A Method for Iterative Implementation of Dialogue Management

Lars Degerstedt and Arne Jönsson
Department of Computer and Information Science
Linköping University, Sweden
larde@ida.liu.se, arnjo@ida.liu.se

Abstract

This paper presents an approach to implementation of dialogue management modules for dialogue systems. The implementation method is divided into two distinct, but correlated, steps; Conceptual design and Framework customisation. Conceptual design and framework customisation are two mutually dependent sides of the same phenomena, where the former is an on-paper activity that results in a design document and the latter results in the actual program code. The method is iterative and conforms with software development methods, such as, extreme programming, scenario-based design and reusable object-oriented software development. In the paper, this is further elaborated and how it relates to dialogue systems development.

1 Introduction

Implementation of dialogue systems for new applications could be viewed as a process of customising a generic framework to fit the needs of a more specific application. For fairly simple applications this can be carried out using predefined building blocks, e.g. CSLU [McTear, 1999]. However, although we conform to *The Practical Dialogue Hypothesis* [Allen *et al.*, 2001], e.g. that conversational competence for practical dialogues is significantly simpler to achieve than general human conversation, realisation of more advanced dialogue systems still involves substantial work, cf. [McRoy and Ali, 2001].

In this paper we will outline a method for the implementation of dialogue management, (DM), modules[1] for dialogue systems, (DS), from generic frameworks, such as, TRIPS [Allen *et al.*, 2000], and TRINDIKIT [Larsson *et al.*, 2001]. We see the contribution of our implementation method as a step towards a tool for building dialogue systems that can be "adapted to each new task relatively easy" [Allen *et al.*, 2001]. However, this vision rests on *The Domain-Independence Hypothesis* [Allen *et al.*, 2001], e.g. that the complexity in the language interpretation and dialogue is independent of the task being performed, which still needs to be verified.

In Section 2 we briefly present our own experience from developing dialogue systems which also motivated our need for a more systematic method. Section 3 presents an overview of the implementation method. One novelty is the separation of the conceptual design from framework customisation, discussed in Section 4. Our aim is to formulate a working method that supports the creative process rather than giving a set of mechanical rules. In Section 5 we introduce a set of capabilities that is used to organise the implementation work of the dialogue management module. These capabilities are discussed subsequently in Section 6–8. In Section 9 we briefly present initial experience from using the method and discuss, in Section 10, the method as a software development method.

2 Background

The method is based on our experience from developing dialogue systems for various applications and purposes. The need for a framework and corresponding development method emanates from a re-implementation of our lisp-based dialogue system, CARS, to Java. CARS is a typed interaction only natural language dialogue system developed from the LINLIN dialogue model[2] [Jönsson, 1997] to an application allowing fairly simple information requests from an SQL-database. CARS utilised a parser tool, but had no framework for dialogue management.

The CARS dialogue system served as inspiration for the MALIN[3] [Dahlbäck *et al.*, 1999] framework. MALIN was developed as a framework by generalising from the development of CARS to a system for local bus time table information, ÖTRAF. Time table information systems require more complex requests as the system needs to handle various tasks, not only simple information retrieval and, thus, LINLIN, was modified to account for this, and more. The development of

[1]We will not use the term Dialogue Manager as it, in this paper, is important to distinguish between the running module in the customised dialogue system and the generic framework from which it is developed. We will instead use the terms DM module and DM framework to denote the running module and the framework to be customised respectively.

[2]By *model* we understand a formal or informal theoretical analysis of dialogue. Such a model can be present in a system either implicit as a *basis for the system design*, or explicit as a *knowledge model (*KM*)* e.g. a grammar.

[3]Multi-modal application of LINLIN.

ÖTRAF was the starting point for our need for methods for implementation of dialogue system modules, but the method was not really used in that project.

The method has, however, been utilised in two other projects, SCIN and a project on developing a natural language interface for a digital TV box. SCIN is an interactive news reader with amongst other things a talking head. SCIN will be based on the MALIN framework. The iterative nature of the method presented in this paper is, however, more prominent in that work. SCIN is developed in steps, where a simple interface will be iteratively refined by adding more and more functionality, see Section 9. The TV box project instead customised the MALIN framework into a working system by iteratively refining the knowledge sources. Little added functionality was needed as the application very much resembles the previously developed CARS dialogue system.

