The Dialogue Engineering Life-Cycle

Laila Dybkjær and Niels Ole Bernsen
Natural Interactive Systems Laboratory, Odense, Denmark
laila@nis.sdu.dk, nob@nis.sdu.dk

Abstract

Software engineering life-cycle models have been around for about 30 years. Various models have been proposed, refined and adapted to the needs of different sub-areas of systems development. Dialogue engineering is a fairly new sub-area of software engineering, commercial spoken language dialogue systems (SLDSs) having been in the market place for only about a decade. While an iterative software engineering life-cycle model may apply to SLDS development at a general level, the model is insufficient at lower levels of detail. To better support SLDSs developers in industry and research, there is a need for a model which specialises software engineering life-cycle modelling to the development and evaluation processes which are specific to SLDSs and their components. This paper describes and illustrates dialogue engineering life-cycle modelling and how to tailor it to the field of SLDSs.

1. Introduction

Even if software engineering life-cycle modelling is not new, a continuing variety of general models having appeared over the last 30 years, the field remains of major interest to industry and research. For industry, a life-cycle model is an important tool to efficiently control the development and evaluation process and ensure quality. In research, professional engineering practice is becoming increasingly important as researchers are moving from component exploration to advanced systems development. In several sub-areas of systems development, general models have been refined and adapted to better suit the particular needs of those areas and thus provide better support. An example is safety-critical systems where evaluation is emphasised much more than in many other areas of software development. One reason why life-cycle modelling is a research topic in itself is that the adaption process has not yet happened in all sub-areas. Dialogue engineering is one such sub-area in need of a model which specialises software engineering life-cycle modelling to the development and evaluation processes that are specific to SLDSs and their components.

This paper proposes a dialogue engineering life-cycle model which is tailored to SLDSs development and evaluation. Section 2 reviews general software engineering life-cycle models and briefly describes the main models or their main representatives, i.e. the waterfall model, the iterative model, prototype development, the spiral model, and extreme programming. Section 3 presents the general DISC dialogue engineering model which specialises iterative life-cycle modelling to the special needs of dialogue engineering. Section 4 describes and illustrates the DISC dialogue engineering model in detail, including life-cycle issues, integration of so-called 'grid' issues, addition of evaluation criteria, and description of the resulting dialogue engineering development and evaluation life-cycle.

2. Software engineering life-cycle models

The first software engineering life-cycle model was introduced about 30 years ago. Before that time developers used what has been called the *tunnel development model* (Muller 1997). Projects were initiated and, some day later, maybe, a system resulted. There was no model of the development process itself. But there was a clear need to get some structure to the process, e.g. to better understand software development and justify the resources spent on development.

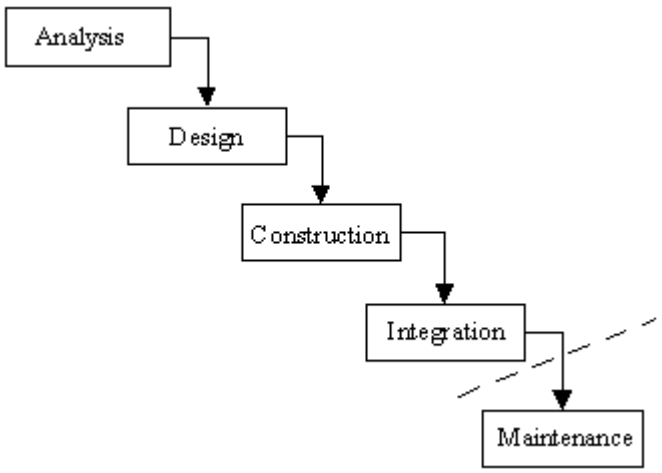

Figure 1. The waterfall model of software development.

This first model (Royce 1970) became known as the *waterfall model* because it basically describes the development process as a linear execution of four steps followed by a maintenance phase when the software has been put into operation, cf. Figure 1. The five steps

are known by different names. We shall refer to them as analysis, design, construction, integration, and maintenance, respectively. The model was well received because it provides structure and visibility to the development process.

The identification of major activities and the shifts in overall focus as development proceeds was the prime contribution of the waterfall model and is still valid today. However, the model is too simplistic and rigid. The strictly separated phases lead to costly handling of changes and poorly reflect human cognitive and social processeses.

The first problem identified in the model was the assumption that a system or component is only being evaluated in the integration phase, i.e. towards the very end of the development process. Until then, only documents are available for validation. This means that even serious problems may only be discovered at a late stage when the cost of making changes is high. A second problem is that the model assumes that a particular phase is being fully completed before the next phase is initiated. This means that, e.g., all requirements must be known and made explicit in the analysis phase before continuing to the design phase, which is often in conflict with the reality of software engineering practice.

