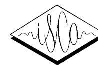


The Queen's Communicator: An Object-Oriented Dialogue Manager

Ian O'Neill, Philip Hanna*, Xingkun Liu*, Michael McTear°*

*School of Computer Science
Queen's University, Belfast, N. Ireland
{i.oneill,p.hanna,xingkun.liu}@qub.ac.uk

°School of Computing and Mathematics
University of Ulster, Jordanstown, N. Ireland
mf.mctear@ulster.ac.uk

Abstract

This paper presents some of the main features of a prototype spoken dialogue manager (DM) that has been incorporated into the DARPA Communicator architecture. Developed in Java, the object components that constitute the DM separate generic from domain-specific dialogue behaviour in the interests of maintainability and extensibility. Confirmation strategies encapsulated in a high-level DiscourseManager determine the system's behaviour across transactional domains, while rules of thumb encapsulated in a suite of domain experts enable the system to guide the user towards completion of particular transactions. We describe the nature of the generic confirmation strategy and the domain experts' specialised dialogue behaviour. We describe how rules of thumb fire given certain combinations of user-supplied values – or in the light of the system's own interaction with its database.

1. Introduction

The aim of the current research is to explore the manner in which mainstream object-oriented development techniques might be used to create a spoken dialogue manager (DM) that encapsulates generic and domain-specific dialogue management strategies. Implemented in Java, the DM receives semantically tagged user input via the Phoenix Semantic Frame Parser [1] and generates output using the Festival speech synthesiser [2]. The DM communicates with these and other service providers within the DARPA Communicator architecture [3], based on the Galaxy hub, a software router developed by the Spoken Language Systems group at MIT [4] and subsequently released as an open source package in collaboration with the MITRE Corporation [5]. Our working 'testbed' application is based on the components supplied with the CU Communicator [6], from which we have removed the dialogue management components and replaced them with components of our own. At present the system accepts keyed natural language input via the parser and outputs key phrases rather than well-formed sentences to the speech synthesiser: a fully implemented speech user interface is planned.

The present Java implementation is based broadly on a Prolog++ prototype [7, 8] which was used to explore the relationship between generic confirmation strategies and domain-specific heuristics for furthering transactions. By using suites of domain-specific dialogue-furthering heuristics – coded declaratively and then parsed – the Java implementation has captured much of the intuitive programming style of its Prolog predecessor. Moreover, these heuristics or 'rules of thumb' have now been expanded to encompass not only the interaction between the system and the user, but also the interaction between the dialogue manager and a database. Thus, there are two flavours of 'rules of thumb'. On the one hand *user-focussed rules* determine the system's response to particular combinations of data supplied by the user: the system might attempt a database lookup or ask the user for more information. On the other hand *database-focussed rules* guide the system as to how it should attempt an alternative database request when the user has provided an invalid combination of data values. In each case generic behaviour determines how the system's domain-specific determinations are conveyed to the user.

2. Architecture

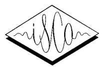
2.1. Overview

Although some currently available dialogue systems use object components in accordance with the latest software engineering orthodoxy [9], little published research addresses the question of how established techniques of object-oriented software engineering [10, 11] can contribute to the dialogue management task. It is hoped that our OO approach to spoken dialogue management will provide a framework within which generic confirmation strategies and rules-of-thumb specific to particular business domains can be intuitively maintained and extended. *Figure1* shows some of the key components of the Java DM, as well as the inheritance hierarchy.

2.1.1. DialogServer, DialogManager and DomainSpotter

DialogServer provides an interface to the hub. It contains a DialogManager, which as well as co-ordinating dialogue turn-taking, has a suite of business domain experts (AccommodationExpert is one example). These domain experts are grandchildren and great-granchildren of DiscourseManager below.