Both the TV box system and SCIN was developed iteratively from prototypes that were evaluated and gradually refined, cf. [Hulstijn, 2000]. Such evaluations need not necessarily be carried out with end users, especially not the first set of prototypes.

It is our aim that the method should give concrete support as concerns both attitude and working steps. This boils down to the following contributions concerning implementation of a dialogue management module in this paper:

- a simple and natural implementation work chart.
- a prototypical work flow schema from prerequisites to final design and framework customisation.
- guidelines for application-specific points for the implementation work.

3 Method Overview

The implementation of a dialogue management (DM) module should be done with specific means but yet conform to a generic working schema to be in harmony with realisation of other modules. Through iterative thinking the realisation process can be divided into more manageable pieces (cf. [Krutchen, 2000]). Through successive refinements and incremental development, the solution can be reached gradually, as the understanding of the problem increases, in an evolutionary manner. Our method suggests to work iteratively from the two angles **conceptual design** and **framework customisation**. Conceptual design and framework customisation are seen as two mutually dependent aspects of the same phenomena. Thus, just as advocates of extreme programming [Beck, 2000], we view coding and design as a joint activity.

Prerequisites for the DM design is a requirements specification of the dialogue system and selected dialogue theory.

The framework customisation starts out from a selected framework (i.e. some development environment and tools) and (some version of) the other modules of the dialogue system. Figure 1 shows the twofold character of **one iteration**. The solid lines depict the creative progression step. The dotted lines show, point-by-point, how the two levels of the iteration correlate.

At the level of the iterative scheme, our method seems fairly module independent. However, it is our final aim to be as concrete as possible w.r.t. the implementation of a DM module. Therefore, we introduce an auxiliary notion of more refined and DM dependent steps, called DM **capability steps**, within each iteration. These steps are centered around a set of prototypical DM capabilities that we introduce – a set that can be updated, refined or changed when needed. In this way, the method is generic but can be adjusted to fit the DM module at hand.

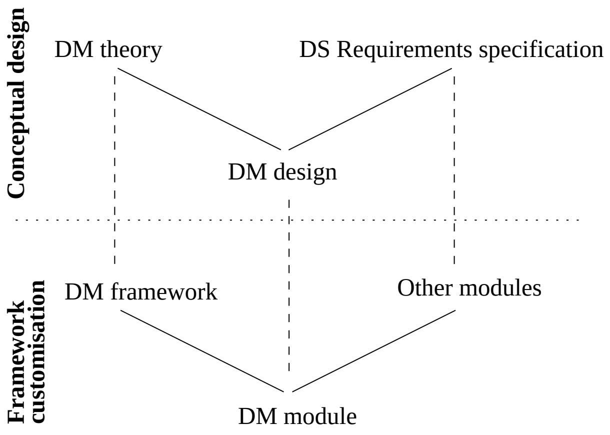

Figure 1: The Dialogue Management (DM) iteration

3.1 Method Prerequisites

The suggested implementation method is open for adjustments of its prerequisites within a particular project. The selection of DM theory and DM framework chiefly affects the representation of the dialogue and how its associated functionality is attached. The choice is based partly on previous experience and partly on applicability.

System requirements is mainly acquired using suitable empirical methods such as WOZ experiments and dialogue distillations [Jönsson and Dahlbäck, 2000], and guidelines (cf. [DISC, 1999]). By analysis of the material we may formulate the requirements specification. We suggest to include the following two parts, as a minimum, in the specification:

- classification of possible dialogues and identification of the main **use-cases**.
- specification of the **system behaviour** in terms of the selected DM theory, for each identified class of dialogues.

Use-case-based development (cf. [Carroll, 1995]) fits nicely with the design of multi-modal dialogue systems using our iterative implementation method. In our case the use-cases consist of selected central examples of user-system dialogues. These dialogues serve as the starting point of the transition from DS requirements to DM design. The selected dialogues can be refined iteratively during the implementation as our understanding of the system increases.

3.2 Our DM Framework

We mainly use concepts from the LINLIN model together with the MALIN-DM[4] dialogue management framework. It is not our intention to describe the LINLIN dialogue model or the MALIN framework. However, to illustrate our method we will use concepts and notations from the LINLIN dialogue model upon which the MALIN-DM framework is based.

