Decision-driven Maintenance

Filippo Lanubile and Giuseppe Visaggio

Dipartimento di Informatica, University of Bari
Via Orabona 4, 70126 Bari, Italy

Summary

This paper presents our approach to design recording aiming to facilitate the impact analysis of changes in data, functions, or the external environment. A whole software system is represented as a web which integrates the different work products of the software life cycle and their mutual relationships. A traceability relationship associates the objects with each other so that impact analysis can be performed. Internal traceability is provided by semantic links between software objects representing the work products of a development phase, while external traceability is assured by the cognitive links between software objects from different phases. System understanding is supported by the decisions which are involved in the transformation process. The history of these decisions is retained over time so that previous decisions can be examined for maintenance and reuse activities. The approach has been implemented through a Traceability Support System, a maintenance tool which combines the characteristics of program abstractors, project databases and design rationale capture tools. The approach and the tool also both support traceability in heterogeneous systems, which have subsystems implemented on different platforms. Finally, analysis is made of the results of an empirical investigation carried out to assess the approach.

1. INTRODUCTION

Changeability is an inherent property of modern software systems, together with complexity, conformity and invisibility (Brooks, 1987). A useful software system is required to evolve in order to incorporate new functional and data requirements which derive from experiencing the system itself. Laws, business rules and habits change, thus forcing software to adapt to new user environments. The need to exploit new technologies, programming environments and design methods also causes requests for change.

Owing to high development costs, these changes tend to be incorporated into existing systems, thereby lengthening the system life cycle. Papers on software evolution, such as (Lehman, 1989; Lehman and Belady, 1985), have stated that the most important aspects to be controlled are: the system complexity, the available documentation, the informal knowledge acquired through experience, and the accessibility to both documentation and knowledge.

The wide variety of available processors, peripheral devices, operating systems, database management systems, graphical user interfaces, and programming languages causes programmers to implement the same software system on different platforms, or to build a heterogeneous system upon components from different hardware and software technologies. This heterogeneity further complicates the maintenance process.

Several solutions have been proposed for organising maintenance support tools around information repositories. These solutions can be divided into three main classes: program abstractors, project databases, and design rationale.

Program abstractors, such as CIA (Chen *et al.*, 1990), DATA_tool (Canfora *et al.*, 1992) and CARE (Linos *et al.*, 1993), summarise the structure information of a program in a database, according to a conceptual data model. Objects and links are automatically created in the database during analysis of the source code or the CASE repository. A user can browse the database and perform dependence analysis. The main advantage of these tools is that automation is fully supported so that their use does not

represent a burden for the software engineer. However, the stored information lacks semantics and belongs to a single life-cycle phase, often the coding phase.

Project databases, such as PACT (Simmonds, 1989), ESE (Ramamoorthy *et al.*, 1990), and DIF (Garg and Scacchi, 1990) model all kinds of information relevant to the software life-cycle in terms of objects and the relationships between these objects. Version control capability is also provided to keep track of the system as it evolved during its life. Impact analysis is supported by using traceability relationships to detect changing artifacts. However, this information alone is not sufficient to satisfy the maintainer's needs as it does not explain why things have been done and what alternatives have been discarded.

Design rationale capture tools keep track of the designer's or maintainer's reasoning, thus allowing the transfer of knowledge to new maintainers. Examples can be found in OSC (Arango *et al.*, 1991), D-HyperCase (Wilde *et al.*, 1991), SYBIL (Lee *et al.*, 1991), MACS (Desclaux, 1992; Georges, 1992), and DRCS (Klein, 1993). These tools differ as regards the decision representation model, the computational services offered and the capability of maintaining the decision history during the evolution of the system. The information provided by these tools, if kept up to date, is precious for software comprehension, modification and reuse. But the danger of inconsistencies increases if rationales are managed independently from the work products of the software life-cycle. Although these tools link decisions to software artifacts they are still isolated, as relationships between objects from different life-cycle phases are not made explicit in the absence of alternative transformations, and direct relationships between objects of the same work product are neither automatically created nor represented.

The system proposed in this paper serves the purposes of all the above tools. The functionalities of the three kinds of information repositories are complementary provided for the management of an all-inclusive base of information that should fully document a system as it evolves.

