Recording the Reasons for Design Decisions

Colin Potts and Glenn Bruns

MCC Software Technology Program

Abstract: We outline a generic model for representing design *deliberation* and the relation between deliberation and the generation of method-specific artifacts. A design history is regarded as a network consisting of artifacts and deliberation nodes. Artifacts represent specifications or design documents. Deliberation nodes represent issues, alternatives or justifications. Existing artifacts give rise to issues about the evolving design, an alternative is one of several positions that respond to the issue (perhaps calling for the creation or modification of an artifact), and a justification is a statement giving the reasons for and against the related alternative. The model is applied to the development of a text formatter. The example necessitates some tailoring of the generic model to the method adopted in the development, Liskov and Guttag's design method. We discuss the experiment and the method-specific extensions. The example development has been represented in hypertext and as a Prolog database, the two representations being shown to complement each other. We conclude with a discussion of the relation between this model and other work, and the implications for tool support and methods.

Problem Statement

A software development method supports two aspects of designing: it provides a series of representations and analyses for different kinds of *artifacts* (i.e. design documents), and heuristic support for design *deliberation* (deciding *what* artifacts to derive, and why). Methods differ greatly, both in the rigor of their artifacts and analyses, and in the richness of their heuristics. Most methods, however, provide some degree of support for both aspects of designing.

The use of a method involves the production of *intermediate artifacts* - artifacts prior to executable code. These may include informal documents describing the functional specification of the system, architectural sketches, detailed designs, pseudo-code, structure diagrams, or formal specifications. The progression of a single development project can be regarded as a graph, in which the nodes are artifacts and the links are derivation paths (Figure 1). One interpretation of Figure 1 would be that three detailed designs are derived from and comply with a single architectural design. Traceability is the problem of recording and maintaining information about these derivations: specifically, an object in an earlier artifact (e.g. a required feature) must be demonstrably implemented by the objects contained in some later artifacts (e.g. modules), although a sequence of derivations may distribute a unitary object throughout the design. Conversely, objects in later artifacts must be traceable back to the point in the design history at which the information they embody was introduced. An example of this model of software development is transformational implementation [Broy and Pepper, 1981]. This is an extreme example in that the artifacts have formal semantics and the derivation rules are strictly defined, but less formal methods may also have the same basis. In JSD [Jackson, 1983], for example, the artifacts are more heterogeneous than in transformational implementation, they do not have a formal semantics (although some analysis is possible), there are fewer of them, and the derivations do not preserve correctness. Nevertheless, the progression from one stage to the next is relatively smooth and easy to understand.

Intermediate artifacts help document the design decisions made during a multi-phased development process. Early design artifacts can be much more useful to maintainers than the final code, in which the effects of an early design decision may have become difficult to trace. Without such a record a maintainer may repeat mistakes that were made by the original designer but not documented or may undo earlier decisions that are not manifest in the code. But although intermediate artifacts may help document design decisions, they can only do so in an indirect fashion. A

0270-5257/88/0000/0418$01.00 © 1988 IEEE

Recommended by: W. Riddle

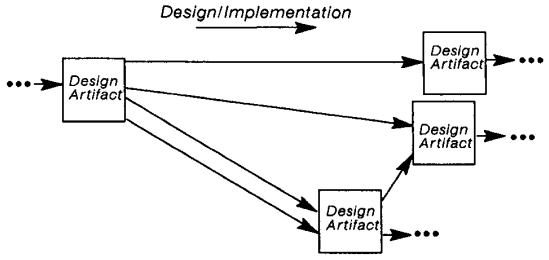

Figure 1: Schematic model of derivation of design artifacts

design artifact typically documents the *results* of a phase of designing, not the *process* followed.

The recording of design results (the nodes in Figure 1) and the recording of process (the arcs) are different development functions with different purposes. Many development tasks (especially implementing subsequent modifications) may require access to both kinds of information, but others do not. This suggests the need for two kinds of design documentation: documentation of the process (deliberation or rationale) and documentation of design results (artifacts). Recording rationale independently from artifacts has two advantages: the designer need not hunt inside an artifact for explanatory comments that may not be there, and considerations which impinge on many artifacts are recorded in one place (Figure 2). When the intermediate artifacts are viewed, either electronically or in the form of printed documents, it may be valuable for the rationale to be inserted into the artifact as a set of annotations. Conceptually, however, the rationale is separate: the artifact records the design; the rationale records the decisions made in producing the design.

