in natural language the main functional areas of a mixed initiative, cross-domain dialogue management system. Let us now look in greater detail at how the functional areas identified might translate into classes and objects with their own attributes and methods. In particular we shall explore the manner in which the various levels of polling within the system might operate, and how patterns of inheritance and collaboration make for a practical, maintainable system whose generic dialogue processing sits easily with an evolving body of domain- and instance-specific expertise.

6.1 Designing with an eye to implementation: a rule-based approach to dialogue

The design of a system will inevitably influence the choice of implementation language. It is often also the case that the preferred implementation language works

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

more easily with some design features than with others. This may lead to rather pragmatic design choices. However, it goes without saying that an object-oriented design is implemented most authentically in an object-oriented programming language. In commercial applications C++ is a well established and well proven choice, and Java is now making considerable inroads too: for many software developers such languages are easily assimilated variations on a familiar procedural theme. For a system that is attempting to recreate something like the behaviour of human dialogue partner, the developer may choose to take a rather different approach to the design – a declarative or rule-based approach where the emphasis is on establishing the rules of thumb that characterise spoken transactions in the real world, then making these rules available to the system *en masse*, and letting the system itself choose which rule is most appropriate to use in a given set of circumstances. Now the developer is more concerned with what the system does than how it will do it. In a rule-based approach the aim is to capture as directly as possible, in formalised rules, the business know-how and conversational abilities of a human dialogue partner. Rule-based languages like Prolog make implementations that adopt this design philosophy possible, while an object-oriented variant like Prolog++ additionally caters for the main features of object-based programming: inheriting, collaborating classes and objects with their own encapsulated data and methods. To explore the viability of the architecture proposed here, we have been experimenting for the last eighteen months with a prototype created in Prolog++. While the scope and purpose of this paper preclude a detailed treatment of the implementation, it is perhaps as well that the user is aware of the broad philosophy behind the design and the manner in which that design can become a working program.

6.2 From use cases to class relationships: a working system takes shape

As we have mentioned briefly earlier, the textual descriptions that accompany the use case diagrams provide a basis for a textual analysis, typically focusing on nouns (potential objects and attributes) and verbs (potential methods, potential 'is a' and 'has a' relationships, and so on). This high level understanding of the features of the problem domain translates into class relationship and object interaction diagrams in the UML.

In a detailed design exercise, based on a close study of actual business practice, the textual analyses of the use cases might each yield complete mechanisms of collaborating objects of different types. Here, though, we shall assume that the functionality of each of the use cases can be implemented by a single class: not only can this serve as the basis of an actual, implemented prototype, but it also keeps reasonably transparent the most important paths of collaboration and inheritance between the main dialogue system components.

In moving from the use case description to the object model there is an important change in perspective. Having documented the behaviour the user expects from the system, we are now looking at the processing that takes place within, and the interactions that occur between, 'implementable' system components to make that behaviour a reality. At the most trivial level, this change of perspective is reflected in

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

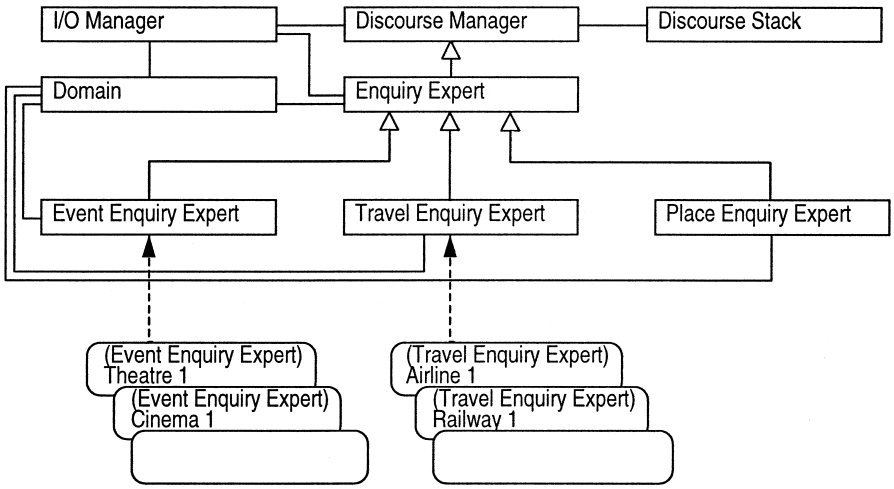


Fig. 5. Main system components.