In the LINLIN dialogue model the dialogue is structured in terms of discourse segments, and a discourse segment in terms of moves and embedded segments. Utterances are analysed as linguistic objects which function as vehicles for atomic move segments. An initiative-response (IR) structure determines the compound discourse segments, where an initiative opens the IR-segment by introducing a new goal and the response closes the IR-segment. The LINLIN dialogue model classifies the discourse segments by general speech act categories, such as *question* (Q) and *answer* (A), rather than specialised (cf. [Hagen, 1999]), or domain related [Alexandersson and Reithinger, 1995].

The dialogue segments form a dialogue tree. The nodes in the tree, termed dialogue objects, hold information such as the current objects and properties, the user request in focus, information on speaker, hearer, type of general speech act, etc. The LINLIN dialogue tree is naturally specified in terms of a grammar. The MALIN-DM framework expects such a grammar for the top-level control of the DM module. The system design is thereby kept in close relation with the empirically developed models. The grammar is represented in a formalism for so-called *dialogue grammars* using a Java-based DM tool called MALIN-DG.

The model assumes that decisions as to what to do next are made on the basis of focus information (cf. [Leceuche *et al.*, 2000]), i.e. depending on how the focal parameters have been specified. Focus information can be copied between nodes in the dialogue tree, either horisontally by *focus inheritance*, from one IR-segment to the next, or vertically by *answer integration* to handle sub-dialogues.

4 The Twofold DM Iteration

Design and customisation of the DM module are performed by point-wise connecting conceptual issues with those of the selected DM framework. Conceptual DM design is an on-paper activity that results in a **design document**. The result of the DM framework customisation is the actual **module code**. At the end of each iteration we expect to have a readable version of the DM design document and a runnable DM module prototype.

The DM design constructs must be effectively realisable within the selected framework. The DM framework customisation should strive for a visible connection to the design. Moreover, the design and customisation should strive for a visible connection to the DM theory and DS requirements from which they start.

The remains of this section contains a subdivision of both the design and customisation that describes the results of the

[4]The MALIN-DM framework is a sub-framework of the MALIN framework.

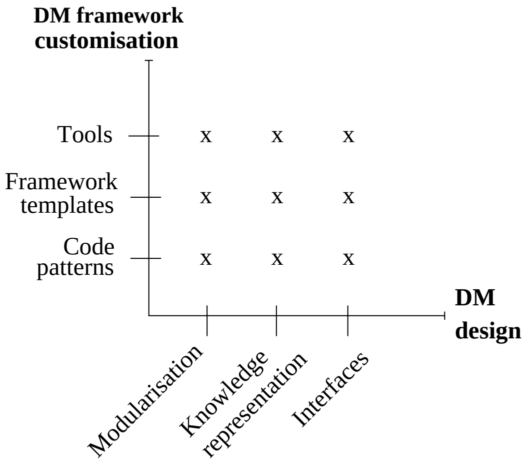

Figure 2: Development space for the twofold DM iteration

twofold implementation iteration. We view these subdivisions as two orthogonal parameters which forms the space to explore during the implementation process, as shown in Figure 2. Each tuple within this space represents a special sub-issue that should be tackled during the realisation of the DM module, e.g. how to structure the dialogue grammar using the MALIN-DG tool at the tuple of tools and knowledge representation.

4.1 DM Design

The DM design is suggested to focus on the representation and flow of information in the DM module. The DM design document is recommended to be relatively brief, since its contents will be iteratively refined. There is no need to put down near-coding information on paper since the DM module is built in parallel with the design. However, a mature version of the DM design preferably establishes some notion of correctness and completeness. This is typically done by identifying a set of expected user-system dialogues and verifying that the designed system behaves in accordance with the requirements specification.

The finished design document for the DM module should normally include discussions on:

- DM **modularisation**: identification of central sub-units of the DM module and definition of their responsibilities. Submodules are suggested to be identified on three levels: control, handlers and methods. For the MALIN-DM framework, examples from these levels are the description of the dialogue tree construction process, the focus inheritance algorithms, and a method for background system access, respectively.
- DM **knowledge representation**: identification and abstract formulation of data items for the DM module. The formulation is preferably kept in formal or semi-formal terms and based on the selected use-cases. With the MALIN-DM framework, an example of such formulation is the representation of the user request in focus.
- DM **interfaces**: formulation of interface functionality

and (sub)module dependencies that defines the central data flows of the DM module, both internally and towards other modules. For the MALIN-DM framework, this includes definitions of the formats for interaction with other modules, such as the parsing module and the domain knowledge module.