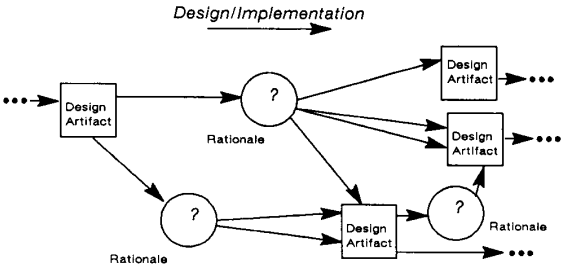

Figure 2: Schematic model of design deliberation and artifact synthesis.

Artifacts in a design language can be subjected to a range of formal analyses, and tools such as structure editors and consistency checkers can be developed even for relatively informal languages. Deliberation, on the other hand, is currently supported only by method manuals and training courses. Although this is unsatisfactory, it is not necessary to go to the opposite extreme by attempting to formalize the micro-structure of design deliberation or to implement intelligent design automation tools. Encoding design expertise and domain-specific knowledge may be desirable long-term goals, but much can be done immediately to support design deliberation with more modest semi-formal representations.

Overview

In this paper we outline a generic model for representing design deliberation and its relation to the derivation of method-specific artifacts. We apply the model to a 'vivisection' of Liskov and Guttag's [1986] design of a text formatter. In so doing, we treat the design process as a network consisting of deliberation and artifact nodes. We have represented Liskov and Guttag's design process as a hypertext network and as a Prolog database. We discuss this experiment and the extensions needed to provide method-specific tools to represent deliberation and artifacts.

We have adopted an *issue-based* model of deliberation. This is summarized in the entity-relationship diagram of Figure 3. The model has been influenced by Rittel's IBIS method for policy decision making [Rittel and Webber, 1973] and Conklin's [1986] proposed Design Journal. Existing artifacts, including requirements documents, give rise to *issues* about the evolving design. For example, if the artifact is an informal specification of a text formatter, the

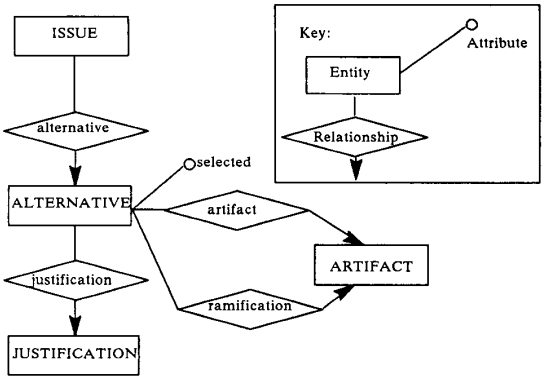

Figure 3: Data model for design deliberation

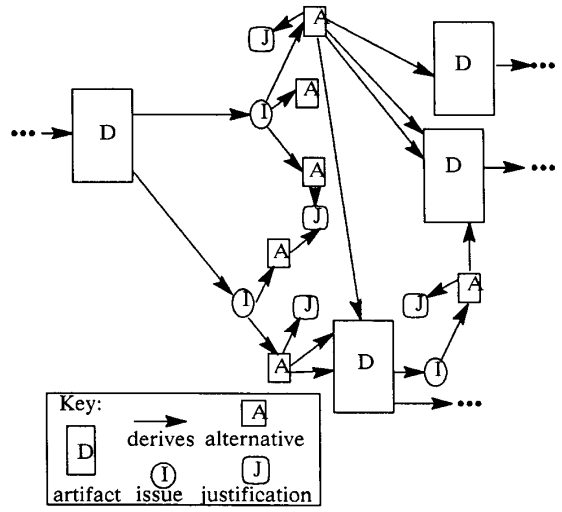

Figure 4: Expanded schematic model of design deliberation and artifact synthesis

issue may arise 'how is the input text going to be read?' An *alternative* is one of several positions that respond to the issue. For example, the alternative 'we need a procedure to read lines' is one possible response to the above issue. Not all alternatives directly suggest the need to create new artifacts; many reflect the need to modify or refine existing artifacts, or state that no design changes need to be made. A *justification* is a statement giving the reasons for and against selecting the related alternative; for example, 'we should read the input line-by-line because there are two kinds of lines (text lines or command lines), which must be treated differently'.

A skeletal example of the deliberation process at work is shown in Figure 4, which is obtained by exploding the 'deliberation' nodes of Figure 2 according to our data model.

To be put into practice, this approach must be specialized for a particular design context; that is, a particular design method, application domain, or set of solution technologies. The specialization of the model for a specific *method* is illustrated in the next section.