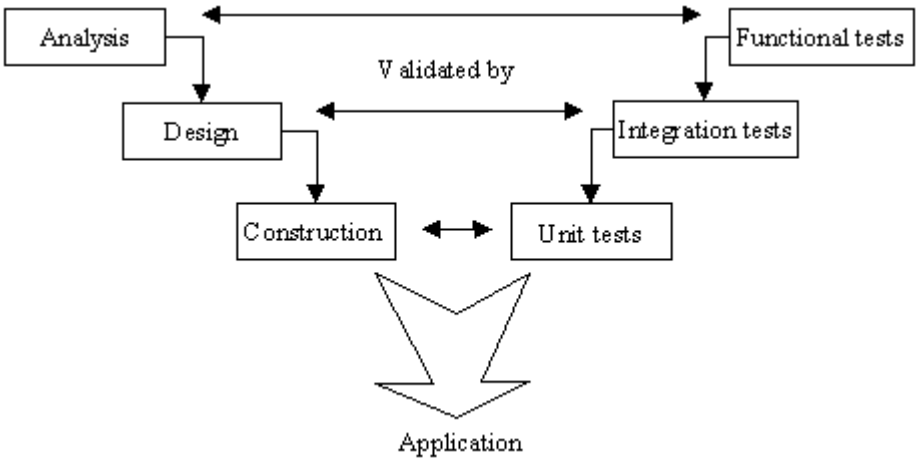

Figure 2. Example of a V development model. Tests are developed in parallel with the software (Muller 1997).

One way to address the evaluation problem is to introduce elements of evaluation at earlier stages in the life-cycle. Later versions of the waterfall model are thus sometimes presented in the form of the letter V, indicating that the development of test plans and test data is done synchronously with software development. Even if, in the waterfall model, actual evaluation is only carried out once the software is ready, the *V-model* is likely to improve developers' understanding of the life cycle stages and the software being developed by encouraging them

to decide on detailed evaluation plans and data at each stage. Figure 2 shows an example of a V model in which functional tests are specified during analysis, integration tests during design, and unit tests during the construction phase.

However, the V-model remains rigidly sequential and puts too little focus on the human processes. Several alternative, general life-cycle models have been proposed in the attempt to overcome the problems of the waterfall model and meet the needs of complex systems development, see e.g. (Pressman 1997, Sommerville 1992 or later) for an overview. Some of these models have not been used much in practice, such as the formal transformation model (Pressman 1997, Sommerville 1992). Other models appear under different names but are, in fact, quite similar and may be viewed as variations on the same model, e.g. exploratory programming and prototyping. In this brief overview we shall only mention what we consider the main models or their main representatives, i.e. the iterative model, prototype development, the spiral model, and extreme programming. The overview focuses on process structure. Other important aspects that have been identified in the 1980s and 1990s are not included, such as user involvement, organisational and social implications of introducing new systems, teamwork, or design rationale representations.

A frequently used model is the *iterative model* which has many variations depending on, e.g., project size and domain complexity (Muller 1997). The iterative model incorporates the idea of iterating one or more of the first four phases of the waterfall model, cf. Figure 3. Moreover, phases may overlap and sometimes a breakdown into more detailed phases is introduced. Today, the trend is towards shorter cycles, focusing on prototypes. In fact, both prototype development, the spiral model and extreme programming are iterative models and may be viewed as further developments of the first iterative software engineering life-cycle models.

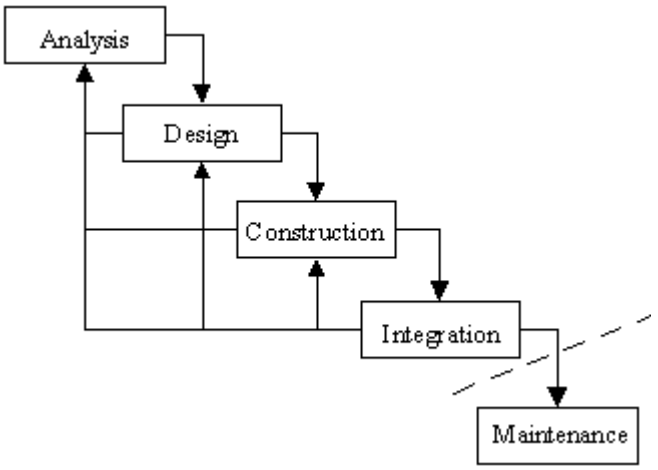

Figure 3. The iterative model.

Prototype development has been used widely since the heydays of AI 20 years ago. Prototyping is a useful method when it is not clear in advance what the system should be like, as may be the case with, e.g., systems exploring new forms of interaction. Prototypes are also well suited for rapidly constructing an early version of the intended system. Prototyping requires good planning and process control. One risk is that programmers do not want to discard code from the prototype even if the final system could be improved if (part of) the prototype code were re-implemented.