That is, we suggest a light-weight design document where knowledge representation has been emphasized and issues related to flow of control de-emphasized. This is motivated by the knowledge intensive character of a DS interface and its lack of complex subsystem cooperation.

4.2 DM Framework Customisation

Coding of a DM module starts off from the selected DM framework. The DM module is created iteratively by various customisation steps. We distinguish between three forms of re-use from a DM framework that complement each other:

- DM **tools**: customisation through well-defined parameter settings and data representation files. The main tool in the MALIN-DM framework is the MALIN-DG compiler.
- DM **framework templates**: framework templates and application-specific code are kept in separate code trees with clear-cut borders and only one-way module dependencies from framework to application (cf. [Fayad *et al.*, 1999]). A central example of such a template in the MALIN-DM framework is the domain-independent taxonomy of Java classes for dialogue objects. The hierarchy can be further instantiated at application level, if needed.
- DM **code patterns**: submodules, edited code patterns and other forms of textual re-use (cf. [Gamma *et al.*, 1995]). For instance, code patterns are useful in more domain-dependent parts of request handling and focus inheritance, with the current version of the MALIN-DM framework.

Tools are a strong form of re-use but often limited in scope and flexibility. The use of framework templates is typically a more complex process but offers more support for construction of a complete application. A code pattern is the weakest form of re-use. It yields an important prototypical code or data skeleton to work from.

A tool that introduces a new formalism pays off for conceptually important notions only. The gain of using a new representation language lies mainly in increased conceptual clarity. For the MALIN-DM framework, the dialogue tree and the request formats are examples of such clarifying concepts. Framework templates are useful for more system-related tasks, such as module communication. They are the glue between the DM tools and the underlying system. In the MALIN-DM framework, this is the case with, for instance, templates for incoming messages from the interpretation module and handling of user-threads. A code pattern is useful for principal parameters of the DM module, i.e. application-dependent pieces of data or code. For the MALIN-DM framework, examples of principal parameters are the dialogue grammar and the representation of focused objects and properties.

The overall aim of the customisation process is to increase re-use during implementation of DM modules for different DS applications (cf. [Karlsson, 1995]). Clear-cut borders between the DM framework and the DM module is crucial for a high degree of re-use. A sign of successful customisation is a clear separation of the directory trees for the DM framework and the DM module in the final system.

5 Our View of DM Capabilities

We define a set of DM capabilities that further instantiate the method scheme of Figure 2. The identified DM capabilities should be viewed as notions of **module requirements** rather than design concepts. That is, these capabilities are only loosely related to system design choices such as the choice of DM architecture. The capabilities are intended to support the organisation of specification properties for a DM module, not to give the final implementation automatically. We have, so far, found it sufficient to focus the design and coding around the following fairly general types of DM capabilities:

- **dialogue history modelling**[5]: the DM module creates and holds a knowledge model (KM) that represents the dialogue history for the current user session. The dialogue history KM is accessible both internally in DM and externally through search interfaces. In the MALIN-DM framework it is the dialogue tree that constitutes the dialogue history KM.
- **user request handling**: the DM module groups and interprets each user action as a user request and coordinates the corresponding system task. The MALIN-DM framework contains a collection of request handlers for this purpose. The request handling of the MALIN-DM framework include formats for: query-answering, slot-filler-request, and command-execution.
- **sub-dialogue control**: the DM module adjusts the dialogue strategy at each dialogue state. In particular, there is a choice whether to proceed with a user-initiative or a system-initiative sub-dialogue. In the MALIN-DM framework the top-level control of the sub-dialogue structure is contained in the dialogue grammar.

This distinction divides the implementation process into three work steps. The work steps focus on a high-level picture of the dialogue management trajectory, specific DM functionalities, and the central DM sub-processes, respectively.

We suggest to organise the implementation work mainly from the perspective of these DM capabilities. For each capability it is suggested to solve the related DS specification requirements from the two viewpoints of design and customisation, as given by the tuples of Figure 2. The DM capabilities can be seen as a third, more DM specific, dimension that is orthogonal with the design and customisation dimensions of Figure 2.