Example

We illustrate how deliberation can be integrated into an existing design method by discussing a worked example in some detail. In doing so, we try to point out what is specific to that example and what is generic. The example is the development of a simple text formatter as documented by Liskov and Guttag [1986, Chapter 13].

Example Method

Liskov and Guttag's book is an exposition of an approach to software design based on information hiding. Henceforth, the method will be abbreviated as 'L&G'. In L&G, design artifacts take the form of abstractions with well-defined interfaces and hidden bodies. Depending on the stage of development and the amount known about an abstraction these properties may be defined formally or informally; for example, a data abstraction may be defined formally as an algebraic specification, and a procedural specification may be defined in terms of formally stated preconditions and postconditions. On the other hand, the artifacts may have a formal *structure*, while their *contents* are specified informally; for example, the operations of a data abstraction, and the preconditions and postconditions of a procedural specification can both be defined by informal text.

In addition to the specialization of the artifact types, the data model of L&G includes one new object type: an application domain *task*. A task is anything performed by a procedural abstraction. L&G includes heuristics for inventing suitable abstractions to accomplish tasks. For example, the 'read input' task of the text formatter (the formatter being a high-level procedural abstraction) could be encapsulated in a variety of ways:

- an input document buffer data abstraction;
- the input document could be read incrementally in application-sized chunks (for example, line-by-line) with a procedural abstraction hiding the details of character input;
- there might be no encapsulation of the task, the text formatter reading the text by direct calls to primitive input operations.

In our terminology, making such a decision amounts to resolving an *encapsulation issue*. Encapsulation issues arise often during an L&G development: many of the decisions documented by Liskov and Guttag are resolutions of issues of this type. An encapsulation issue is a special class of issue that is concerned only with the possible encapsulations of a concept (e.g. a task) within an abstraction artifact (e.g. a procedure). Rules apply to encapsulation issues that do not apply to issues in general: for example, an encapsulation issue is raised for every task. The enlarged data model, incorporating specialized artifact and issue types and the method-specific task object type is shown in Figure 5.

Example Application

We have chosen Liskov and Guttag's text formatter development, because they provide an excellent and (apparently) thorough account of the design deliberations in-

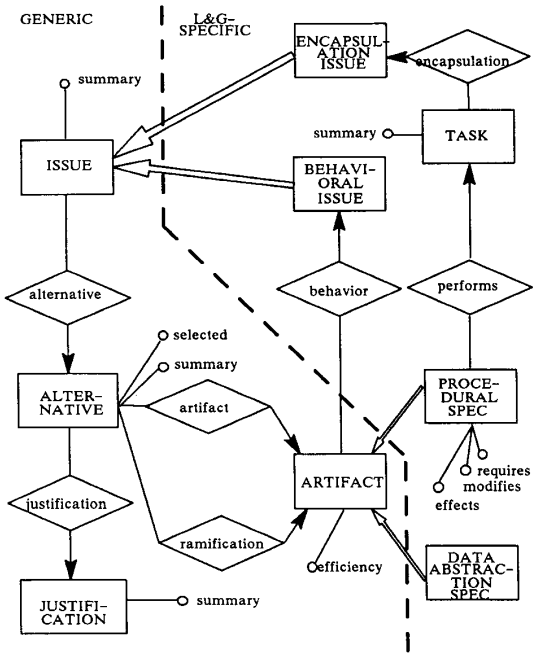

Figure 5: Data model for L&G

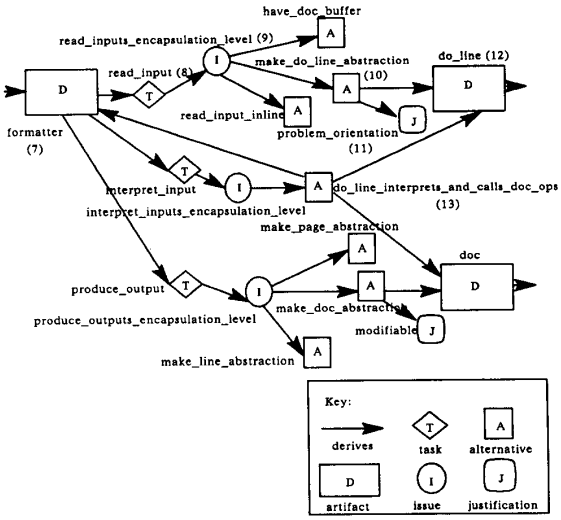

Figure 6: Expanded schematic model of design deliberation and artifact synthesis

volved in designing using L&G. In fact, because of their pedagogical objectives the example is sanitized: it contains no mistakes or abandoned design paths, and the deliberations that are documented are those that Liskov and Guttag feel make the final design easy to understand and modify, not those that may really occur during a design process.