the names of the classes that emerge, based on nouns rather than verbs: no longer do we simply have an area of functionality within the system whose purpose is to Process enquiry; rather we have an Enquiry Expert, an implementable entity that will deal with enquiries at a highly generic level.

Drawing on the use case understanding of the relationships between the system's main areas of functionality, we can readily create a class relationship diagram (see Figure 5). As in the use case model, the open-headed arrow indicates a generic-specialised relationship, i.e. an inheritance relationship. To give an idea of how the system might expand as new expertise is required, we have also added a Place Enquiry Expert – a provider of information about places of touristic interest, say.

A closer view of (a portion of) the class relationship model might reveal detail along the following lines - now the main functions and attributes within the design of the system are beginning to emerge (see Figure 6).

7 A suite of inheriting, collaborating components for spoken dialogue management

While the design of encapsulated, inheriting and collaborating functionality described here is by no means the last word on the matter, it will, we hope, serve as a reasonable illustration of the sort of object-oriented solution that is suited to spoken dialogue management.

The prototype system used to illustrate the concepts underlying this paper represents a Dialogue Manager in isolation. It receives input semantic constructs such as might reasonably be produced by the combined efforts of Speech Recognition and Natural Language Processing components. Likewise, it produces semantic constructs that might form the input to a Natural Language Generation component. Currently we do not specify the architecture that would enable the Dialogue Manager to interact with these other modules. However, a likely architecture for a fully integrated system is the currently evolving DARPA hub (MITRE 1999), which allows each