DialogueManager also has a DomainSpotter that helps select domain expertise, and a DiscourseHistory that maintains a record of user-system interaction across domains. The DomainSpotter supplies each domain expert with the output of the semantic parse. Each expert scores that parse against the semantic categories that it can process and returns the score to the DomainSpotter. The domain expert that



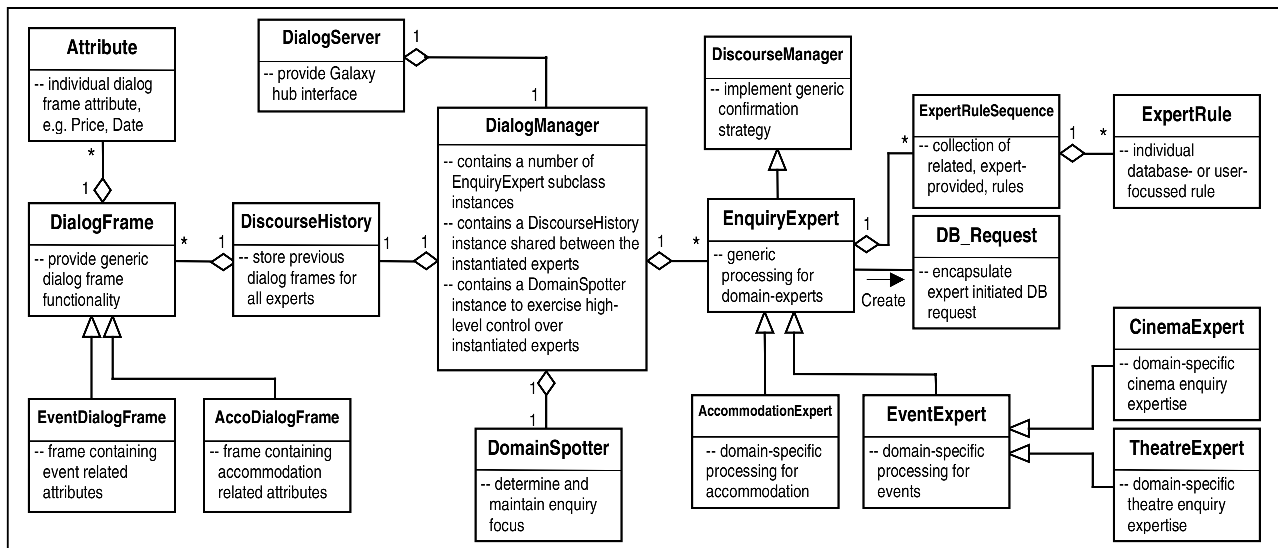


Figure 1: High level UML class diagram for the Java dialogue manager

scores highest will be the one that the DialogManager will ask to apply its domain-specific heuristics to the more detailed processing of the enquiry. For example, an AccommodationExpert might score highest and so become handling expert if the user has been asking about hotels in Belfast. The 'enquiry focus' will remain with this handling expert, until the parsed input indicates that another domain expert would make a more appropriate handling expert.

In a variation on this approach we are now implementing functionality that will allow expert superclasses – e.g. EnquiryExpert – to poll their subclasses and have them supply a natural language phrase that conveys their area of expertise, information that can be used to direct the user when he or she has made a very vague or ambiguous enquiry that could potentially fall to a number of handling experts. We believe this will simplify the process of adding new expertise to the system.

2.1.2. DiscourseManager, EnquiryExpert and subclasses

The DiscourseManager is responsible for the DM's generic discourse behaviour. It determines the system's response to new, modified or negated information from the user, and it determines when the domain-specific rules of thumb, encapsulated in the suite of domain experts, should be allowed to fire. The system's utterances typically take the form of an implicit confirmation of *new* information supplied by the user (in a fully generated form the system utterance might be "So, I've got you staying at the Hilton in Belfast from June 20th -") followed immediately by the system's next question ("- what day will you be leaving?"). If the user has *modified* or *negated* information that the system had previously recorded, the system's next utterance concentrates on correcting the modified or negated information rather than seeking further information. The DiscourseManager is at the top of the inheritance hierarchy. Its behaviour therefore colours the manner in which its grandchildren and great-granchildren (EnquiryExpert subclasses like AccommodationExpert and TheatreExpert) interact with the user in their own domains (accommodation, events, etc.).