[5]By the term *dialogue history* we understand the interplay between the user and system in terms of user utterances and system responses. That is, the term denotes the actual phenomena, and not a knowledge model.

The DM capability steps split the iterative implementation schema into more manageable pieces. Each such DM capability step constitutes a work flow step during an iteration or a use-case realisation. Subsequently, in Section 6–8, we elaborate on these capability steps, one by one, from our dual perspective of design and framework customisation.

6 The Dialogue History Modelling Step

The work step for dialogue history modelling concerns identifying a set of tokens and their definitions that reflects the DM interpretation of the on-going dialogue between user and system. Taxonomies of such tokens are normally given by generic categories of the DM framework and DM theory that need to be further instantiated to be useful for the application at hand.

Typically a dialogue history KM includes tokens at the following levels:

- **sub-dialogue**. A non-terminal category symbol of the dialogue grammar is an example of a sub-dialogue token from the MALIN-DM framework.
- **utterance**. A terminal category symbol of the dialogue grammar is an example of an utterance token from the MALIN-DM framework.
- **part of utterance**: A focal parameter for a dialogue object is an example of a token that represents a part of an utterance in the MALIN-DM framework.

Dialogue history modelling determines the overall structure of the DM module. Thus, modelling of the dialogue history is normally important in the beginning of the implementation process.

The dual implementation step of dialogue history modelling is a combination of conceptual data modelling and abstract datatype definition. The DM design is naturally focused around the use-cases from the requirements specification. The DM framework customisation is mainly concerned with value domains for the generic parameters of the DM framework.

6.1 Dialogue Knowledge Model Design

Knowledge representation is the main issue in DM design for dialogue history modelling. Each phenomena that occurs in the dialogue should be placed at one of the levels of sub-dialogue, utterance or part of utterance. For the MALIN-DM framework, for instance, issues of filling a complex request structure occur at the level of the sub-dialogue; issues of classifying a user sentence according to the LINLIN speech acts occur at the level of the utterance; issues of how to represent the focal content of a user phrase occur at the level of the part of utterance.

Modularisation issues for the dialogue history mainly concerns the separation of different interpretation layers of the incoming user sentences. For the MALIN-DM framework, such separations concerns the role of, for instance, the linguistic interpretation, reference resolution, and the request interpretation.

Interfaces should be defined both vertically and horisontally. Vertical interfaces concerns the interaction between the format levels of sub-dialogue, utterance, and part of utterance. Horisontal interfaces concerns the transformations between the formats of the layers of interpretation, such as the parser interpretation format, the request format, and the system response format.

6.2 Dialogue Knowledge Model Customisation

The main focus in DM framework customisation of the dialogue history KM lies in the framework templates. Generic value domains of tokens of interpretation should be used coherently for all subsystems in the DS application. For the MALIN-DM framework, examples of such domains are the names of system commands, and requests that relate to database field names and values. There is a trade-off between using weak generic types, or stronger more robust typing. For the MALIN-DM framework, the use of a strong type has often proved more effective in the long run. The use of explicit domain definitions in the code of the system means that domains can be lifted out completely from the design document.

The DM tools and their formats are often fairly straightforward to use, since they have a strong connection to the selected DM theory. The main choice is here whether to represent dialogue phenomenon explicit or implicit. Explicit structures are favorable as a way of documentation and thus useful for structures that are shared by several modules. Implicitness is preferable to keep down size and thereby improve on readability and maintainability of the dialogue history KM. When using the MALIN-DG tool one such issue is whether to represent error and system messages explicitly in the dialogue tree or not.

Code patterns occur on both the declarative, or data, level and the imperative level for the dialogue history KM. Parts that lie close to the grammatical side of the dialogue system tend to become declarative. For parts that are related to the background system, however, imperative structures often seem more adequate. For the MALIN-DM framework we typically get declarative notions for request structures and imperative notions for system responses. The interplay between these levels, e.g. for focus inheritance, is often a non-trivial part of the customisation coding.