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

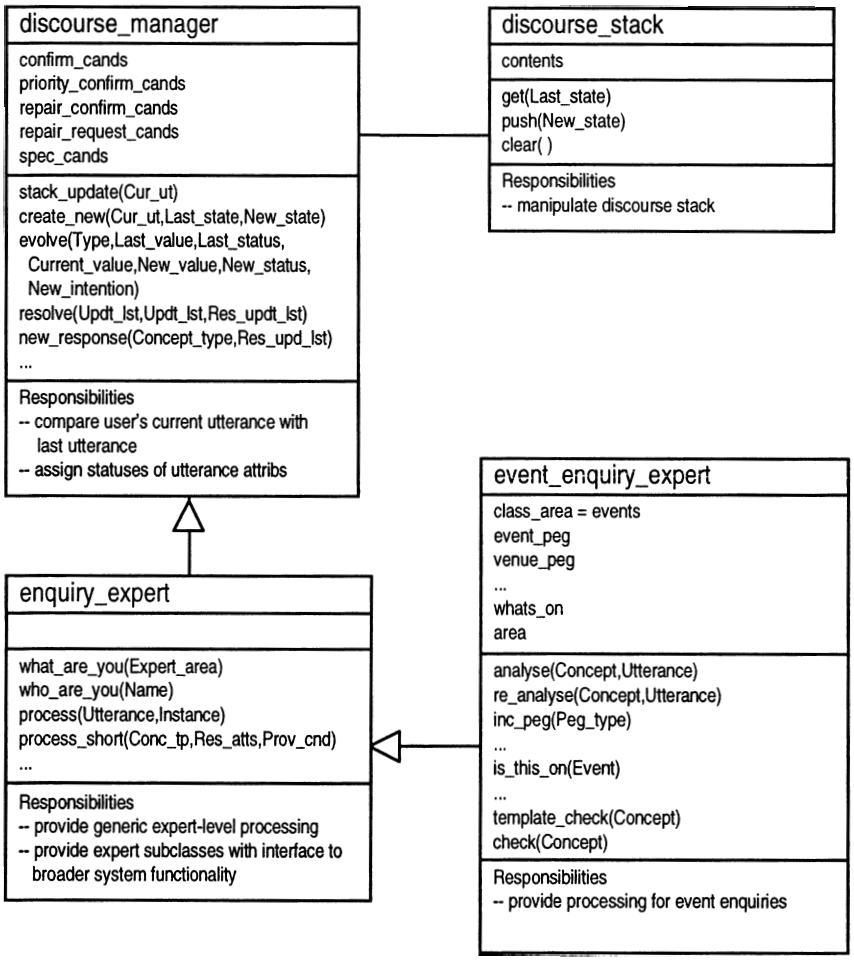


Fig. 6. Classes, attributes, methods and responsibilities.

major system module to be treated as a server that communicates with its peers by passing frames of data through a central router or hub. In such an architecture the Dialogue Manager described here would receive and produce hub-compliant frames – rather than the more idiosyncratic Prolog++ constructs of the current implementation. The DARPA architecture also allows implementations considerable latitude as to the precise scope of the functionality provided by an individual server, though the current technological state-of-the-art gives rise naturally to some typical functional demarcations – compare Aberdeen *et al.* (1999) and Ward and Pellom (1999).

Inheritance and collaboration relationships set the tone for the entire system operation, encouraging a development philosophy that gives higher level components only a limited knowledge of the capabilities of the lower level components. Indeed, in the design proposed here, higher level components will periodically poll lower level components to see what expertise within the system is best suited to particular

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

types of enquiry or stages within an enquiry. From the point of view of maintenance this has the advantage that new, detailed, task-specific expertise can be easily added in the lower reaches of the object hierarchy without affecting broader dialogue strategies.

Let us first examine briefly the roles and relationships of the classes in the system.

7.1 Introducing the main components

The I/O Manager receives, forwards for processing and outputs the semantic content of utterances. Again, it is envisaged that input to and output from the dialogue management component of a spoken dialogue system would be in the form of a semantic structure (implemented in Prolog++ for example).

Upon receiving the input utterance from the I/O Manager, the Domain Spotter identifies key semantic content in the user's utterances and checks available system expertise. It then either queries the user further – if the understood meaning of the user's utterance appears to bear very little relevance to the system's competencies – or it forwards the user's utterance to the most appropriate expert or class of expert for further processing.

The Discourse Manager implements the system's generic confirmation strategy. It indicates the circumstances in which values of the attributes in the user's utterances should be regarded as confirmed or uncertain. It maintains a check on degrees of 'confirmedness' by assigning one of several status values to the attributes, the main ones being: new for system, repeated by user, modified by user, negated by user, inferred by system (Heisterkamp and McGlashan 1996). It reviews these statuses to help the system select possible next actions – for example, to have the system confirm attribute values that are new to the system, or have the user provide a new attribute value if a previously recognised value for the attribute has been negated. It tells domain experts when they should increment or decrement the discourse pegs relating to the attributes: only when attributes have been confirmed or 'pegged' to a sufficient degree will they be used to complete a transaction. Perhaps most significantly, it provides opportunities for domain-specific rules of thumb to fire, once a generic domain expert or a specific instance of a domain expert has been selected to process the transaction: domain experts inherit the functionality of the Discourse Manager.

The Discourse Stack is used by the Discourse Manager for logging what it believes the user has said, the extent to which the contents of the user's utterance have been confirmed, and the manner in which the system has decided to respond. The Discourse Manager uses the Discourse Stack to retrieve the user's previous utterance, in order to compare it with the current utterance, and thereby determine the discourse status and level of confirmation of the attributes of the current utterance.