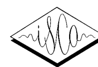
The DiscourseManager is able to work out what has been repeated, modified or negated by the user by comparing a frame of information relating to the user's latest utterance, with a corresponding frame representing the last discourse state. This last discourse state indicates what information had been provided by the user, the status of each piece of information (repeated, modified, negated, etc.) and the system's previous intentions for confirming or repairing supplied or missing information. Using the last discourse state the system is therefore able to interpret the user's latest utterance in the light of the intention behind its own last utterance (e.g. if the user does not attempt to modify the information conveyed by the system's implicit confirmation "So, I've got you staying at the Hilton…", then the Hilton can (unless it is subsequently modified) be regarded as the confirmed accommodation name.

The DiscourseManager makes use of a number of other components. In order to let each of its domain-specific EnquiryExpert subclasses update the record of the evolving dialogue, it takes on the DialogManager's DiscourseHistory as an inheritable attribute of its own. DiscourseHistory maintains a record of the evolving dialogue in the form of a stack of DialogFrames, each of which in turn comprises a set of Attribute objects relevant to the particular business domain. EnquiryExpert and its subclasses represent the domain-specific expertise that augment the behaviour of DiscourseManager once the latter's generic confirmation strategies have been applied.

2.1.3. Discourse History, Dialogue Frame, Attribute

The DiscourseHistory is a stack of DialogFrames and contains methods that assist the DiscourseManager in adding frames to and retrieving frames from the stack

The DialogFrame is a set of attributes (of class Attribute – more about this presently) that corresponds to the frame of slots that must typically be filled to complete a transaction in a particular domain - events or accommodation, say. The generic DialogFrame has methods that are not domain-specific and that enable calling objects to (among other things) *addAttribute* and *getAttribute*. Specialisations of DialogFrame are initialised with attributes relevant to a particular enquiry type: for example a frame for an accommodation booking (AccommodationDialogFrame)



might include the attributes accommodation type, date from, date to, etc. We tag all instances of a frame of a particular type with a distinctive *identifier* – e.g. "Accommodation" for an AccommodationDialogFrame. This gives us the option of using DiscourseHistory's method *getLastMatchingFrame* to retrieve a frame that furthers a particular discourse strand (an accommodation enquiry, say), from among other types of frames in the DiscourseHistory's stack. These other frames may be generated if, for example, the user enquires about going to a show in the course of an accommodation enquiry.

Each object of class Attribute within a DialogFrame comprises a number of data values – attributeName, attributeValue, confirmationStatus (*modified_by_user*, etc), discoursePeg (incremented as the value is repeatedly confirmed by the user, reset to zero when modified etc.) systemIntention (*repair_confirm*, etc.) – which collectively inform the system of the status of each piece of information that will be used to complete the transaction. Here object-orientation is being used to create a multi-facetted view of each piece of information being considered by the system.

2.2. The domain experts' heuristics

Terminating the inheritance hierarchy are the domain experts: AccommodationExpert, EventExpert, CinemaExpert, TheatreExpert, etc. (CimemaExpert and TheatreExpert are children of EventExpert and represent further specialisations of their parent's event-handling expertise.) These experts contain a battery of domain-specific rules that enable them to respond appropriately to the user, given that the user has supplied a particular combination of confirmed attribute values. The behaviour inherited from the DiscourseManager ensures that domain experts confirm or query information appropriately, before assuming that it has been supplied and recognised correctly. Only when information has been (at least implicitly) confirmed is it used to trigger the handling domain expert's heuristics, expressed as sets of transaction rules. Provision has been made within the object hierarchy to allow rules that are more domain-specific to fire first and rules that are more generic to be tried next in the case where the object hierarchy is extended below the first level of domain experts.