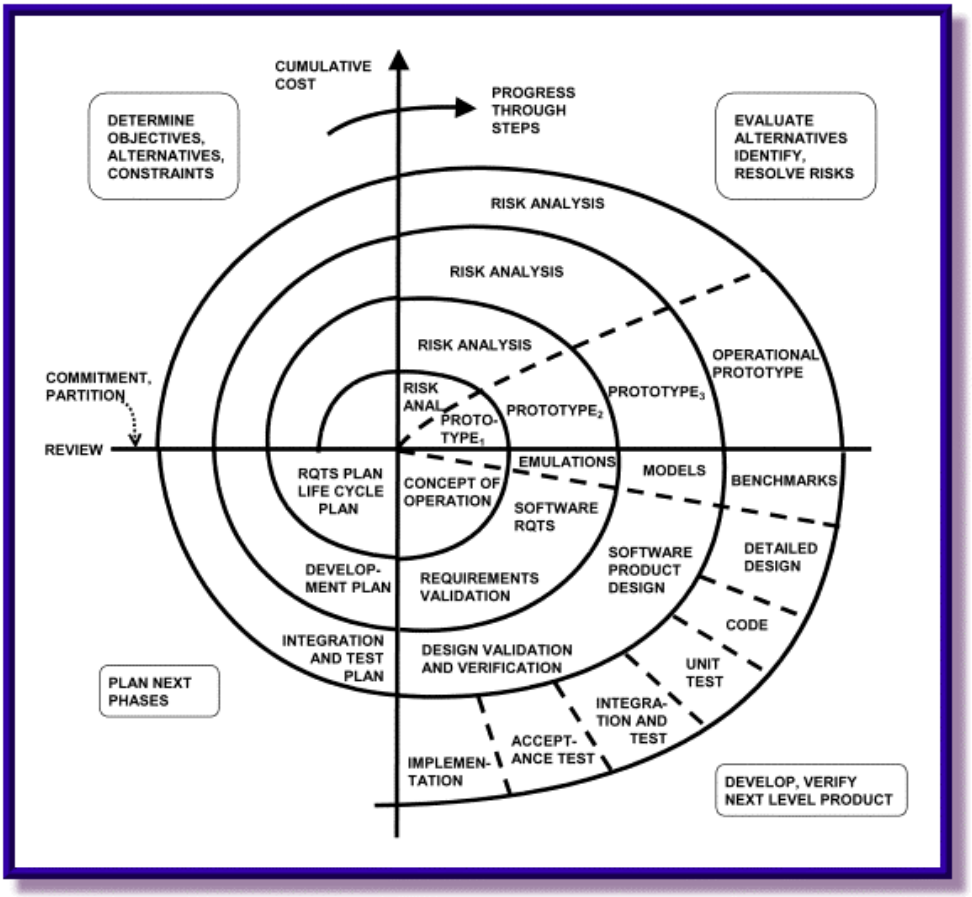

Figure 4. Spiral development diagram
(http://www.stsc.hill.af.mil/crosstalk/2001/may/boehm.asp).

The *spiral model*, cf. Figure 4, was introduced in the mid-1980s (e.g. (Boehm 1988)). "It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions" (Boehm and Hansen,

http://www.stsc.hill.af.mil/crosstalk/2001/may/boehm.as). Roughly speaking, the model combines prototype development with use of the waterfall model for each step. The spiral model is intended to help manage risk. The system is defined and developed stepwise, corresponding to cycles in the spiral. Each cycle is in principle ended by answering the question: "Should we continue?" either negatively or positively depending on the risks involved. Each new cycle starts by an analysis to determine the best way in which to deal with the current cycle. Basically, each cycle in the spiral includes the following four steps: determine objectives, analyse and evaluate, develop and test, and plan next phase. The spiral model has strongly influenced later systems development methods, such as DSDM (Dynamic Systems Development Method) (www.dsdm.org).

Extreme programming is a fairly recent model. It basically prescribes to do everything (analysis, design, etc.) all the time, i.e. using very rapid iterations. Programming is done pairwise, i.e. programmers sit together in pairs when writing code, providing real-time analysis and code review. A limited-functionality prototype is developed very early in the project. For instance, the prototype may be able to handle a single key functionality issue only. Every day, every week, or whatever may be the turn-over time, an improved version of the prototype is made ready. In this way, the prototype is eventually extended to have the functionality required of the final system. Thus, the prototype keeps changing rapidly, and components are being updated independently (e.g. (Beck 1999a, Beck 1999b)).

The two major problems in the waterfall model, i.e. (too) late evaluation and strict phase seriality, cf. above, are both addressed in the iterative model and its variations discussed above. By being iterative, those models by definition overcome the problem that a particular phase is being fully completed before the next phase is initiated. Moreover, by consequence of their iterative nature, the models share the idea that system or component evaluation is not only done once towards the end of the development process but is done throughout development. The main difference among the models as regards evaluation would seem to be the time it takes to make an iteration, with extreme programming assuming the faster turn-around time.

Software engineering life-cycle issues have remained an important research area ever since the first model was presented, and their scope includes both custom-made and off-the-shelf software. Using an adequate life-cycle model is of key importance to efficient and successful software development. Nevertheless, too little attention is often paid to this fact by development teams. This may partly be due to ignorance but an important reason probably also is that detailed software life-cycle models are still missing in many areas or are still at the

research stage. An area in point is that of spoken language dialogue systems (SLDSs). A specialised dialogue engineering life cycle model is discussed in more detail in the following sections.

3. The DISC dialogue engineering model

The first, very simple commercial SLDSs were introduced only about 10 years ago. As increasingly advanced and complex SLDSs are being developed in research and industry, the need has emerged for a detailed dialogue engineering life-cycle model.

At a general level, the iterative life-cycle model in Figure 3 also applies to SLDSs development. However, a model which specialises to the development and evaluation processes which are specific to SLDSs and their components could provide far better support to SLDS developers. Also, there is a need for a model which not only focuses on software development but also on its evaluation, on the development and evaluation of documentation, and on the continuous evaluation of factors which may influence the development and evaluation process at any time throughout the life-cycle.

In the European DISC project (www.disc2.dk) on best practice in the development and evaluation of SLDSs, we developed a draft dialogue engineering model for SLDSs and components. The model includes a life-cycle model, a so-called 'grid' (see below), and a set of evaluation criteria. The draft model, and in particular its life-cycle part, has been further developed and refined after the DISC project ended in early 2000. The resulting dialogue engineering model is described below using dialogue management for illustration.