The Enquiry Expert Class supports very high level domain expert functionality – providing, for example, the functionality that allows a domain expert (a Travel Enquiry Expert or an Event Enquiry Expert, for instance) to tell the broader system what its specialism is. The Enquiry Expert Class also serves as the interface between

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

individual domain experts and the resources of the system at large: the I/O Manager and the Domain Spotter for instance. Very significantly, the Enquiry Expert Class inherits the generic confirmation strategy from the Discourse Manager, and thus the enquiry experts for the different subdomains (travel, events, etc.) have that same generic confirmation strategy at their disposal. When they come to use their domain-specific rules of thumb (see below), they will be applying their rules to information that they have confirmed according to the mechanisms of the system-wide confirmation strategy.

The Enquiry Expert subclasses (the 'domain experts') contain the functionality that allows the system to conduct enquiries in particular business domains – in the example system under discussion here we have a Travel Enquiry Expert, an Event Enquiry Expert and a Place Enquiry Expert. The Enquiry Expert subclasses know what confirmed information they need to complete an enquiry, when they should elicit additional information, and how they should offer alternative services. These capabilities take the form of detailed rules of thumb specific to the processing domain – and it is these rules that are given appropriate opportunities to fire by the inherited generic dialogue behaviour.

For an enquiry to be processed to a successful conclusion a 'handling agent' – a specific instance of an Enquiry Expert subclass – must be selected. These instances, which form the leaves of the inheritance tree, have access to the hard data (in the prototype, event programmes and travel schedules) that characterise individual real-world businesses. In real-world terms the handling agent would represent an actual airline, train company, theatre or cinema. Processing by an such an agent is characterised by domain-specific and inherited, generic dialogue behaviour, all brought to bear on the data that are specific to the agent itself.

7.2 How dialogue exchanges are processed: the object-based approach in action

So, let us now examine how a transaction based on this architecture might work out in practice. The UML provides an appropriate notation for modelling the detailed interaction of the objects involved in a particular transaction.

Take as an example a situation where the user's utterance indicates only that he or she wants to make a booking and gives no further details. In these circumstances Domain Spotter is programmed to interrogate the Enquiry Expert subclasses to find out which ones can handle bookings. (It is perhaps worth mentioning that Prolog++ allows coded 'classes' to be treated as instances of 'generic objects' – and this is reflected in the terminology we use here: a Travel Expert class is synonymous with a generic Travel Expert object.) The interaction diagram (see Figure 7) illustrates the process.

If a subclass does handle bookings, it simply pushes its class area attribute (indicating its area of competency: travel, or events, say) on to the class candidate list within Domain Spotter. Otherwise it lets the call pass it by. In the example, the Place Enquiry Expert does not handle bookings. In the system's next dialogue turn the Dialogue Manager uses the contents of the list to offer the user a selection of business areas to choose from. The important thing, from the point of view of

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

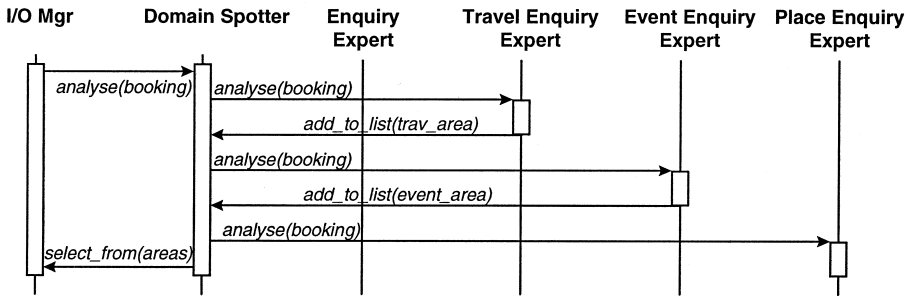


Fig. 7. Finding the relevant subclass.