The transaction rules encapsulated in the domain experts fall into two main sequences:

- *user-focussed rules*: rules that are used to trigger the system's immediate response to specific confirmed combinations of information supplied by the user and recorded in the evolving dialogue frame – the rules may cause the system to ask for more information, or may initiate a database search.

  e.g. IF (the user has not given
  accommodation name [e.g. 'Hilton']
  or *accommodation type* [e.g. 'Hotel'])
  THEN ask for *accommodation type* (1)

- *database-focussed rules*: rules that are applied in the light of the system's failed attempts to retrieve information from or validate information against the database. These failed searches may result from a particular combination of search constraints, whether these are supplied by the user, or by the system when it attempts to retrieve information to assist the user. The database-focussed rules may therefore recommend that a constraint (e.g. the class of hotel) be relaxed in order to get a database match for other user requirements (e.g. the hotel location that the user has requested).

  e.g. IF (failed search was to find *accommodation name*
  [e.g. Hilton, Holiday Inn, etc.]
  AND constraints were *location Belfast* and *class four-star* and *accommodation type hotel*)
  THEN relax constraint *class four-star* and re-do search (2)

The database-focussed rules represent recovery strategies that enable the system to offer viable alternatives when an enquiry might otherwise reach an impasse. The user remains free to reformulate the enquiry in a way that differs from the system's suggestion; indeed, in circumstances where the system has no specific recommendation to make, the system will simply explain why the database search has failed and pass the initiative back to the user.

The user-focussed and database-focussed rules that are encapsulated in the domain experts are representative of the kinds of decision making that characterise a human expert in the particular domain – a booking clerk at a theatre, or a desk clerk at an hotel. We intend to refine the rules in the light of more detailed studies of interactions between human enquirers and human agents. For example, in (2) above, it might on occasion be preferable to search for a different hotel *location*, while maintaining the *class* constraint.

Within each domain expert, each rule is specified declaratively. For example, (1) above appears as

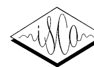
```
String userFocussedRule1 = "
[RULE]
    { AccoName UNSPECIFIED }
    { AccoType UNSPECIFIED }
[ACTION]
    { INTENTION AccoType SPECIFY }
[RULE-END]";                                  (3)
```

while (2) above appears as

```
String dbFocussedRule1 = "
[RULE]
    { AccoName TARGET }
    { AccoType CONSTRAINING }
    { Location CONSTRAINING }
    { AccoClass CONSTRAINING }
[ACTION]
    { RELAX \" AccoClass \" }
[RULE-END]";                                  (4)
```

Specifying rules declaratively in this manner recreates some of the intuitiveness of rule-based programming – the suite of rules can be easily extended or reduced to capture the subtlety of human behaviour. In creating the rules the developer is not so much concerned with how the behaviour will be implemented as with what the behaviour should be.

However, implementing the behaviour needs to be addressed somewhere. The rule specifications are used as parameters for building ExpertRule objects, which contain methods for extracting and analysing the contents of the rule, and these rule objects are in turn built into ExpertRuleSequence objects (typically, for each domain expert, there will be a sequence for user-focussed rules and another for database-focussed rules). Each instance of EnquiryExpert (whether an AccommodationExpert, an



EventExpert or a still more specialised subclass) is permitted by the generic confirmation strategy to test its rule sequences when there are no user-initiated modifications or negations to be addressed. A user-focussed rule may thus cause a SPECIFY intention to be set against an attribute in a dialogue frame, or it may initiate a database search, and if this search fails to return the value(s) sought, the query may be resubmitted in amended form in accordance with the expert's database-focussed rules. System output is currently in the form of key phrases – so an implicit confirmation followed by a SPECIFY intention might be output as: "Implicit_Confirm AccoType = Hotel; Specify Location." We intend to develop a natural language generation module that will accept this or similar semantic output and generate a well-formed utterance.