The current version of the DISC dialogue engineering model is based on (i) first ideas in (Bernsen et al. 1998), (ii) the DISC approach to dialogue engineering, (iii) analysis by the DISC partners of the actual life-cycles and properties ('grid' issues, see below) of a large number of different SLDSs and components, and (iv) the draft DISC model with subsequent elaborations. All DISC results are available at www.disc2.dk.

In the DISC approach, an SLDS has six aspects: speech recognition, speech generation, natural language understanding and generation, dialogue management, human factors, and system integration. In simple systems, the natural language understanding and generation aspect may be non-existent but the five other aspects probably must be present for the system to be an SLDS at all (even low quality human factors are human factors). From the point of view of best practice, an SLDS should be the result of (a) correct choices among the available

options, technological and otherwise, within each aspect and (b) correct development (including evaluation) practice.

Based on analysis of 25 existing SLDSs and components (Bernsen et al. 1999), DISC has developed a 'grid' best practice guide per aspect. Each grid defines a space of aspect-specific issues which the developer must, or may have to, address, primarily depending on the complexity of the SLDS or component to be developed. When developing a dialogue manager, for instance, the developer should consider if the dialogue manager should provide some form of graceful degradation in order to handle cases of repeated user-system communication error. For each issue, the currently available options are laid out in the grid together with the pros and cons for choosing a particular option (cf. (a) above). For instance, feedback to the user is an important issue in dialogue management. The grid offers two options, i.e. process feedback and information feedback. Process feedback informs the user that the system is still working even if it takes some time to, e.g., query the database. Information feedback allows the user to verify that spoken input has been correctly understood by the system. Both types of feedback can be provided in several different ways which are presented as options together with their pros and cons.

In addition to the aspect-specific grids and again based on the analysed exemplars, DISC has developed a life-cycle best practice guide per aspect (cf. (b) above). The current DISC dialogue engineering life-cycle model has two interrelated levels. At the overall level, and assuming a general iterative software engineering life-cycle model, the model is used in developing and evaluating entire SLDSs as well as individual SLDS components. At the more detailed level, additional support is provided for addressing a particular aspect of an SLDS by specialising the life-cycle to this aspect, i.e. by taking into account the particular grid issues and evaluation criteria which are relevant to that aspect, such as dialogue manager development.

4. Life-cycle issues, grid issues and evaluation criteria

This section describes the current status of the DISC dialogue engineering model. We take the life-cycle issues as point of departure, integrate grid issues, add evaluation criteria, and describe the resulting dialogue engineering development and evaluation life-cycle.

4.1. Life-cycle issues

The dialogue engineering life-cycle may be depicted following the V-model as having two legs. One leg addresses the development phases. The grid issues must be addressed in these phases. The second leg shows what is being evaluated. Particular evaluation criteria, which depend on the aspect considered, are derived from the chosen grid issues and used at specified points during evaluation. Evaluation is an integral part of the development process and is carried out throughout the life-cycle. What is being evaluated depends on the particular life-cycle phase.

Figure 5 shows the five major development phases (the maintenance phase is not shown) which are typical to SLDSs, and the major evaluation activities in focus during those phases. It should be noted that simulation is frequently used in the development of (advanced) SLDSs. This is why simulation is mentioned as a separate (optional) phase. Unlike the V-model in Figure 2, Figure 5 shows what to evaluate in a particular phase but does not show when a particular type of evaluation is planned.

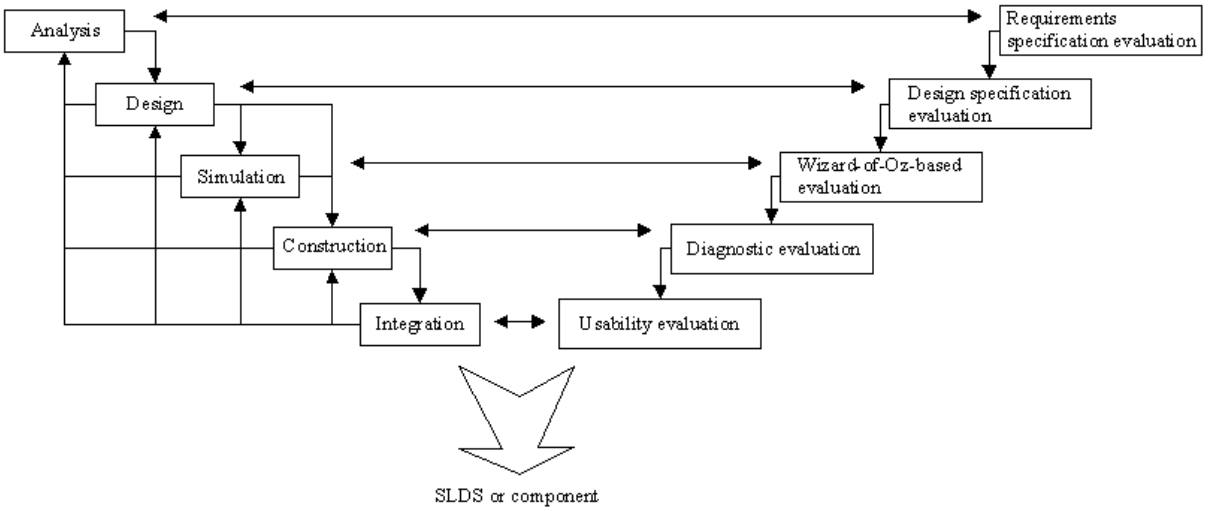

Figure 5. The dialogue engineering life-cycle phases apart from the maintenance phase, and the overall issues to evaluate.