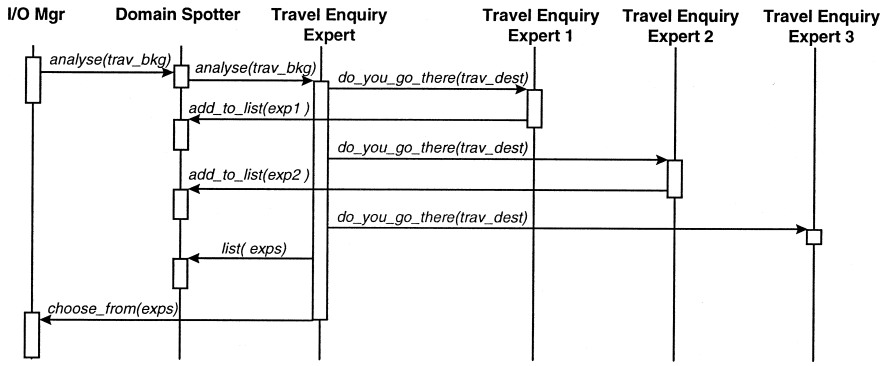


Fig. 8. Identifying appropriate instances.

system maintenance, is that the Domain Spotter does not know in advance which volunteers will offer their services. It has to check with them first.

One should also note that, to support mixed initiative dialogues, the dialogue management system must be able to receive and process more information than it specifically requests. When, in this example, the system asks the user to select a business area (an Enquiry Expert subclass in effect – travel or events), the user in a single turn may not only name the area but may even go on to provide enough information for a handling agent to be selected – a particular instance of a Travel Enquiry Expert, for example. This would have the effect bringing the transaction quickly into its most detailed phase, with the possibility that it might be completed in terms of the service offered by the selected agent. However, if there is no clear candidate for handling agent, the system has to help out.

In working through a representative example, let us say that the Domain Spotter has got as far as working out that a Travel Enquiry Expert should be handling an enquiry for a plane journey at a particular time and to a particular place. Domain Spotter passes the query to the generic Travel Enquiry Expert, which in turn interrogates its instances to see how many have airline as an attribute, and travel to the destination on the day and at the time requested. The interaction diagram (see Figure 8) illustrates the process.

If the instance is unable to meet the criteria it simply passes the call to the

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

next instance. Any instance that can provide the required service adds its name (in Prolog++ this is its unique mnemonic) to Domain Spotter's candidate list. If there are no candidates, the system may suggest another service; if there are several candidates, the system will get the user to choose; if there is one, processing will commence in terms of that handling instance.

Finding the handling instance, in other words, is analogous to the process of finding the appropriate Enquiry Expert subclass. The two interaction diagrams showing the polling process in action are closely comparable.

As soon as the Domain Spotter identifies an appropriate domain expert – either a generic object or an instance representing a specific service provider – that expert starts using the system's generic confirmation rules, which it has inherited from the Discourse Manager. These rules ensure that all attribute values supplied by the user are adequately confirmed before they are used to complete the transaction. We have already mentioned the confirmation statuses and the discourse pegs that the Discourse Manager uses to guide the system's reaction to information received from the user: only when a value is repeated by the user (or explicitly confirmed with a 'yes') is its status formally taken to be 'confirmed' and its discourse peg incremented; if the parameter is subsequently changed by the user, its status is set to 'modified', its peg is reset to zero; if negated, its status is set accordingly, and its peg set to −1. The Discourse Manager's rules are applicable right across the system, regardless of the business domain.

Once the confirmation status and the discourse peg value for each attribute has been set, further generic confirmation rules determine which attributes the system might question the user about in its next dialogue turn: according to the relative priorities of the possible actions, the system might mark a negated value as being appropriate for a repair request, or values that are new for the system as appropriate for a confirmation request in the next dialogue turn. These are the system's provisional next actions and are purely generic in nature. However, in the prototype implementation, the generic confirmation strategy, inherited by the domain experts, also provides opportunities for domain-specific rules of thumb to fire, before the system proceeds with its provisional next actions. These domain-specific rules of thumb – encapsulated in each class of domain expert – typically indicate that:

- there is enough confirmed information to select a single handling agent and allow it to immediately apply the rules of thumb to its specific service offering;
- or that there is enough confirmed information to offer the user a choice of handling agents;
- or that the user should provide certain additional information at this point to further the transaction – the rules that implement these prompts aim to capture something of the transaction-guiding expertise of a human expert;
- or that there is enough already confirmed information to complete the enquiry with a particular handling agent – again, the rules that allow the transaction to be concluded have the flavour of a human agent's decision making.