7 The User Request Handling Step

The user request handling step mainly concerns classification of each user action in terms of a **request type** in order to group request structures with similar computational content. The computational behaviour associated with the request type should be defined and implemented. In general, we distinguish between the following classes of request types, cf. [Allen and Core, 1997]:

- **task requests**: requests that involve the background, or task-execution, subsystem.
- **system information**: requests for help and explanations.
- **communication management**: user discourse signals that control the flow of the dialogue, e.g. greetings and farewells.

Strategies for request handling is often directly connected to notions of the dialogue history KM at the level of sub-dialogue or utterance.

The starting point of the user request handling design is the system behaviour, as described in the DS requirements specification. The customisation process typically starts from the generic task management modules of the selected DM framework.

7.1 User Request Handler Design

Modularisation is the main issue of user request handler design. Task modules that meet the request types should be identified and their behaviour described. In the MALIN-DM framework we identify DM sub-modules for request coordination that controls task execution in external sources, when more than one request type is present in the system. For information retrieval requests complexity normally lies in the query expression. The focal parameters of the request structures becomes fairly large and focus management can become an intricate problem. For the MALIN-DM framework, a careful design of a focus handler has been needed in some cases. Moreover, for more command-oriented tasks the design problem rather lies in the selection of the "right command". With the MALIN-DM framework, this is reflected by the need for a command interpretation sub-module.

For system information and communication management, knowledge representation also become an important issue. For system information requests the problem is to represent an open-ended domain – the domain of help-questions. The problem mainly lies in how to define a taxonomy of help texts and how to map questions into that taxonomy. In the MALIN-DM framework we designate an auxiliary handler for system information, although the handler output is only tokens for which concrete language maps must be performed later by the generation module. The selection mechanism for communication management for user dialogue control is yet another handler in the MALIN-DM framework with focus on knowledge representation for dialogue control. This communication management module handles both high-level dialogue statements, such as greetings and farewells, and near-system expressions, such as restart and emergency stops.

Interfaces between request formats and task execution modules should be defined. In systems other than those for information retrieval this means an interface from data to more procedural behaviour, such as command execution. Task handlers that rely on information from the dialogue history, such as focus management, also need an interface for the dialogue history KM.

7.2 User Request Handler Customisation

The main issue of user request handler customisation is a mixture of re-use through framework templates and code patterns. There is a double choice of request handler modularisation within the DM module – in terms of modules applied in sequence and in terms of branching during module application. There are a number of programming rules of thumb that are often helpful here: Place all related stuff in one place to avoid redundant if-cases. Perform transformations as early as possible (but not too early). Modules should be logical units with a simply formulated responsibility (preferably in close connection with the used DM theory). Keep smaller modules internal to DM if the module uses global references to the dialogue history.

Code patterns have to be used where the DM framework fails to support. This is typically the case if the DM module deals with a form of request that has not been attempted before with the framework.

Domain-independent user request handlers may need simpler forms of knowledge representation structures to be easy to adjust. It may be worthwhile to develop or use a simple DM tool for this purpose, especially for task requests with complex focus inheritance and request constructions. For the MALIN-DM framework, we are thinking about such efforts for domain-independent focus management of information retrieval.

8 The Sub-dialogue Control Step

The control of sub-dialogue strategy constitutes a major control unit of dialogue management. In the sub-dialogue control step the possible dialogue situations that may occur are grouped and realised according to a suitable dialogue strategy. We suggest to use the following generic grouping of sub-dialogues as a starting point for this work:

- **the request types**: decide upon a suitable dialogue behaviour for each type of request that the system handles.
- **exceptional system responses**: decide upon a suitable dialogue behaviour for each type of exceptional system response.
- **clarifications and error handling**: decide upon a suitable strategy for each type of clarification and system error situation.

Realisation of the control of sub-dialogues follows the guidelines concerning dialogue system functionalities of the DS requirements specification (cf. [DISC, 1999]).

The sub-dialogue control design starts off identifying the mechanisms that underlie the extraction of user request for the each class of sub-dialogues. The customisation of the step mainly has its starting point in looking at the dialogue strategies that occur in the selected DM theory and in the use-cases of the DS requirement specification.

8.1 Sub-dialogue Control Design

The sub-dialogue control design has its focus mainly on modularisation and interfaces. The characteristics of entry and exit points are interface points for the flow of information between sub-dialogues. For the MALIN-DM framework the sub-dialogue control is designed in close connection with the dialogue grammar. Modularisation definitions for the sub-dialogue strategies for system-initiative segments often need more effort than those for user-initiative, since the former tend to be more elaborate. Moreover, it is important to identify how user-initiative and system-initiative segments are intertwined.