Figure 5 focuses on software development phases and evaluation. Documentation is not considered even though this is an important part of the development and evaluation process. Also, the figure does not show or relate to parameters, such as resources, skills and unexpected problems which may strongly influence development and evaluation at any point in the life-cycle, and which should therefore constantly be monitored.

Figure 6 shows the dialogue engineering life-cycle model, including life-cycle phases, system/component evaluation, document evaluation, and monitoring of other key factors. Documentation is assumed to develop along with the system or component during the individual life-cycle phases as described in more detail below.

Life-cycle phase	**System/component evaluation**	**Document evaluation**		**Management factors evaluation**
Analysis	Requirements specification evaluation	Requirements specification documentation	Development and evaluation process description	Development time Personnel resources Mastery of the development and evaluation process Problems during development and evaluation Management quality
Design	Design specification evaluation	Design specification documentation		
Simulation (optional)	Wizard-of-Oz-based evaluation	Wizard-of-Oz documentation		
Construction	Diagnostic evaluation	System/component documentation		
Integration	Usability evaluation	User manual and guide		

Figure 6. The dialogue engineering life-cycle model (apart from maintenance).

A number of life-cycle issues pertain to one or more of the life-cycle phases in the left-hand column of Figure 6. Similarly, what is included in the development process must also be evaluated and thus reflected in the evaluation criteria to be used (see below). Figure 7 shows a set of life-cycle issues which are general and which are assumed by the dialogue engineering model presented here. The issues all relate to the software being developed (general, constraints, ideas and preferences, and criteria), the documentation relating to the software (documentation and references), or the management factors which influence the development process (management factors). The last group of issues in Figure 7 (post-development issues) relate to the maintenance phase and will not be discussed any further. Evaluation criteria will be discussed later.

General

Overall design goal(s) define the general objective(s) of the development process.

Constraints

Hardware constraints refer to any a priori constraints on the hardware to be used in the design process.

Software constraints refer to any a priori constraints as regards use of software (e.g. development platform or pre-existing modules or components).

Customer constraints are constraints imposed on the system/component by the customer, if any. Customer constraints may include hardware or software constraints.

Organisational aspects address if and how the system/component will have to fit into some organisation or other.

Other constraints than hardware, software and customer constraints may be imposed on the system/component and influence the development process.

Ideas and preferences

Design ideas are ideas which developers may wish to realise in the development process, such as to develop a generic dialogue manager.

Developer preferences may impose constraints on system/component development which were not dictated from elsewhere. Developer preferences may relate to many different issues, e.g. preferred programming language.

Criteria

Realism criteria describe if the system/ component will meet real user needs, will meet them better, in some sense to be explained (cheaper, more efficiently, faster, other), than known alternatives, or if the dialogue manager is "just" meant for exploring specific possibilities to be explained, or if there are other realism criteria which should then be explained.

Functionality criteria concern which functionalities the system/component should have.

Usability criteria concern the usability aspects of the system/component. Usability criteria may be seen both from a system developer's and from an end-user's point of view. Usually the main focus will be on usability for end-users.

Documentation and references

Requirements specification documentation addresses how the requirements specification is documented.

Design specification documentation addresses how the design specification is documented.

Wizard-of-Oz documentation addresses the documentation of the (partially) simulated system/component.

System documentation addresses the documentation of the implemented system/component.

User manual and guide concerns the description of the system/component to its users.

Development and evaluation process description (including references) is used to capture specifications, choices and decisions made, their justifications and their consequences and results. The description should include, e.g., information on how requirements were established, how the system/component was developed and evaluated, implementation issues, and tests done on the system/component.

Management factors

Development time addresses the time planned for development and the actual time used.

Personnel resources address the amount of person months allocated to, and actually spent on, the development of the system/component.

Mastery of the development and evaluation process addresses the question of which parts of the process the development team has sufficient mastery in advance and of which parts they don't have such mastery.

Problems during development and evaluation should be described and handled.

Management quality concerns the way in which the project is managed.

Post-development issues

Portability addresses ease of portability.

Modifications concern what is required if the system/component is to be modified.

Additions, customisation addresses how additions to, and customisation of, the system/component can and should be carried out, e.g. if there is a strategy for resource updates and if there is a tool to enforce that the optimal sequence of update steps is followed.

Figure 7. Issues addressed by the dialogue engineering life-cycle model.

Some life-cycle issues are important to several development phases whereas others are restricted to a single phase as explained in more detail below. For each life-cycle issue a number of details must be considered, including the phase(s) to which it primarily belongs, evaluation methods to apply, the influence of the issue on the SLDS or component, the importance of taking it into account and possible effects of not addressing it, particular difficulties relating to the issue, and people with major influence on the issue. Space does not allow us to address all of these details. Only the two first-mentioned points will be discussed below.

We distinguish three categories of stakeholder in development: the procurer or customer, the users, and the developers. Most SLDSs are custom-made software (although several of their

components are not) and therefore the customer also plays an important role. In case of off-the-shelf software, the development process roughly remains the same but the input from the customer is not available, so developers have to manage without it, contribute the information themselves or obtain it from representative users.

