Goal-oriented software: when the development process is different...
S. Bull\textsuperscript{a} \& T. Musson\textsuperscript{b}

\textsuperscript{a}Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, UK

\textsuperscript{b}Department of Computer Studies, Napier University, 219 Colintraive Road, Edinburgh, EH14 1DJ, UK

ABSTRACT

Two types of software are contrasted: output oriented and goal oriented. The typical development process of output oriented software is presented and compared with the approach necessary for developing goal oriented software. This paper describes the development of one specific goal-oriented system (an intelligent computer assisted language learning system) in order to highlight the differences in approach.

INTRODUCTION

Typical descriptions of ‘good practice’ in software development specify slight variations of one process. This process has been described under many names and using different terminology, for example Freeman [1] labels the phases of the software development life cycle as: needs analysis, specification, architectural design, detailed design, implementation, maintenance. The phases specified by Boehm [2] are: system feasibility, software plans and requirements, product design, detailed design, code, integration, implementation, operations and maintenance. These different descriptions do not represent significant differences in practice, but more in the way that practice is described. Essentially, most attempts to describe the development process include the following stages:

1. analysis of the problem and its environment
2. functional specification
3. high-level design
4. low-level design
5. coding
6. testing and debugging
For conceptually straightforward systems the development would be carried out in a linear fashion, and each stage completed before starting the next. This has been represented by the waterfall or cascade model of the development life cycle, first defined by Royce [3], and which still describes much of current practice. In reality however, the process is likely to be iterative and it is frequently necessary to return to an earlier stage and then rework some of the subsequent stages. Nevertheless it is not normally expected that major revisions will be necessary.

When Quality Assurance procedures are introduced into the development process these usually involve the specification of ‘milestones’ which must be completed and signed off at each stage; a person or a committee takes responsibility for each milestone being correct in the context of the previous one. Thus, for example, it should be possible to justify every aspect of the high-level design by tracing back to the functional specification. This possibility of justification is known as traceability. One unfortunate effect of this approach is that it can lead to a false assumption that all previous steps in the development are correct. This can reduce the likelihood of returning to a previous step, even when it would be appropriate, thereby leading to a more linear development process.

The fundamental problem in any software development process is to ensure that the final system actually satisfies the real (as distinct from stated) requirements. This problem has two main aspects. The simpler of these, which is however still far from trivia', is the testing or verification issue: how to be sure that the system as delivered complies with the functional specification. If this can be achieved then at least the contractual obligations have been fulfilled. The harder problem is that of determining whether the functional specification was correct in the first place. It is a common experience that, even though clients may sign off a functional specification as correct, they may still wish to change it at any stage during or after the development process. As a result a great deal of software is written which does not satisfy the real requirements, and a significant proportion of it is never used. Formal specification methods, using mathematical notation, have been developed in an attempt to overcome this problem. These have shown a reasonable level of success in removing ambiguity and contradictions from functional specifications, but have proven to be too costly for general use. However, it seems that even these methods would make little impact on the development of what is described below as goal-oriented software.

We believe that the distinction between goal-oriented software and output-oriented software is useful in explaining the different levels of difficulty in determining whether a system satisfies the requirements, and in selecting the most appropriate development method. In this paper we describe a different approach from the above for the development of goal-oriented software.
GOAL-ORIENTED SOFTWARE AND OUTPUT-ORIENTED SOFTWARE

All software is written to fulfil a goal of some sort. We make the distinction between goal-oriented and output-oriented software in terms of both the level of difficulty of moving from the goal to a specification of the required output, and the use which is made of that output. This distinction should be reflected in the development process adopted. The above description of the development cycle is appropriate for output-oriented software (as exemplified by typical data processing programs), however when developing goal-oriented software, investigation of the relationship between the needs of the user and the output necessary to satisfy these requirements is harder to determine, and thus the output is more difficult to specify.

In the case of output-oriented software the movement from goal to output is at least feasible. Thus we may wish to write a program with the goal of producing bank statements which contain information on current accounts, to meet customer requirements. After a small amount of customer research it should be quite possible to specify in detail the required output, and the correctness of the program can later be assessed relative to this. For such software the output is of intrinsic value and once it has been specified we would be unlikely to worry too much about the effect it has on users or customers.

At the other end of the spectrum we have goal-oriented software. Here it is impossible (or, at least, extremely difficult) to specify its functionality other than in terms of high-level goals. We are no longer interested in the output as a product, but as a means to influence the behaviour of the user (this is the goal). Examples of goal-oriented software include Intelligent Learning Environments (ILE) and the HCI components of interactive systems. The goal of the developer of an ILE may be to facilitate a student’s learning of a particular topic in algebra. The developer of such a system should think of output primarily in terms of the resulting interactions and consequent learning, rather than as output for its own sake.