The formatter is a simplified version of the UNIX[1] text formatter nroff. It reads documents that contain embedded command lines. A command line starts with a period and is followed by a command name. There are only a few commands, and the syntax for commands is very simple. The output of the formatter is the document formatted according to the commands embedded in the input file, and according to built-in parameters specifying the page length, margin widths, and so forth.

Liskov and Guttag give an implementation of the formatter consisting of about 300 lines of CLU. Thus the example is small but non-trivial.

[1] UNIX is a trademark of AT&T Bell Laboratories

Text formatter development

An overview of the part of the design process we consider is shown in Figure 6. Compared with the more schematic Figure 4, Figure 6 is populated with concrete, L&G-specific nodes. Some of these nodes are numbered: the numbers refer to later figures depicting the nodes' contents.

Some deliberation has taken place prior to the point at which we start our analysis in this paper of Liskov and Guttag's design. Artifacts have been synthesized that document the requirements, and issues arising out of these have been posted. We start our analysis with the identification of the primary abstractions of the problem. This is a standard L&G issue that is always addressed early in the design.

According to Liskov and Guttag there is one primary abstraction in the text formatter - the **formatter** procedure. Liskov and Guttag's specification for this procedure is given in Figure 7. We have changed their notation slightly to make every artifact and deliberation node have a record structure in which the fields correspond to the relations (or their inverses) and the attributes of the data model. The changes are purely notational and the deliberation nodes and artifacts are faithful to Liskov and Guttag's design.

The rest of this section contains examples of the nodes types specified by the data model.

```
formatter = PROC (ins, outs, errs: stream) SIGNALS
(badarg(string))
MODIFIES:
    ins, outs, errs
EFFECTS:
    If ins is not open for reading, or outs or errs is not open
    for writing, badargs(s) is signaled, where s identifies an
    argument that was opened improperly (e.g. badarg
    "input stream")). Otherwise format proceeds as
    described in the text [i.e. Liskov and Guttag(1986) pp.
    271-273], taking input from ins and producing output on
    outs and error messages on errs. Ins, outs and errs are
    closed before returning.
TASKS:
    read_input
    interpret_input
    produce_output
EFFICIENCY:
    Where n is the number of characters in ins, time is
    O(n), space added (the storage for outs) is O(n) and
    temporary space is much less than n.
```

Figure 7: Procedure specification for fomatter

Having identified the major abstractions, L&G proceeds by identifying the principal tasks accomplished by a procedural abstraction and considers how these are encapsulated. Some may require the invention of 'helper abstractions'; others may be accomplished by refining the specification of abstractions already existing in the design. In the formatter example, three task nodes have been created and corresponding encapsulation issues have been raised addressing each task.

One of the tasks, **read_input**, is documented in Figure 8, and its corresponding encapsulation issue, **read_inputs_encapsulation_level**, is shown in Figure 9.

```
read_input = TASK
DONE-BY:
    format
SUMMARY:
    formatter reads input
```

Figure 8: A task node.

```
read_inputs_encapsulation_level = ISSUE
CONCERNING:
    read_input
SUMMARY:
    How should the format task read_input be encapsulated?
ALTERNATIVES:
    read_input_inline
    have_doc_buffer
    make_do_line_abstraction
```

Figure 9: An issue node

```
make_do_line_abstraction = ALTERNATIVE
SELECTED
SUMMARY:
    Make a do_line procedural abstraction to process each
    input line.
RESPONSE-TO:
    read_inputs_encapsulation_level
ARTIFACT:
    do_line
```

Figure 10: A design alternative node

```
problem_orientation = JUSTIFICATION
CONCERNING:
    make_do_line_abstraction
SUMMARY:
    1. The problem is line-oriented (lines are either text or
       commands)
    2. Reading the entire document into a buffer contravenes
       one of formatter's efficiency constraints on
       temporary space.
```