In the prototype application the above rules are implemented as suites of Template

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

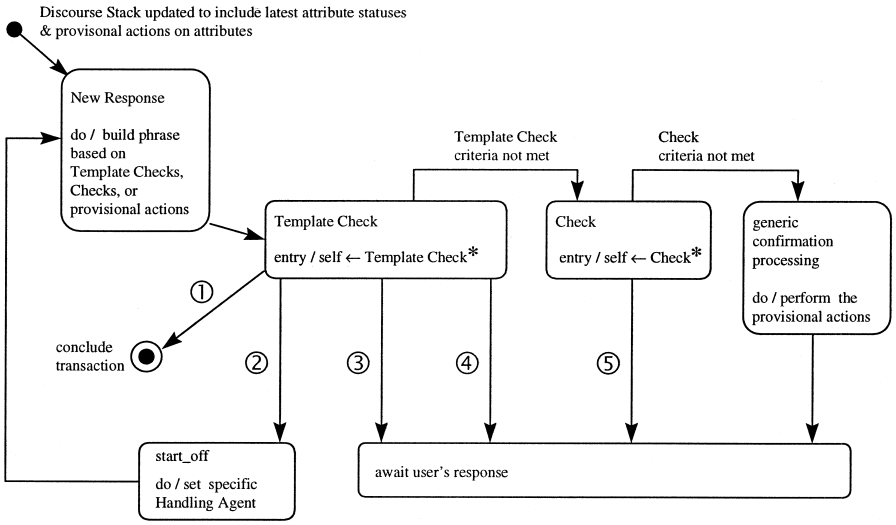
Checks in the implemented domain experts – used when the transaction is still broadly on course to a successful conclusion. However, there are also domain-specific rules to handle the situation where the transaction cannot be completed with the combination of attribute values confirmed so far by the user: in this case rules will either suggest alternative values or else advise the user that the requested service is not available, again, broadly in the manner of a human expert. In the prototype application these rules are implemented as remedial Checks.

The invocation of Template Checks and remedial Checks is the opportunity the inherited generic confirmation strategy offers the domain expert to apply its own domain-related knowledge to the dialogue management task. Depending on the type of domain expert handling the transaction, very different and highly domain-dependent dialogue control may be exercised at this point: in a travel enquiry, in which the user has supplied and confirmed a destination in accordance with the system's generic confirmation criteria, he or she might next be prompted for a day; in an events enquiry, in which the user has supplied and confirmed a movie and a date, he or she might now be offered a selection of venues. Specifically, the prompts are intended to encourage the user to provide information the system needs to complete a template, or frame, of attribute-value pairs for the particular kind of transaction. In the prototype implementation, if no domain-specific rules of thumb apply, the domain expert proceeds with the provisional next actions it had formulated previously using the Discourse Manager's generic rules alone – confirming new values, seeking confirmation of ones that have changed, and so on. Figure 9 illustrates the manner in which the Discourse Manager's generic strategy for formulating a new response is given a domain-specific implementation by the active domain expert.