4.2. Grid issues specialising life-cycle issues

As mentioned in Section 3 we shall use the dialogue management aspect for grid illustration. Dialogue management grid issues may be structured into the following categories covering the issues listed in parentheses (Bernsen and Dybkjær 2000a):

- goal;
- system varieties (multimodal systems including speech, multilingual systems, and multi-task, multi-user systems);
- system speech and language (are the speech and language layers OK, do the speech and language layers need support from the dialogue manager, and real-time requirements);
- getting the user's meaning (task complexity, controlling user input, who should have the initiative, input prediction/prior focus, sub-task identification, and advanced linguistic processing);
- communication (domain communication, meta-communication, other forms of communication such as greetings, expressions of meaning, i.e. how is the meaning of what has to be conveyed to the user expressed by the dialogue manager, error loops and graceful degradation, feedback, and closing the dialogue); and
- history, users, implementation (dialogue histories, novice and expert users, user groups, other relevant user properties, and implementation issues).

Jointly with the system/component-specific grid issues, the first four categories of life-cycle issues in Figure 7 (general, constraints, ideas and preferences, and criteria) determine the software that is being developed. The life-cycle issues are general but, taken together with the grid issues, they aim at a particular aspect of an SLDS. Let us look at some examples. The dialogue management grid issue categories from the above list are referenced in parentheses. The grid issues are described in detail at http://www.disc2.dk/slds/dm/DMgrid.html and more comprehensively in (Bernsen and Dybkjær 2000a).

The life-cycle issue of customer constraints covers constraints imposed on the system/component by the customer. Customer constraints will typically include grid issues.

Many different customer constraints may be imposed on dialogue manager development. For example, the customer may want the system-provided service backed up by human operator fallback during the company's opening hours.

Design ideas are ideas which the developers want to realise in the development process. For dialogue management, examples of design ideas involving particular grid issues could be to explore internal dialogue manager modularity (grid: implementation), investigate different ways in which the dialogue manager can support natural language understanding (system speech and language), explore system co-operativity in dialogue (communication), experiment with different dialogue control strategies (getting the users meaning), or explore ways in which to exploit contextual information to improve the dialogue (history, users, implementation).

Usability criteria may be viewed both from a system developer's and from an end-user's point of view. Usually the main focus will be on usability for end-users. End-users will not experience the dialogue manager as a stand-alone component but only as part of an entire system. To a large extent, however, it is the dialogue manager which is responsible for how satisfactory the SLDS is to use. Grid issues which may generate usability criteria include, e.g., natural, flexible and robust dialogue, which has to do with initiative, feedback, error loops and graceful degradation, histories, user properties, etc. (getting the users meaning, communication, and history, users and implementation), sufficient meta-communicative facilities (communication), and users' backgrounds (history, users, implementation).

The first four categories of life-cycle issues in Figure 7 and the grid issues which become subsumed by them, are typically in focus already in the analysis phase or, at the latest, in the design phase, depending on whether they are brought in as requirements or as part of the design specification. Some grid issues may be involved in the analysis phase as part of a life-cycle issue whereas others are added to that life-cycle issue in the design phase. For instance, meta-communication may be a grid issue brought in as a usability constraint in the requirements specification, i.e. in the analysis phase, whereas, e.g., error loops and graceful degradation may be added as another usability constraint in the design phase as part of the design specification. Overall goals and customer constraints are usually specified in the analysis phase whereas developer preferences typically belong to the design phase. The remaining life-cycle issues (first four categories) may be specified during analysis and/or during design, depending on their importance in the actual development process. All the life-cycle issues from the first four categories continue to play a role not only in analysis and/or design but also in later phases, i.e. in simulation, construction, and integration. These later

phases serve to progress the development of a system or component in accordance with the requirements and design specifications developed during analysis and design.

The documentation issues, on the other hand, are related to a particular phase, cf. Figure 6, except for the last one which is an accumulating document throughout development across phases and iterations.

The management issues in Figures 6 and 7 do not belong to any specific phase but must be monitored throughout development. Development time and personnel resources may appear as, e.g., customer constraints and be part of the requirements specification. Mastery of the development and evaluation process may be a factor which influences the resources set aside in terms of time and personnel and which therefore also influences the requirements specification in an indirect way. Throughout the development process, it is important to monitor that the resource budget (time and personnel) holds and, if not, take the necessary actions immediately. Problems during development and evaluation may influence the resource budget. It is therefore important to keep an eye on such problems and consider solutions and consequences as soon as they are spotted. Management quality is determined by how well monitoring and management of the development and evaluation process is done, whether the appropriate actions are taken, and whether this is done in a timely fashion.

4.3. Evaluation criteria derived from grid issues

Since evaluation is an integral part of the development process, we also need to take a look at evaluation criteria and the evaluation process. Evaluation criteria are aspect-specific in the same way as grid issues are aspect-specific.