Figure 11: A justification node

Liskov and Guttag consider three alternative ways to encapsulate the **read_input** task; to accomplish it inline within the formatter itself (that is, the implementation of **formatter** should directly call primitive character input operations), to read the entire document into a buffer, or to have a new abstraction, **do_line**, read the document one line at a time. The third alternative, the one calling for a **do_line** procedure, is chosen (Figure 10). Its justification is given in Figure 11. In response to this chain of deliberation, a new artifact is created – the procedure specification for **do_line** (Figure 12). **Formatter** will call this procedure whenever it needs to read and process the next line in the input document.

```
do_line = PROC (ins: stream, d: doc, errs: stream) SIGNALS
    (all_done)
REQUIRES:
    can_read(ins) & can_write(errs) & d has not been
    terminated.
MODIFIES:
    ins, errs, d.
EFFECTS:
    If ins is empty, signals all_done. Otherwise processes
    one input line from ins to d as defined in the
    specification of formatter, writing error messages on
    errs. The entire line is processed including the end of
    line character.
EFFICIENCY:
    The time and space taken to scan and parse a line
    should be proportional to the number of characters in
    the line.
```

Figure 12: A procedure specification (do_line)

Turning to the issue **produce_outputs_encapsulation_level** (refer back to Figure 6), we see that three alternatives are considered, each calling for the synthesis of a data abstraction encapsulating a body of text: **make_page_abstraction**; **make_line_abstraction**; and, **make_doc_abstraction**. The decision is made that the entire document is the best abstraction for producing output. The justification given is that this is the choice that makes the rest of the design most immune to plausible future modifications; likely future modifications (perhaps by comparison with other text formatters) include new commands to format bodies of text that straddle page boundaries. As a result of this decision a **doc** data abstraction is created.

Detection and recording of ramifications

The third issue, **interpret_inputs_encapsulation_level**, appears similar to the input and output issues. The reasonable alternatives for the encapsulation of interpretation of inputs are fairly constrained given the decisions concerning the encapsulation of input and output task. Only one alternative is seriously entertained – **do_line_interprets_and_calls_doc_ops** (Figure 13). That is, **do_line** will interpret the input, formatting text lines and interpreting command lines, and will do this by means of appropriate operations in **doc**. Liskov and Guttag do not consider alternatives or justify this decision other than to say that it is a simple solution.

Several ramifications are noted by Liskov and Guttag (see Figure 13). All three of the artifacts produced so far are potentially in need of modification. For example, the input to **do_line**, the behavior of **formatter** and the operations contained within **doc** have all been constrained by this decision.

```
do_line_interprets_and_calls_doc_ops = ALTERNATIVE
SELECTED
CONCERNING:
    interpret_inputs_encapsulation_level
SUMMARY:
    do_line interprets input and calls appropriate doc
    operations to format text.
RAMIFICATIONS:
    do_line must be passed a doc object.
    formatter creates output.
    formatter finishes output.
    doc has create op.
    doc has terminate op.
```

Figure 13: An alternative with ramifications.

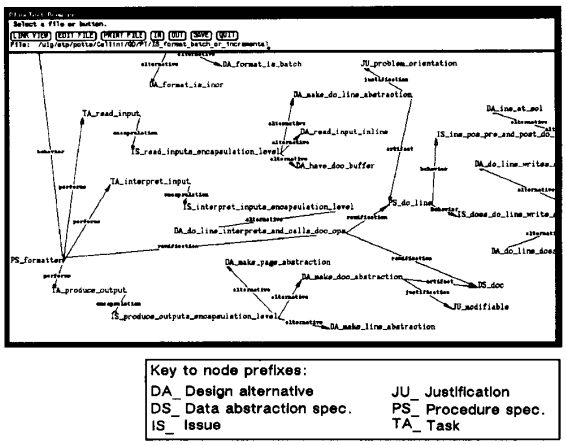

Figure 14: Planetext browser screen for formatter vivisection

Hypertext and rule-based implementation

An amalgam of formal structure and semi-structured, largely textual, content is well-suited for representation in a hypertext system [Conklin, 1987]. We have implemented the Liskov and Guttag formatter development, of which the the portion discussed in this paper constitutes just a part, in Planetext, a prototype hypertext system (see Conklin [1987] for an overview of Planetext). Planetext includes a graphical browser that enables the user to view the network as a whole and to move around it. Figure 14 shows a Planetext browser window. Apart from differences in layout, and the fact that node types are encoded by two-letter prefixes, it is similar to Figure 6. The individual nodes in the deliberation network are represented by single Planetext nodes.