The following pseudocode examples further illustrate the sorts of rules of thumb that might be applied by domain experts. In a travel booking for a single journey, if the day, time, departure point and destination are confirmed, and the transportation provider handling the enquiry offers the requested service, then according to one of the Template Checks encapsulated in the generic Travel Enquiry Expert, the system can complete the transaction:

```
IF
   (the discourse pegs for
     departure point,
     destination,
     day and
     departure time are > 0
    AND
     the Handling Agent's schedule
     includes a service for
     departure point,
     destination,
     day and
     departure time)
```

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press



** The notation '**self** ←' means that the functionality required at this stage of the processing is provided not by the generic Discourse Manager, but by the active object that has inherited and is currently implementing the Discourse Manager's functionality.*

The numbered transitions have the following conditions or conditional actions:

① *Handling Agent ≠ Anon, all required attribute values confirmed, AND value combination valid. (A Handling Agent in the prototype implementation is termed Anon if it corresponds to a generic domain expert, rather than a specific instance with access to details of a specific service offering.)*

② *Handling Agent = Anon, AND confirmed attribute values correspond to 1 Handling Agent.*

③ *Handling Agent = Anon, AND confirmed attribute values correspond to > 1 Handling Agent: get user to choose between appropriate Handling Agents.*

④ *Handling Agent ≠ Anon, AND not all required attribute values for transaction found: get user to provide most relevant missing details.*

⑤ *Typical invalid confirmed combination of attribute values: get user to provide appropriate alternative details.*

Fig. 9. Inherited Discourse Manager processing: how domain experts typically formulate a new response.

```
THEN
   instruct the Dialogue Manager
   to generate a final system utterance
   confirming a reservation for
      departure point,
      destination,
      day and
      departure time.
```

Similarly the Travel Enquiry Expert might include a remedial Check along the following lines:

```
IF  (the discourse pegs for
      departure point,
      destination,
      day and
```

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

```
      departure time are > 0
   AND
    the Handling Agent's schedule
    DOES NOT include a service for
     departure point,
     destination,
     day and
     departure time
   AND
    the Handling Agent's schedule
    includes a service for
     departure point,
     destination,
     day and
     another departure time)
THEN
   instruct the Dialogue Manager
   to generate a system utterance
   suggesting
      another departure time.
```

Rules such as these, which are generic to any Travel Enquiry Expert, are inherited by and are used by the selected 'handling instance' of Travel Enquiry Expert. It is the schedule data specific to that instance that will be used either to complete the transaction, or to start out on the process of suggesting alternatives.

The generic dialogue control functionality is such that domain-specific Template Checks or remedial Checks fire whenever particular combinations of confirmed attribute values are supplied by the user, either on the user's own initiative, or as a result of a system prompt. Likewise, the generic confirmation strategies guide the user to provide appropriate confirmations of information that the user may have provided as a result of a system prompt, or that the user may simply have volunteered, or that her or she may have provided as an alternative to the information specifically requested. The fact that the user, assumed to be a cooperative one, is effectively able to ignore system prompts makes for a relatively free-form dialogue between user and system. In this respect the dialogue management strategy adopted here resembles that described by Ward and Pellom (1999) – like the CU Communicator this dialogue management system is also 'event driven', determining its next action according to the combination of attribute values the user has supplied and their state of confirmedness.

The transactions supported by the prototype application are relatively simple ones, with relatively simple rules of thumb used to direct the system's next move. However, the fact that domain-specific rules of thumb are encapsulated within appropriate domain experts means that it would be possible to add rules for greater processing complexity without impinging on higher level control routines. For example, an instance of a Travel Enquiry Expert, given a confirmed point of

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

departure and a confirmed destination, for which it cannot provide a direct service, might have guidelines that allow it to examine its schedules to create a connecting service. The generic Travel Enquiry Expert might even contain rules allowing it to poll several of its own instances to build a itineraries from their combined service offerings. At present the system's Discourse Stack is used solely in the process of comparing current user utterances with previous discourse states in order to support the system's generic confirmation strategy. A more complex system might use this Discourse Stack as a means of allowing the user to 'change plans' by returning to a previous discourse state – or to compare and choose between alternative valid combinations of confirmed data.