2.3. Some new generic behaviour

In order to deal with the novel situation of the domain expert using its database-focussed rules to reformulate database queries that were originally composed on the basis of the user's confirmed utterances, we have had to create two new generic confirmation statuses to extend the set originally proposed by Heisterkamp and McGlashan [12]. Thus, alongside *new for system*, *inferred by system*, *repeated by user*, *modified by user* and *negated by user*, we have added *modified by system* and *negated by system* – to deal with situations where the system, after running a modified or 'relaxed' database query, has found, respectively, one or several alternative values to the failed constraint supplied by the user. The generic confirmation strategy encapsulated in the DiscourseManager uses an enhanced set of *evolve* rules, extended from the original set described in [7], to set these statuses. The system's response to the user must now address the possibility that a domain expert may have negated a user value but found no alternatives, negated a value and found several alternatives or modified an invalid constraint to a valid one. Accordingly system intentions now include requests for the user to reformulate the enquiry, choose one of the suggested alternative constraints, or explicitly confirm a constraint value that the system has modified.

2.4. Working with the Galaxy hub

A further element of the object-oriented solution is the means by which the DialogueManager communicates with the database server via the Galaxy hub. Whenever an EnquiryExpert subclass needs to make a database search, it creates a DB_Request object whose attributes record which values are sought, which search constraints are to be used for the database search, and which constraints have been relaxed (i.e. require new values). The object must pass between two servers (going from the DialogServer to the DatabaseServer and back again) via the Galaxy hub. The DB_Request class therefore includes the encoding and decoding functionality that allows its instances to be encapsulated at the DialogServer as a bitstream within a Galaxy hubframe and reconstituted at a receiving DatabaseServer as an object. The contents of the DB_Request object are then used to formulate an SQL database query. The DB_Request object is populated with the results of the database search. It is encoded again and sent back via the Galaxy hub to the dialog manager where it is reconstituted and passed back to the domain expert that initiated the search. The domain expert can then apply its rules of thumb to the data in the DB_Request object.

3. Conclusions

In creating our prototype dialogue manager in Java we have continued to explore the possibility of separating generic from domain-specific dialogue behaviour. A range of dialogue components are now represented as objects, with data content and methods to manipulate those data. The objects within our system include data items supplied by a user and tagged with confirmation status and system intention; enquiry-specific frames of information in an evolving discourse history; domain experts with their own agent-like behaviours; and high level, inheritable confirmation strategies. As well as extending our range of domain experts to cover other areas of expertise (e.g. travel enquiries), we will be expanding our parser grammars to support the more comprehensive phrase-spotting required for free-form spoken input. For spoken output we will be exploring means of converting the system's current output concepts into well-formed natural language utterances.

4. Acknowledgements

This research is supported by the EPSRC under grant number GR/R91632/01.

5. References

[1] Ward, W., "Understanding Spontaneous Speech: the Phoenix System", *Proceedings of the International Conference on Audio, Speech and Signal Processing (ICASSP)*, 365-367, 1991.

[2] http://www.cstr.ed.ac.uk/projects/festival/

[3] http://www.darpa.mil/iao/communicator.htm

[4] http://www.sls.lcs.mit.edu/sls/technologies/galaxy.shtml

[5] http://fofoca.mitre.org/

[6] http://communicator.colorado.edu

[7] O'Neill, I.M., McTear, M.F., "A Pragmatic Confirmation Mechanism for an Object-Based Spoken Dialogue Manager", *Proceedings of ICSLP-2002*, Vol. 3, 2045-2048. Denver, September 2002.

[8] O'Neill, I.M., McTear, M.F., "Object-Oriented Modelling of Spoken Language Dialogue Systems", *Natural Language Engineering* 6 (3-4), 341-362, Cambridge University Press, 2000.

[9] Allen, J., Byron, D. Dzikovska, M., Ferguson, F., Galescu, L. and Stent, A., "An Architecture for a Generic Dialogue Shell", *Natural Language Engineering* 6 (3-4), 1-16, Cambridge University Press, 2000.

[10] Booch, G., *Object-Oriented Analysis and Design with Applications* (2nd Edition). Benjamin/Cummings, Redwood City, CA, 1994.

[11] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley Longman, Reading, MA, 1998.

[12] Heisterkamp, P. and McGlashan, S. "Units of Dialogue Management: An Example", *Proceedings of ICSLP96*, 200-203, Philadelphia, 1996.