In Planetext, nodes are represented by text files. Nodes have associated link files that collectively represent the network. Figure 15 is a Planetext edit window containing the alternative node **make_do_line_abstraction** (cf. Figure 10). To link a Planetext text node to others in the network, the user selects the 'LINK' option and specifies the link type (e.g. 'justification') and the name of the destination node (e.g. 'JU_problem_orientation'). The cursor position at the time the link was created is taken to be the point from which the link emanates. Thus a link's true source may be a component inside the node (for example a field or keyword), not the node as a whole. The entries '{justification}' and '{artifact}', have been placed in the node by Planetext, to indicate the sources of the links emanating from the node.

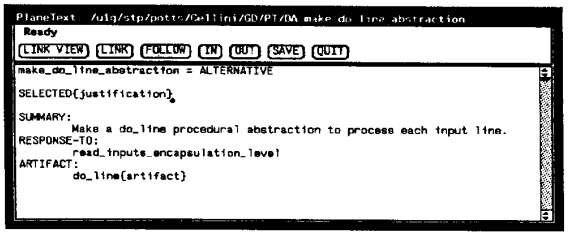

Figure 15: Planetext edit window showing emanating links

Planetext has no mechanism to check whether a network complies with a data model. Simple networks can be inspected visually, but this becomes difficult and error-prone for more complex networks. We would like to be able to check, for example, whether there are instances of issues which have been responded to with only one alternative (like **do_line_interprets_and_calls_doc_ops** above), whether there are instances of selected but unjustified alternatives, and so on. This can be accomplished by translating the Planetext link file information into Horn clauses and using a Prolog interpreter to perform the analyses as queries.

Augmenting a hypertext system with a general-purpose inference mechanism has the advantage that analyses can be performed by formulating the appropriate query. It is also possible to formulate special transitivity rules to define virtual links that could not be represented explicitly in the hypertext network without incurring a substantial maintenance overhead. For example, one artifact can be said to be derived from another if it arises out of deliberations concerning the originating artifact. Intuitively, we would like to be able to 'telescope' deliberation nodes out of existence to produce a view of the network that only contained derivations between artifacts (cf. Figure 1). This is trivial in Prolog. Thus (in sugared Prolog syntax):

```
derived-from(Artifact1, Artifact2) if
        raises(Artifact1, Issue) &
        alternative(Issue, Alt) &
        selected(Alt) &
        artifact(Alt, Artifact2)
```

where, in L&G:

```
raises(Artifact, Issue) if
        procedural-abstraction(Artifact) &
        performs(Artifact, Task) &
        encapsulation(Task, Issue)
```

and:

```
raises(Artifact, Issue) if
        behavior(Artifact, Issue).
```

Notice, that method-specific knowledge of node types and relationships is vital in defining these kinds of rules. Simple reachability is not sufficient to define derivations: links must be of the correct type.

Initially, separate Planetext and Prolog descriptions of the text formatter development were developed. A translation program has been developed that reformulates the Planetext linkfile information as a Prolog database. Periodic analyses can thus be performed on the network. Currently, however, Planetext does not have a Prolog interface, so changes to the network are not automatically reflected in the Prolog version.

Discussion

Conclusions

We have described a simple model of design deliberation, and its relation to the derivation of design artifacts. Its simplicity and the separation of the representation of artifact and rationale facilitate the understanding of decisions made during a design effort. The model has been easily tailored to a specific method. This suggests that it is possible to systematize design deliberation in a fairly uniform way. A design method provides a set of representations for the artifacts produced at each stage and a corpus of standard issues, candidate alternatives and justifications that systematize some of the deliberation entered into at each stage.

Related Work

Our approach can be compared to 'process modeling' – the development of descriptive models of software development practices (Potts [1984], Dowson [1986]). Because software development is very complex, with organizational factors overlaying the purely technical, some published process models are very complicated. Yet few process models have become the basis for project support environments, and those that have are among the simplest and most capable of specialization for particular contexts (e.g. Stenning [1986]). Our deliberation model has been developed with the same goals of simplicity and tailorability in mind.

L&G seems to be representative of informal, but systematic methods which can be supplemented by the semi-structured representation of deliberation. Semi-structured deliberation can also be applied in the case of more formal methods. For example, Bjorner [1987] describes a formal approach to software development in terms of 'software development graphs'. (His Figure 3.1 is identical in kind to our Figure 1). Bjorner has in mind artifacts and derivation rules constructed according to the principles of VDM [Jones, 1986]. But having a formal (or 'rigorous') method does not, of course, completely free the designer from the

need to deliberate about different potential refinements of a specification or how to discharge proof obligations. VDM-specific deliberation occurs between the artifacts in a software development graph in an identical fashion to the deliberation occurring between the L&G artifacts in our vivisection.