‘Goal-oriented’ and ‘output-oriented’ represent positions at the extremes of a dimension of ‘goal-orientedness’. Most data processing systems are firmly positioned at the output-oriented extreme, while other applications such as learning environments tend to be more goal-oriented. Real-time process control systems tend also to be goal-oriented in that there is the goal of controlling the behaviour of equipment. However, specification of the required output is not as difficult as in systems designed to facilitate learning, because the link between system output and achievement of the goal is easier to establish and more obviously deterministic, even though it is the control of the behaviour of the process which is important rather than the output per se.
When developing goal-oriented software we need to consider system evaluation from two perspectives:

1. does the system produce the required output?
2. does this output result in the required changes in the user?

The first perspective is very little different from testing as usually understood and applied to output-oriented software. However, with goal-oriented software the second type of evaluation is also necessary. Indeed, it is this second type of evaluation which is the most crucial for determining the quality of a goal-oriented system, for example, an ILE can only really be assessed by evaluating improvements in learning or knowledge, i.e. cognitive changes in the user. Only if these changes are insufficient (as measured against expected learning outcome) is it necessary to return to reconsider system output.

DEVELOPING GOAL-ORIENTED SOFTWARE: AN EXAMPLE

In the previous sections we described the typical development process of output-oriented software, and claimed its unsuitability for use in creating software which is more goal-oriented. In this section we present, as an illustration, the development process of a specific piece of goal-oriented software: an intelligent learning environment to facilitate second language acquisition (SLA). The ultimate aim of this project is the production of more useful intelligent computer assisted language learning (ICALL) systems, i.e. educational software which knows both about the subject matter to be acquired by learners, and also about the way students learn languages. The goal is to aid students in their acquisition of a foreign language through the identification of more efficient methods to facilitate language learning. In order to achieve this, the most effective method is to first develop a prototype which illustrates the educational benefits of this approach, as the value of the system can only be assessed properly by examining it in use. The final (tested) prototype will then demonstrate guidelines for the development of good language software. It is the research and development of this prototype which is described here.
As stated above, the main aim of the prototype system is to provide guidelines for the development of future language learning software, as CALL systems have so far tended not to be greatly concerned with theories of SLA (thus imposing limitations on their effectiveness). Consequently it was not important to decide in advance which language should be used, but rather, the crucial consideration was the identification of issues important in the field of SLA. Once this had been achieved, an environment (including specification of language) could be created to take account of these factors; this environment could later be adapted for specific purposes. Thus the following design and development stages were undertaken. Initial work concentrated on surveying the SLA literature. Numerous issues influencing language learning are discussed in this literature, examples include: the effects of the age of learners, their personality etc. However many such questions are less suitable for treatment in the ICALL situation, and the problem was to identify issues which are both applicable to, and implementable in the ICALL environment. The following four factors were selected as fulfilling these requirements, and as sufficient to support the system's student model (described below):

1. acquisition order of the target rules
2. language transfer
3. language learning strategies
4. language awareness

It is believed that these issues are adequate to achieve the requirements of the 'functional specification'. This functional specification is not only described in terms of a set of desired outputs, but relates to facilitation of learning (it is therefore cognitive changes in the student which must be examined in order to evaluate the quality of the software, and not simply the program output).

The functional specification in this case has to be concerned with modelling the student because, as stated above, it is the facilitation of cognitive changes in the learner which is the purpose of the system. (The final student model compared to the original model will indicate the extent of learning.) Therefore a simple linear or branching program is insufficient; an intelligent system is required to capture the representation of the above issues. These four issues have been incorporated into a student model (see below). Briefly, the requirements are as follows:

1. **Acquisition order of target rules**: The model of the student reflects the typical acquisition order of the grammatical rules (see Bull, Pain & Brna, [4], Bull & Pain, [5].) This has been achieved by constructing a continuum to serve as the basis of the model; this continuum traces from one end the progress of a novice in the domain (clitic pronoun placement in European Portuguese), through the various stages in the developmental sequence to the other extreme of domain expert. In addition to this typical progression, an individual model of each learner is also maintained as it cannot be assumed that every student will follow this sequence exactly. Together these two aspects of the student model can be used to diagnose the causes of errors and anticipate future performance. This anticipation of performance, which is not normally explicitly present in learning environments, is a useful supplement to the contents of the current model.
when determining system action. Figure 2 illustrates this dual approach to student modelling:

![Current Model Diagram]

The current model contains information relating to the student’s present knowledge state - the more conventional component of the student model. The learning history is a record of the learner’s actual previous performance (which is expected in most cases to reflect the usual acquisition order of the criteria for clitic pronoun placement in European Portuguese). The future progress is based on the typical sequence that most learners follow (allowing contents of the current model to influence this section of the student model in cases where a learner has not adhered to the normal sequence of acquisition).