For sub-dialogues dealing with exceptional responses, clarification and error, the design of knowledge representation also plays an important role. It is important that the formats of such messages are kept in a form that guides the exceptional sub-dialogues. Handling of empty or too large results are examples of such exceptional responses. One may distinguish between repairable and not repairable exceptional sub-dialogues. In the first case a sub-dialogue strategy should be suggested, in the latter a suitable sub-dialogue exit is preferred.

8.2 Sub-dialogue Control Customisation

The sub-dialogue customisations mainly deal with issues of DM code patterns. Strategies for sub-dialogues depend on the type of request at hand, i.e. whether they concern information retrieval, operational control, problem solving or some other type of functionality. Sub-dialogues follow patterns that may be re-used in other applications that share dialogue behaviour. For the MALIN-DM framework these patterns are found as fragments of the dialogue grammar that can be re-used between grammars of different DM modules.

There is often a need for a new sub-dialogue control for each newly introduced request type in a DM module, such as a slot-filler frame for system-initiative sub-dialogue for complex information requests or orders. DM framework templates can be developed for this purpose. For instance, the MALIN-DM framework includes a generic handler handling slot-filler requests that supports system-driven request formulation.

9 An Example – SCIN

The SCIN dialogue system is the first system where the method is used. The SCIN system consists of an interactive news reader using a multi-modal interface with an animated face, in a way similar to an ordinary news program. A user of the SCIN system is able to not only find and read news, but also, for instance, to retrieve background to news articles, analyses and encyclopedic information related to a news item.

9.1 The Iterative Process

A first prototype is up and running, utilising only very few of the features to be included in the final version. This is in line with the our implementation method; i.e. instead of completely specifying the final system, we develop a first running prototype which is to be further refined. One reason for wanting to have a running prototype before the whole system was specified was that we did not know what services to include in SCIN when the project started. Thus, instead of discussions on, and specifications of, capabilities of an intelligent news service, we developed our first prototype, the intelligent tape recorder, IB, which is to be iteratively enhanced with more capabilities. IB does not have much advanced functionality, in fact it hardly has capabilities to render it to be called a dialogue system. IB can understand a limited set of written input commands such as *Read next; What is there about Telia; Stop*. However, this made it much easier for us to develop a first running prototype where we, for instance, had to spend a considerable amount of time on interfaces between modules; code that is useful for all further prototypes.

IB will be evaluated and function as a framework when iterating the method a second time, adding more functionality towards a fully interactive news system.

9.2 The DM Capability Steps

The design of IB was done in parallel with the implementation. For the most basic parts the DM module has only been an embryo. The initial use-cases have concerned straightforward search for and reading of news telegrams. Our initial work with the dialogue history modelling has been performed mainly on the level of singular utterances, more specifically on the request formats. Our initial dialogue history KM will be further extended at the level of sub-dialogue representation and part of utterance as we proceed. For request handling, we have initially focused on the basic task execution modules of background database system with news texts, and the user interface for the read commands. A first prototype version of parser interpretations and request formats have been suggested, but we expect to refine these formats iteratively as the project continues. Much of the basic functionality of the SCIN system concerns the user interface, so the initial work has focused mostly on those issues. The role of the DM module will gradually become more central when new dialogue behavior is added, as the project continues.

10 Discussion

We have presented a method for implementation of dialogue management modules. The method unifies issues of conceptual design with a clear correspondence to the components of the customisation of a generic framework. In this paper we have focused on the dialogue management module, but we believe that the method applies to other modules in a dialogue system as well.

The method advocates that coding and design goes together and that a dialogue system is implemented iteratively with cumulatively added capabilities. Coding should be carried out as soon as possible, before all details of the system's design are ready; coding instead of chart diagrams. A prototype is developed from the start which is gradually refined based on evaluations of its behaviour. The prototype should use an existing framework, or tools if such exists, which imports results from previous projects. The framework might well be further developed, but we believe that this shall be carried out as a separate project. Within one dialogue systems development project only existing frameworks and tools are used.