Wile [1983] has developed a system (POPART) and a language (Paddle) for describing as a 'meta-program' the transformational development of a program. A Paddle program is a formal description of the program development that can be used to explain the development to subsequent maintainers. Using Paddle, the designer can write commands to perform general-purpose problem solving methods (e.g. 'divide and conquer'). Our approach is complementary. Support like POPART and Paddle will be necessary for transformational methods to be usable in the development of large systems, as it effectively abstracts away from source-to-source transformations to the fulfilment of larger design goals. However, a meta-programming language is not the best vehicle for documenting *why* a problem should have been tackled in a specific way; for example, Paddle cannot be used to explain why a design goal was 'divided and conquered' in the chosen fashion, or how the goal itself arose.

In addition, it should be noted that transformational techniques and other formal methods are only applicable once a formal specification has been written. In a real project, this is a significant development effort in its own right. Many analysis and design decisions must be made prior to that point, and informal intermediate artifacts will be produced *en route* (cf. Finkelstein and Potts [1986]).

The design we have vivisected in this paper was produced for educational reasons and includes no mistakes or changed decisions. Thus it is atypical of design processes in real development projects. It remains to be seen whether the explicit recording of design rationale is helpful or intrusive during *constructive* design, or whether it is best done in order to 'fake' a rational design process after the fact [Parnas and Clements, 1986]. We have also chosen a level of granularity in representing deliberation that may not be appropriate for larger projects or other design methods. These are empirical questions.

Non-intrusive tool support is vital. To record design deliberations, the designer must create many objects – issues, alternatives, and justifications. Thus tools must help the designer create and use such objects easily and maintain them behind the scenes. One mundane problem that must be overcome is the naming of nodes. The nodes in the formatter development are all given long, mnemonic names. If every decision made by a designer were to require the creation and naming of five or six deliberation nodes, there would be little time left for designing. In the same vein, redundant data entry should be minimized. For example, much of the nodes' contents in the examples above could be deduced by a tool from the network's structure.

Some of these requirements are fulfilled by existing general-purpose hypertext systems. Undoubtedly, hypertext is very suitable for maintaining large, non-linearly related sets of documents. Thus hypertext should become a standard vehicle for the preparation and maintenance of system documentation. There have been previous attempts to use hypertext systems to capture structural relationships between design components [Biggerstaff, 1987; Petersen, 1987]. However, these studies were concerned with the representation and inter-relationship of design documents, not with the process of deliberation that gives rise to them. For the latter purpose, a deliberation support tool would have to contain a fairly general query and inferencing capability.

In the foreseeable future we believe that the best route for introducing general inferencing is by representing design artifacts and deliberations in semi-formal records and defining rules that utilize this purely structural information. In this we are following the example of Malone *et al.* [1987], who describe an electronic mail system based on 'semi-structured' messages. Malone's system has superficial knowledge of message types and their attributes; it has no deep knowledge about the domain of office work, but it allows users to define rules governing what categories of message they wish to receive and what priority to accord to them by purely structural criteria. Malone et al. identify several benefits attributable to semi-structured forms in the context of computer-mediated coordinated work, benefits which appear directly transferable to a design application:

- they help in composition of messages: confronting the designer with field names is a simple way to help him or her not to forget to record some categories of information;
- they help in sorting, selecting and assigning priorities to messages: the structure is sufficient to be able to formulate useful semantic criteria for retrieving information;
- they can be analyzed so that subsequent actions (e.g. forwarding or acknowledging a message) can be invoked automatically and they suggest likely responses to other messages: a design agenda management system could easily be based on superficial analysis that helped a designer keep track of the completeness of a design, automatically creating placeholder nodes or modifying related nodes when appropriate (e.g. the creation of a placeholder artifact stemming from a 'make_' alternative).

Future Plans

It is intended that the work reported in this paper is the first step in the development of more deliberation-based design methods, new representation schemes, and more effective tool support for design problem solving. It needs to be generalized in many ways by retracting the simplifying assumptions we have made, by investigating:

- deliberation support for synthesis-oriented methods other than L&G;
- the design of larger systems than text formatters;
- constructive design, as opposed to *post hoc* vivisection;
- the design of reactive, as opposed to input-process-output systems.