An interesting observation made in DISC is that, based on the grid issues, it is possible to derive a set of evaluation criteria per aspect. Suppose that, for instance, the dialogue management grid includes 24 issues for consideration by dialogue manager developers, such as which types of dialogue histories to include in a particular application. If the SLDS to be developed is a relatively simple one, not all of the 24 issues are likely to be relevant, so the developers select options within, say, 14 of the issues and ignore the remaining issues because these are relevant only to more sophisticated dialogue managers than presently needed. In this case, the developers must apply evaluation criteria to 14 chosen dialogue manager options in order to do a complete evaluation of the dialogue manager aspect of the application. Process and results of generating a complete set of evaluation criteria for human factors in SLDSs are presented in (Dybkjær and Bernsen 2000).

Knowing *what* to evaluate, is not enough, however. *How* to evaluate is just as important. To follow best development practice, developers have to evaluate their solution with respect to a chosen option at the right time(s) and in the right way(s). Thus, how to evaluate is a matter of applying a particular evaluation criterion correctly at the right stages during the development life-cycle.

Given that the developer knows, per SLDS aspect and for the particular application at hand, what to evaluate, such as how well the dialogue manager handles error loops and graceful degradation, focus can shift to how to do the evaluation. In DISC, we have iteratively developed an evaluation template to support the 'how' of evaluation. The template is a model of what the developer needs to know in order to apply an evaluation criterion to a particular property of an SLDS or component, such as the histories used by the dialogue manager. This knowledge is specified by the template's ten entries including what is being evaluated, system part evaluated, type of evaluation, method(s) of evaluation, symptoms to look for, life-cycle phase(s), importance of evaluation, difficulty of evaluation, cost of evaluation, and tools. Details on the evaluation template can be found in (Bernsen and Dybkjær 2000b). Examples of filled templates can be found at the DISC website, e.g. http://www.disc2.dk/slds/dm/-DMevaldetail.html.

4.4. Evaluation integrated into the development life-cycle

This section briefly discusses evaluation and evaluation methods as part of the development life-cycle. It should be noted that the evaluation criteria described above which were derived from the grid issues, only form part of what has to be evaluated. The grid-derived evaluation criteria only relate to the software and not to, e.g., documentation or the development process itself.

In the *analysis phase,* the overall goals of the system/component as well as the most important constraints, ideas and preferences, and criteria, cf. Figure 7, are established and described in a requirements specification. This is done in collaboration with the procurer (if any) and the system end-users. An important activity is to specify the evaluation criteria which the final system must satisfy to be accepted by, e.g., the procurer. A first evaluation is made of the feasibility of the requirements specification given constraints and resources. The requirements specification and any additional documentation is produced and evaluated as to sufficiency and clarity.

During the *design phase*, a design specification based on the requirements specification and other sources is worked out, adding additional constraints, ideas, preferences and criteria, and detailing these to a level sufficient to form the basis of simulation or construction. In parallel and closely interacting with this activity, a design analysis evaluation takes place. Design and design analysis evaluation involve using experience and common sense, thinking hard when exploring the design space, doing walkthroughs of models, comparing with similar systems, browsing the literature, applying existing theory, guidelines and design support tools, if any, involving experts and future users, etc. The completeness of the design specification may be judged by checking whether all relevant entries in the DISC "grid(s)" have been considered. Design analysis evaluation also consists in checking whether the design goals and constraints are sound, non-contradictory and feasible given the resources available. The documentation produced in this phase includes the design specification and any additional documents which have been used or produced. For instance, literature should be referenced and walkthroughs and their analysis should be documented. Documentation evaluation consists in judging whether the design specification is appropriately represented and whether all relevant documents have been included.

The *simulation phase* is an optional part of SLDSs development. Typically, simulations are made using the Wizard-of-Oz (WOZ) simulation methodology in which the system or some of its components as specified during design are being simulated by one or more humans with subjects who should preferably believe that they are interacting with a real system (Fraser and Gilbert 1991, Bernsen et al. 1998). The purpose is to gather data early on concerning how well the system or component might work, which means that analysis and evaluation of WOZ data is an important part of this phase. The advantage of early simulation is that, if done extensively and analysed carefully, a large number of problems with the design concepts as evidenced by observed phenomena in the simulated human-system interactions can be spotted, diagnosed, and removed early in the development process. The disadvantage is the cost of setting up and running the simulations, and of analysing the generated data. WOZ data gathering often includes use of questionnaires and interviews for investigating subjects' opinions of the simulated SLDS or component. These may provide crucial insights into the users' perception of the system or component and help capture user observations which might have implications for virtually any kind of deficiency. The documentation produced in this phase addresses the preparation and set-up of the WOZ simulation, the actual experiments carried out, the analysis of the collected data, and the implications for the design specification

and the next WOZ iteration (if any). Documentation evaluation focuses on whether the WOZ process is adequately represented.

In the *construction phase,* the specified system or component is being implemented and debugged. During debugging, two typical types of diagnostic test are glassbox tests and blackbox tests. In a glassbox test, the internal system representation is inspected. The evaluator should ensure that reasonable test suites, i.e. data sets, are constructed which will activate all loops and conditions of the program being tested. In a blackbox test, only input to, and output from, the program are available to the evaluator. Test suites are constructed in accordance with the requirements and design specifications, and along with a specification of the expected output. Expected and actual output are compared and deviations must be explained. In general, either there is a bug in the program or the expectation is incorrect. Bugs must be corrected and the test run again. It may be added that the blackbox test may also suggest that the expected output, even if forthcoming, is flawed, leading to partial redesign and illustrating the iterative nature of development. Test suites should include fully acceptable input as well as borderline cases to test if the program reacts reasonably and does not break down in case of input error. The documentation produced in the construction phase includes comments in the code, proper descriptions of the implemented system or component architecture, modules and interfaces, and test data and results. Documentation evaluation consists in checking whether this information is provided in an acceptable form.