2. **Language transfer**: In addition to the necessary representations of knowledge and misconceptions relating to the target language, the student model also contains information on other languages known by the learner (these languages are currently restricted to English, Spanish and Catalan). This permits the system to identify problems of analogy with other languages by reference to the ‘background languages module’, thus it enables recognition of a wider range of errors.

3. **Language learning strategies**: Good use of learning strategies can result in vastly improved performance. The student model therefore records learning strategy usage (see Bull, Pain & Brna [6] for details) in order to prompt the use of effective strategies, and to suggest the use of alternatives if a student’s approach to learning appears unsatisfactory or inappropriate in some context. The following language learning strategies (drawn from O’Malley & Chamot, [7]) were selected as being the most suitable for treatment in an ICALL environment:

   **Metacognitive Strategies:**
   - organizational planning (of strategies), self-monitoring (production / visual / strategy / double checking), self-evaluation (performance / ability / strategy use).

   **Cognitive Strategies:**
   - resourcing, note-taking, grouping, summarization, deduction / induction, substitution, translation, transfer, inferencing.

   **Social Strategies:**
   - cooperation, question for clarification.

Inclusion of information in the student model about the use of language learning strategies enables the system not only to tackle specific domain issues, but also to take account of a learner’s approaches to learning and to deal with problems relating to carrying out the task. This is clearly an important consideration, but one which is usually ignored.
4. **Language awareness**: A learner’s awareness of language is enhanced by allowing the student access to all information held in the system, i.e. expert domain knowledge, different developmental states, transfer issues and learning strategy usage, and also by enabling the student to view the system’s representation of its assessment of the student himself with reference to these issues (i.e. information in the student model). The student will also be able to discuss all information with the system, and indeed, if he disagrees with any representation in the current student model, he will be encouraged to suggest modifications. The greater awareness resulting from such discussion should aid the student in noticing features in language encounters, and may be explicitly considered in cases where the learner has not yet automated this aspect of language. A further advantage in addition to the increased reflection that this cooperative model-construction promotes, is that the resulting student model is more accurate due to the student’s input. (See Bull, [8] for examples.)

The above descriptions demonstrate how current questions in the field of SLA are being considered and incorporated into the design of language learning software in order to increase its effectiveness, and therefore its quality. A further advantage of this approach to development is that it will help establish the extent to which these issues are relevant and useful in the ICALL situation; information which can then be used to guide further refinement to the system (e.g. to determine the relative emphasis of the different components - for example, how do students react to discussion of learning strategies? Do they answer system requests for information about their strategy use? Do they try out suggested strategies? Do they simply ignore all mention of learning strategies? - if students were found to react negatively to such discussion, interactions concerning learning strategies would be correspondingly reduced in the system). Thus development of the system is not normally linear, but is likely to consist of cycles until the relevance of the theoretical issues has been sufficiently investigated.

If the above factors are to be dealt with by the system, the conventional intelligent tutoring system (ITS) architecture is inadequate. The traditional ITS consists of 3 basic components. The **domain knowledge** comprises the information to be acquired by the learner. The **student model** is a representation of the system’s assessment of the individual student’s knowledge of the domain (including misconceptions). The student model is used by the system to inform the **teaching strategy module**, which seeks to select the most appropriate strategy for teaching the student based on information in the student model (i.e. what it knows about the learner’s current knowledge state, preferences, previous learning history, etc.)

In such conventional ITSs the responsibility for construction of the student model lies solely with the system, which must make appropriate inferences based on student input. It has already been seen from point 4 above that in the system described in this paper, this is not the case. Therefore in addition to the three modules presented above, the system must also contain a discussion component. Further, it requires representations of languages other than that comprising the domain knowledge, in order to account for language transfer. In addition, a learning strategy module is needed to enable the system to promote efficient
The artificial collaborator has access to all components of the system, and through this agent the student may inspect and discuss this information. The student may also suggest modifications to the contents of the current student model.
strategy use. The high level design of the system’s architecture based on these considerations is shown in figure 3.

In order to justify the consideration of these issues beyond the fact that they are cited in the general literature, research in the target domain was undertaken to obtain more precise information (see Bull & Pain, [5]). A summary of results is presented below.

The acquisition order for the rules of clitic pronoun placement in European Portuguese is currently being calculated based on a series of student exercises spread across five weeks. 47 students are being assessed, and the results of this study are expected to suggest the likely acquisition sequence for most learners. At present the acquisition sequence is based on a smaller study (see Bull, Pain & Brna, [4] for details of this estimated acquisition order). However, results of the on-going larger scale study are expected to produce a more accurate prediction of the typical acquisition sequence (which will need to be verified through testing of the system).