Another extension of deliberation-based design is in design planning. Studies of designers (e.g. Guindon, Krasner and Curtis [1987]) have revealed that many design 'breakdowns' occur because of cognitive limitations - for example designers forget to return to design goals they have postponed, or while working on one part or one stage of the design they cannot adequately record opportunistic design decisions affecting another part. Many of these breakdowns could be avoided by incorporating standard deliberation and planning structures in method support tools. The designer could then incrementally generate and manage his or her work agenda. Such tools would assist in the posting of issues for future consideration, or the creation of place-holders (for example, creating null artifacts in response to 'make_' alternatives).

We believe that generic models are limited in scope, and have discussed how the model of deliberation presented here can be tailored to a specific design method. Two other kinds of tailoring would also be useful: those for specific application domains (e.g. elevator scheduling, or library inventory management); and those for established solution technologies (e.g. security, databases, or communications). *Application domain* knowledge is an important resource in design decision making, because some issues crop up whenever a system is designed for a specific application. Fickas [1987] has demonstrated that some application-specific knowledge can be represented sufficiently formally for a requirements analysis program to be able to hypothesize problems with a specification. Most application domains are difficult to codify, however, because they are not part of any formal discipline. Deliberation specific to a *solution technology* could be more easily systematized. Solution technologies give rise to recurring patterns of deliberation, irrespective of the application. For example, there are a limited number of protection strategies, and each gives rise to a set of basic issues with standard alternatives. The authors are currently investigating how this knowledge can be applied during requirements analysis [Bruns and Potts, 1987].

Acknowledgements

Don Petersen implemented the program to translate Planetext outlink files into a Prolog database. Jeff Conklin, Susan Gerhart and Don Petersen made many constructive comments on an earlier version of this paper.

References

Biggerstaff, T. *Hypermedia as a tool to aid large scale reuse*, MCC Technical report STP-202-87, 1987.

Bjorner, D. 'On the use of formal methods in software development' *Proc. 9th Int. Conf. Software Eng.* IEEE Comp. Soc. Press, 1987.

Broy, M. and P. Pepper 'Programming as a formal activity' *IEEE Trans. Software Eng., SE-7(1)*: 14-22, 1981.

Bruns, G. and C. Potts *Requirements by Analogy,* MCC Technical Report, STP-258-87, July, 1987.

Conklin, J. *A theory and tool for coordination of design conversations*, MCC Technical Report, STP-236-86, July, 1986.

Conklin, J. *A survey of Hypertext*, MCC Technical Report, STP-356-86, Rev. 1, February, 1987. A shortened version of this report has been published as 'Hypertext: an introduction and survey' *IEEE Computer 20(9)*: 17-41, September, 1987.

Dowson, J. (ed.) *Iteration in the Software Process: Proc. 3rd Int. Software Process Workshop*, IEEE Comp. Soc. Press, 1986.

Fickas, S. 'Automating the analysis process: an example' *Proc. 4th Int. Workshop on Software Specification and Design*, IEEE Comp. Soc. Press, 1987.

Finkelstein, A.C.W. and C. Potts 'Structured common sense: the elicitation and formalization of system requirements' in D. Barnes and P. Brown (eds.) *Software Engineering 86*, Peter Peregrinus, 1986.

Guindon, R., H. Krasner and B. Curtis 'Breakdowns and Processes during the Early Activities of Software Design by Professionals' *Proceedings of Second Workshop on Empirical Studies of Programmers*, Ablex. 1987.

Jackson, M.A. *System Development*, Prentice-Hall, 1983.

Jones, C.B. *Systematic Development Using the VDM Approach*, Prentice-Hall, 1986.

Liskov, B. and J. Guttag *Abstraction and Specification in Program Development*, MIT Press, 1986,

Malone, T.W., K.R. Grant, Kum-Yew Lai, R. Rao and D. Rosenblitt 'Semistructured messages are surprisingly useful for computer-supported coordination' *ACM Trans. Office Inf. Sys. 5(2)*: 115-131, 1987.

Parnas, D. and P.C. Clements 'A rational design process: how and why to fake it' *IEEE Trans. Software Eng., SE-12*: 251-257, 1986.

Petersen, D. *Software design capture,* MCC Technical Report STP-138-87, May 1987.

Potts, C. (ed.) *Proc. Int. Software Process Workshop*, IEEE Comp. Soc. Press, 1984.

Rittel, H. and M. Webber 'Dilemmas in a general theory of planning' *Policy Sciences, 4*, 1973.

Stenning, V. 'An introduction to ISTAR' in I. Sommerville (ed.) *Programming Support Environments*, Peter Peregrinus, 1986.

Wile, D.S. 'Program developments: Formal explanations of implementations', *Comm. ACM, 26(11)*: 902-911, 1983.