The method conforms with general software development methods, but is tailored to fit dialogue systems development. Dialogue systems are characterised by having processes for the various dialogue system's tasks, such as parsing, dialogue control and domain knowledge management being fairly complex but small, i.e. not much code. Instead, as many AI systems, dialogue systems are knowledge intensive. Furthermore, much knowledge is acquired during the development of the system. This motivates our view of evolutionary development based on running prototypes capable of handling more and more dialogue phenomenon. However, prototype refinement often involves re-design of various aspects, thus design and coding are carried out together.

The method has gradually grown from our work on implementation of dialogue systems, and we will not claim that it is ready yet, but believe that it provides a step towards a software engineering method for dialogue systems development.

We have not yet been able to verify the method, but once the SCIN interactive news reader is complete, or near completion, we have a chance to do a more systematic evaluation of the method.

References

[Alexandersson and Reithinger, 1995] Jan Alexandersson and Norbert Reithinger. Designing the dialogue component in a speech translation system. In *Proceedings of the Ninth Twente Workshop on Language Technology (TWLT-9)*, pages 35–43, 1995.

[Allen and Core, 1997] James Allen and Mark Core. *Draft of DAMSL: Dialog Act Markup in Several Layers*. http://www.cs.rochester.edu/research/cisd/resources/damsl/RevisedManual/RevisedManual.html, 1997.

[Allen *et al.*, 2000] James Allen, Donna Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. An architecture for a generic dialogue shell. *Natural Language Engineering*, 6(3):1–16, 2000.

[Allen *et al.*, 2001] James Allen, Donna Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Towards conversational human-computer interaction. *AI Magazine*, 2001.

[Beck, 2000] Kent Beck. *extreme Programming explained*. Addison-Wesley, 2000.

[Carroll, 1995] John M. Carroll. *Scenario-Based design – Envisioning Work and Technology in System Development.* John Wiley & Sons, 1995.

[Dahlbäck *et al.*, 1999] Nils Dahlbäck, Annika Flycht-Eriksson, Arne Jönsson, and Pernilla Qvarfordt. An architecture for multi-modal natural dialogue systems. In *Proceedings of ESCA Tutorial and Research Workshop (ETRW) on Interactive Dialogue in Multi-Modal Systems, Germany*, 1999.

[DISC, 1999] DISC. Dialogue management grid. Technical report, http://www.disc2.dk/slds/dm/dmgrid-details.html, available February 2001, 1999.

[Fayad *et al.*, 1999] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[Hagen, 1999] Eli Hagen. An approach to mixed initiative spoken information retrieval dialogue. *User modeling and User-Adapted Interaction*, 9(1-2):167–213, 1999.

[Hulstijn, 2000] Joris Hulstijn. *Dialogue Models for Inquiry and Transaction*. PhD thesis, Universiteit Twente, 2000.

[Jönsson and Dahlbäck, 2000] Arne Jönsson and Nils Dahlbäck. Distilling dialogues - a method using natural dialogue corpora for dialogue systems development. In *Proceedings of 6th Applied Natural Language Processing Conference*, pages 44–51, 2000.

[Jönsson, 1997] Arne Jönsson. A model for habitable and efficient dialogue management for natural language interaction. *Natural Language Engineering*, 3(2/3):103–122, 1997.

[Karlsson, 1995] Even-André Karlsson. *Software Reuse – A holistic approach*. John Wiley & Sons, 1995.

[Krutchen, 2000] Philippe Krutchen. *The Rational Unified Process, An Introduction, 2nd edition*. Addison-Wesley, 2000.

[Larsson *et al.*, 2001] Staffan Larsson, Robin Cooper, Elisabet Engdahl, and Peter Ljunglöf. Information state and dialogue move engines. *Electronic Transactions on Artificial Intelligence*, 2001.

[Leceuche *et al.*, 2000] Renaud Leceuche, Dave Robertson, Catherine Barry, and Chris Mellish. Evaluating focus theories for dialogue management. *International Journal on Human-Computer Studies*, 52:23–76, 2000.

[McRoy and Ali, 2001] Susan W. McRoy and Syed S. Ali. A practical declarative model of dialog. *Electronic Transactions on Artificial Intelligence*, 2001.

[McTear, 1999] Michael F. McTear. Software to support research and development of spoken dialogue systems. In *Proceedings of Eurospeech'99, Budapest, Hungary*, 1999.