The importance of language transfer when learning Portuguese was assessed by questionnaire (20 students). Findings suggest that learners compare one of their background languages with Portuguese in cases where they had a good level of proficiency in the background language, and where this language was similar to the target Portuguese (Bull & Pain, [5]). This supports findings in the SLA literature (eg. Ringbom, [9] - proficiency, Kellerman, [10] - similarity) on which the system’s mechanism for calculating the order in which background languages should be checked for transfer was based.

An investigation of 16 students’ general language learning strategies was undertaken using Oxford’s Strategy Inventory for Language Learning (SILL version 1.5, Oxford, [11]). This study (described in Bull, Pain and Brna, [6]) revealed a varied range of strategy use by language students. A separate investigation of 20 students’ approaches to the completion of a specific Portuguese multiple choice exercise on pronoun placement indicated widespread unprompted use of the following strategies: resourcing, studying material prior to attempting exercise, guessing, checking work both during and after task completion. The fact that students use learning strategies justifies their inclusion in the ICALL system and representation in the student model. Moreover, student interest in improving their use of learning strategies (see below) offers further support for this approach.

Prototyping was necessary at this stage as the software must be tested on students as early as possible. At a high level, the functional specification is an ICALL system to effect specific (stated) changes in the learner. However, actually fulfilling this functional specification is no trivial task. For example, what output is required to prompt such change? It is not the output itself which is assessed to determine the effectiveness of the system, but the output should lead to the required cognitive gains, and it is these cognitive changes which are evaluated. However, not only must the software effect the required change in students’ mental states, but the approach adopted must also be acceptable to students. Therefore a top-
down, breadth-first implementation was undertaken, as this enables rapid
and repeated prototyping, which allows evaluation of the software at
appropriate stages. This is necessary because it is otherwise not possible to
predict the outcome of using the software; improving the software is not
merely a matter of changing the output information. Moreover, the same
input by two different students may be dealt with in a totally different
manner depending on the contents of each student model. Extensive
evaluation of the software is therefore essential to ensure that appropriate
cognitive changes occur through the whole range of cases.

The first such evaluation considered student reactions to the system’s
explicit reference to language transfer and learning strategies (Bull, Pain &
Brna, [6]). Sixteen students were involved in the evaluation, and were
selected according to their different degrees of experience in language
learning, and different learning strategy preferences. The aim was to assess
whether students with differing language backgrounds, and who use
different learning strategies vary in their attitudes to a system which
explicitly refers to their language background and learning strategies.

All students reported in the post-interaction interview that they
considered the system’s reference to other languages to be a positive aspect.
(This interaction was the one in which the learner and system jointly
construct the initial student profile which serves as the basic student model
to be updated during future interactions.) Except in one case, all stated that
they would appreciate explicit reference to cases of both positive and
negative transfer (to facilitate acquisition through positive comparison, and
to aid explanation of errors where negative transfer has occurred). The one
learner who was less keen on receiving information on transfer issues
would welcome it only in cases where reference to positive transfer was
involved, but not as a method of error explanation describing negative
transfer.

Of the 16 students, 13 found the subject of language learning strategies
interesting, while 3 did not. However, all students would be prepared to
try to change their strategy usage if there were more efficient alternatives
(even those who did not consider the subject of learning strategies to be
interesting).

Clearly learners do accept, and welcome this approach, providing initial
justification for further work on this technique. The next stage of the
evaluation will be to assess whether the approach actually results in
enhanced learning.

CONCLUSION

The distinction between traditional output-oriented software and goal-
oriented software has been described, and the need for different approaches
to the development of these different types of application has been stressed.
The development process of one particular piece of goal-oriented software
was examined in order to demonstrate the validity of the above claim.
The development cycle of output oriented software tends to be the following: 1. analysis of problem/environment, 2. functional specification, 3. high-level design, 4. low-level design, 5. coding, 6. testing/debugging. The first phase (1) of the development of goal oriented software is similar, as is the functional specification, although this is likely to be expressed in different terms. However the subsequent stage of high-level design is not so simple, as it is the achievement of the goal (here: improved learning) which must be assessed to validate the design, and not the system output (as with output-oriented software). Because the precise functioning of the high-level design components is hard to specify, low level design is even more difficult. As with output oriented software, coding, testing and debugging are clearly necessary. However, a major difference in approach is the need for prototyping of software in order to check that the goal is actually being met at each stage of development. Thus in the ICALL system presented here, the development process is as follows: 1. analysis of the problem. 2. specification of the goals. 3. high-level design. 4. prototyping. 5. testing and debugging prototype. 6. evaluating acceptability to, and effect on the user. Points 3 - 6 to be repeated as required.

ACKNOWLEDGEMENT

Work on the ICALL system described in this paper is funded by the Economic and Social Research Council, award no. R00429124033.

REFERENCES