The paper focuses on the integration of the different representation models of a single software system. Traceability between different views of a software system is assured by a web of decisions, which define the alternatives, individuate the choices, and supply the relative justifications. The modeling of rationales represents a compromise between formality and usability. The former is necessary to provide automatic services which compensate for the extra workload of rendering the decision-making process explicit. The latter must be preserved when dealing with large systems which must have a long life. In this case, the concepts behind decisions taken in previous years must be easily understandable to justify the effort of maintaining the database consistent and traceable. Although descriptions are stored in natural language, we have categorised them to achieve a satisfactory degree of automation when searching for stored decisions.

The different representations of a software system and the related decisions are incorporated into a tool, the Traceability Support System (TSS), which makes the maintenance and reuse of heterogeneous systems possible.

To illustrate our approach, in section 2 we first discuss the use of multiple views of a software system. Section 3 defines the model dependency descriptor, representing the decision web. Section 4 describes the TSS and section 5 shows its use. Section 6 reports the results of an empirical evaluation of the model dependency descriptor. The last section suggests some conclusions and describes future work.

2. SOFTWARE SYSTEM MODELING

There are many aspects in a software solution which are independent of the limitations imposed by the programming system adopted (programming language plus run-time environment). The services required and the need for quality attributes tend to survive the technological innovation, while the hardware and software technologies are subject to frequent changes, so that a system may exist in different versions and variants. In order to reduce the cost of development and maintenance, we must capitalise the knowledge which is produced during the development of a software system by saving the core of the solution and excluding the insignificant details. Brooks in (1987)

describes the essence of a software system as being made of data objects, relationships between data objects, algorithms, functions and invocations of functions. These conceptual constructs are applied without regards to a particular representation.

In this paper, a software system is viewed as a collection of software models: requirements specification, essential design, and language-oriented design. The models correspond to the products of different development phases: analysis, design and implementation. Our intention is to emphasise the most significant information which can be collected during the design phase. In fact, we propose the separation of the initial specification of a software structure, called *essential design*, from the final specification, called *language-oriented design*. First, an essential design is produced, applying the principles of software engineering (separation of concerns, modularity, abstraction, generality, and anticipation of change) on the basis of the program objectives and the required quality attributes. After having evaluated the hardware-software platform, a language-oriented design is produced through specialisation of the essential design. In this way, the language-oriented design is suited to the programming environment. The essential design aims to match the desired functional characteristics to a suitable software quality model. It considers the programming environment as technologically perfect with respect to software engineering principles. As a consequence, the essential design is the best solution a designer can achieve. Each adaptation to the implementation platform, which determines the form of the language-oriented design, must tend towards the same level of quality.

The whole software system is stored in a *design database* which provides a record for the process ranging from taking a requirements specification to producing a description of an implementation from which source code is developed.