The outcome of the *integration phase* is the final system or component. The final integration of system or component parts is done in this phase. The integrated system is tested thoroughly and, when judged to satisfy the requirements, delivered to the customer. The tests involve interaction between the system and real users, either in controlled experiments with selected users and scenarios which they have to perform, or in field studies in which the SLDS or component is being exposed to uncontrolled user interaction. The collected user-system interaction data is analysed and used to evaluate the system or component. The user-system interaction data may be complemented by data from questionnaires and interviews. The main difference is that in the integration phase there should be far fewer problems to diagnose and solve. The final test will often be the acceptance test in which the procurer (if any) tests the system or component according to the evaluation criteria specified early on (cf. above) to verify that it meets the agreed requirements. The documentation in this phase includes a user manual and guide (if any), descriptions of preparations and set-up of controlled user tests and/or field tests, and of the actual tests carried out, analysis of the collected data and possibly

of the implications for the implemented system. Documentation evaluation consists in checking whether the documentation is provided and adequate.

Maintenance is the final (and longest) phase in the life-cycle. The system or component is in actual use during this phase and maintenance includes, e.g., the correction of errors, functionality improvements, and extensions to provide new services.

During the life-cycle phases listed in Figure 6, a global description of the development and evaluation process should be worked out, cf. Figure 7. This document should reference all materials produced and provide a description of the process. The document is useful during maintenance by providing easy access to information, and it can be invaluable when planning new projects. It is also useful for new members joining the development team after the project has started.

Moreover, parameters which cannot be fully controlled in advance and which may negatively influence the development and evaluation process at any time in the life-cycle, should be monitored regularly throughout and action taken as early as possible when needed, cf. the management factors in Figure 7.

Any of the phases in Figure 6 may be iterated as needed. For example, the simulation phase will typically iterate with the design phase a number of times in order to get the design specification right, and the integration phase will normally reveal problems which require (minor) adjustments to the code, thus iterating with the construction phase. Moreover, phases my overlap, so that, e.g., WOZ planning and simulation may be initiated whilst the design specification is being worked out.

5. Conclusion

We have presented an extended DISC dialogue engineering model which aims to support developers of SLDSs and components by providing a life-cycle model which is tailored to the needs of SLDS development and evaluation. The life-cycle model takes a general, iterative life-cycle model as its starting-point and specialises the model through aspect-specific grid issues, i.e. properties of SLDSs and their components, as well as evaluation criteria and methods. The model presented might be transferable to specialised software engineering models in other areas of speech and language engineering and beyond. The model's dependence on a state-of-the-art 'grid' means that the model has to be continuously updated in order to take into account novel technical developments as well as their usability

implications. In the area of SLDSs, new technical developments will include, among others, multimodal dialogue systems which include spoken dialogue, web-based spoken dialogue systems, and the taking of a major step beyond task-oriented SLDSs towards domain-oriented systems which no longer enable particular tasks but allow free-form conversation about any topic in a domain.

Acknowledgements

We gratefully acknowledge valuable comments from Hans Dybkjær on a draft version of this paper.

References

Beck, K. 1999a. *Extreme Programming Explained. Embrace Change.* Addison-Wesley.

Beck, K. 1999b. Embracing Change with Extreme Programming. *IEEE Computer*, October, 70-77.

Bernsen, N. O., Dybkjær, H. and Dybkjær, L. 1998. *Designing Interactive Speech Systems. From First Ideas to User Testing.* Springer Verlag.

Bernsen, N. O. and Dybkjær, L. 2000a. From single word to natural dialogue. Invited book chapter in Marvin V. Zelkowitz (Ed.): *Advances in Computers*, Vol. 52. London: Academic Press, 2000, 267-327.

Bernsen, N. O. and Dybkjær, L. 2000b. A methodology for evaluating spoken language dialogue systems and their components. *Proceedings of the Second International Conference on Language Resources and Evaluation* (LREC 2000), Athens, 183-188.

Bernsen, N. O., Dybkjær, L. and Heid, U. 1999. Current practice in the development and evaluation of spoken language dialogue systems. *Proceedings of Eurospeech'99,* Budapest, Hungary, 1147-1150.

Boehm, B. W. 1988. A spiral model of software development and enhancement. *IEEE Computer*, 21 (5), 61-72.

Dybkjær, L. and Bernsen, N. O. 2000. Usability issues in spoken language dialogue systems. *Natural Language Engineering*, Special Issue on Best Practice in Spoken Language Dialogue System Engineering, 6 (3,4), 243-272.

Fraser, N. M. and Gilbert, G. N. 1991. Simulating speech systems. *Computer Speech and Language* 5, 81-99.

Muller, P.-A. 1997. *Instant UML*. Wrox Press Ltd., Canada.

Pressman, R. 1997. *Software Engineering: A Practitioner's Approach*, European Edition. McGraw Hill.

Royce, W. W. 1970. Managing the Development of Large Software Systems. *Proc. WESTCON*, San Francisco CA.

Sommerville, I. 1992. *Software Engineering*. Fourth Edition, Addison-Wesley, (1992 or newer edition).