8 Conclusion

In this paper we have proposed a way of looking at natural spoken dialogues which encourages the developer to distinguish highly generic dialogue functionality – turn taking, confirmation strategy – from more specialised functionality that deals with, for example, enquiry-type dialogues, and from still more specialised functionality for handling enquiries in particular business domains (travel, events). By means of inheritance, domain-specific components have access to generic dialogue functionality. When, ultimately, a 'handling agent' – an instance of a subdomain expert – takes control of a particular transaction, it has at its disposal data relevant to its own particular real-world business, plus rules that any expert in its domain would use to conduct a transaction, plus skills that any enquiry handler might use to support dialogue more generally. Should the developer decide to introduce more sophisticated generic dialogue functionality, this is immediately available to the more specialised components.

By combining the inheritance strategy with a mechanism that treats classes as generic objects capable of checking and selecting the expertise of their own instances, which in turn have their own domain- or business-specific data, the proposed architecture allows more specialised dialogue management abilities to be added in the lower levels of the system, without affecting higher-level functionality.

Automation of dialogues that address different business domains to the ones discussed here will most probably result in developers taking different decisions as to where generic functionality ends and more specialised functionality begins. Nevertheless, we believe the methodology outlined here presents a practical approach to the task of turning the processes observed in natural spoken dialogues into maintainable, extensible software systems.

References

Aberdeen, J., Bayer, S., Caskey, S., Damianos, L., Goldschen, A., Hirschman, L., Loehr, D. and Trappe, H. (1999) Implementing practical dialogue systems with the DARPA Communicator Architecture. *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI '99): Workshop NLP-2, Knowledge and Reasoning in Practical Dialogue Systems*, Stockholm, Sweden, pp. 81–86.

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press

Allen, J. F., Miller, B. W., Ringger, E. K. and Sikorski, T. (1996) A robust system for natural spoken dialogue. *Proc. of the 34th Annual Meeting of the ACL*, pp. 62–70. San Francisco: Morgan Kaufmann.

Aust, H. and Oerder, M. (1995) Dialogue control in automatic enquiry systems. *ESCA Workshop on Spoken Dialogue Systems*, Vigso, Denmark, pp. 121–124.

Aust, H., Oerder, M., Seide, F. and Steinbiss, V. (1995) The Philips automatic train timetable information system. *Speech Communication*, **17**: 249–262.

Bernsen, N. O., Dybkjaer, H. and Dybkjaer, L. (1998) *Designing Interactive Systems: From First Ideas to User Testing*. London: Springer-Verlag.

Booch, G., Rumbaugh, J. and Jacobson, I. (1998) *The Unified Modeling Language User Guide*. Reading, MA.: Addison-Wesley Longman.

Heisterkamp, P. and McGlashan, S. (1996) Units of dialogue management: An example. *ICSLP96 – Proceedings of the Fourth International Conference on Spoken Language Processing*, Philadelphia, PA, pp. 200–203.

MITRE Corporation (1999) DARPA Communicator Web Page. `http://fofoca.mitre.org/`

Novick, D. G. and Sutton, S. (1996) Building on experience: Managing spoken interaction through library subdialogues. *Proc. of TWLT 11 – Dialogue Management in Natural Language Systems*, pp. 51–60, University of Twente, The Netherlands.

O'Neill, I. M. and McTear, M. F. (1999) An object-oriented approach to the design of dialogue management functionality. *Proc. of the 9th Conference of The European Chapter of the Association for Computational Linguistics (EACL '99)*, pp. 23–29, University of Bergen, Norway.

Wang, K. (1998) An event-driven model for dialogue systems. *ICSLP98 – Proc. of the Fifth International Conference on Spoken Language Processing*, pp. 393–396. Sydney, Australia.

Ward, W. and Pellom, B. (1999) The CU Communicator System. *Proc. of the IEEE Workshop on Automatic Speech Recognition and Understanding*, Keystone, Colorado.

https://doi.org/10.1017/S1351324900002527 Published online by Cambridge University Press