Figure 1 represents the main ER diagram, which shows the composition of a system and the classification hierarchy of software objects. A software system is viewed as a collection of software objects whose composition may form a hierarchy of subsystems. The s*oftware object* entity is broken down into three disjoint subclasses: *requirements specification*, *essential design*, *language-oriented design*.
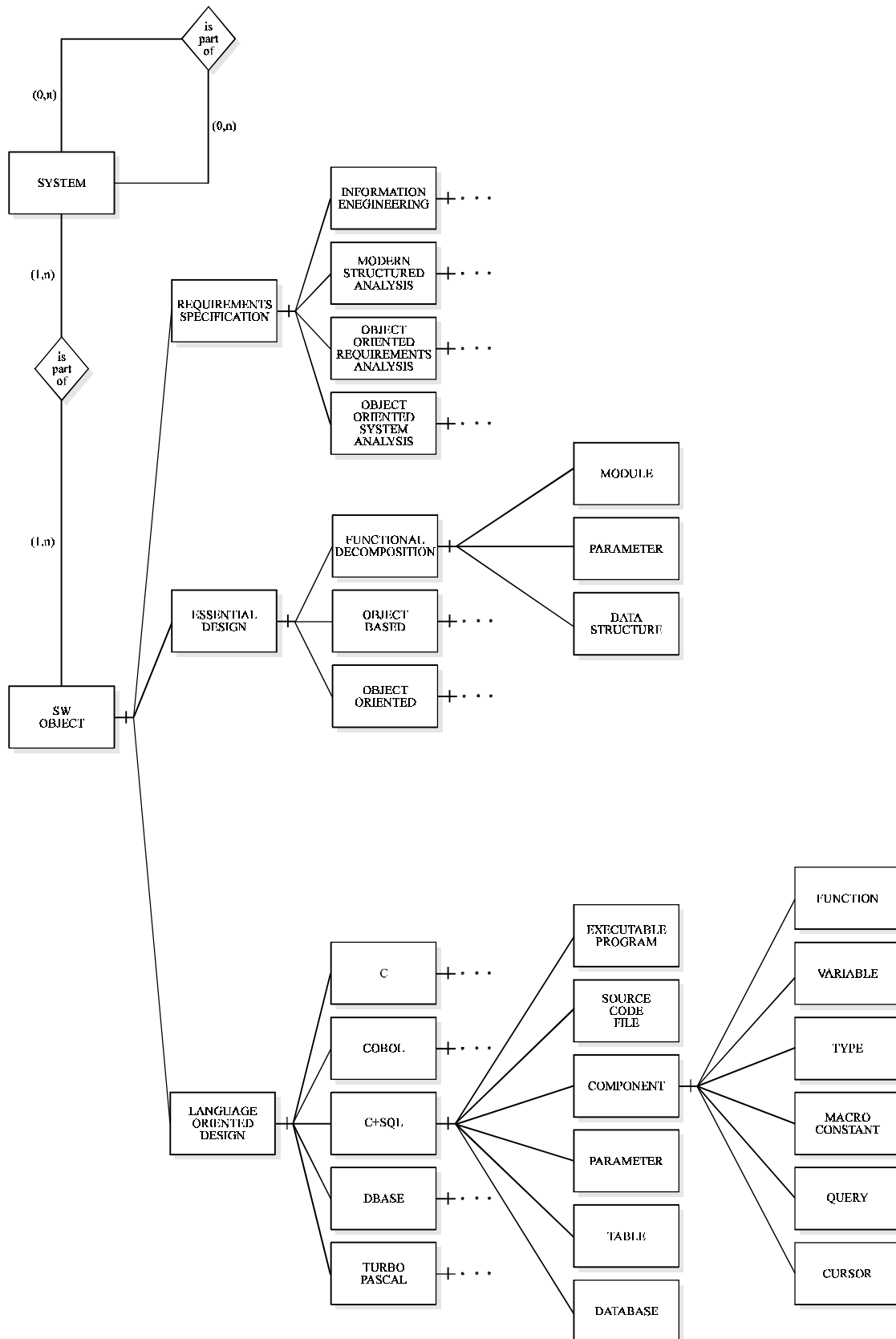
Figure 1. Software object hierarchy

To make the model independent of the methodology in use, we provide for a variety of analysis methods, design paradigms and programming languages. They are viewed as a

lower level of the *is-a* hierarchy. A further specialisation leads to typed software objects which characterize the life-cycle artifacts. For example, an essential design via functional decomposition (EDFD) contain the entity types: *module*, *parameter* and *data structure*, while a language-oriented design in C with embedded SQL defines the entity types: *executable program*, *source code file*, *component*, *parameter*, *function*, *variable*, *type*, *macro constant*, *query*, *cursor*, *table*, and *database*.

A traceability relationship associates the objects with each other so that a maintainer can gain understanding of a program by tracing back from source code to design and from design to requirements. The traceability relationship can be internal or external. *Internal traceability* expresses the dependencies between objects from the same development phase, while *external traceability* refers to relationships among objects across the life-cycle.

Internal traceability is provided by relationship types which link entities. Thus, entity and relationship types form conceptual models which reflect the specific methodology or programming environment. For example, the ER diagram in Figure 2 shows the language-oriented design according to language C with embedded SQL. In order to deal with large programs, we store only objects which can be seen beyond the boundaries of the abstraction mechanism (control or data).

External traceability, on the other hand, is assured by *cognitive* links. The cognition is compounded by the decisions which have a role when mapping a more abstract object (specification) to a lower object (implementation). The decisions describe the problem-solving process, in a semi-structured way, by adding the knowledge of the alternative transformations analysed and the rationale regarding the choice. A dependency graph is used to specify the cognitive links between two life-cycle artifacts.
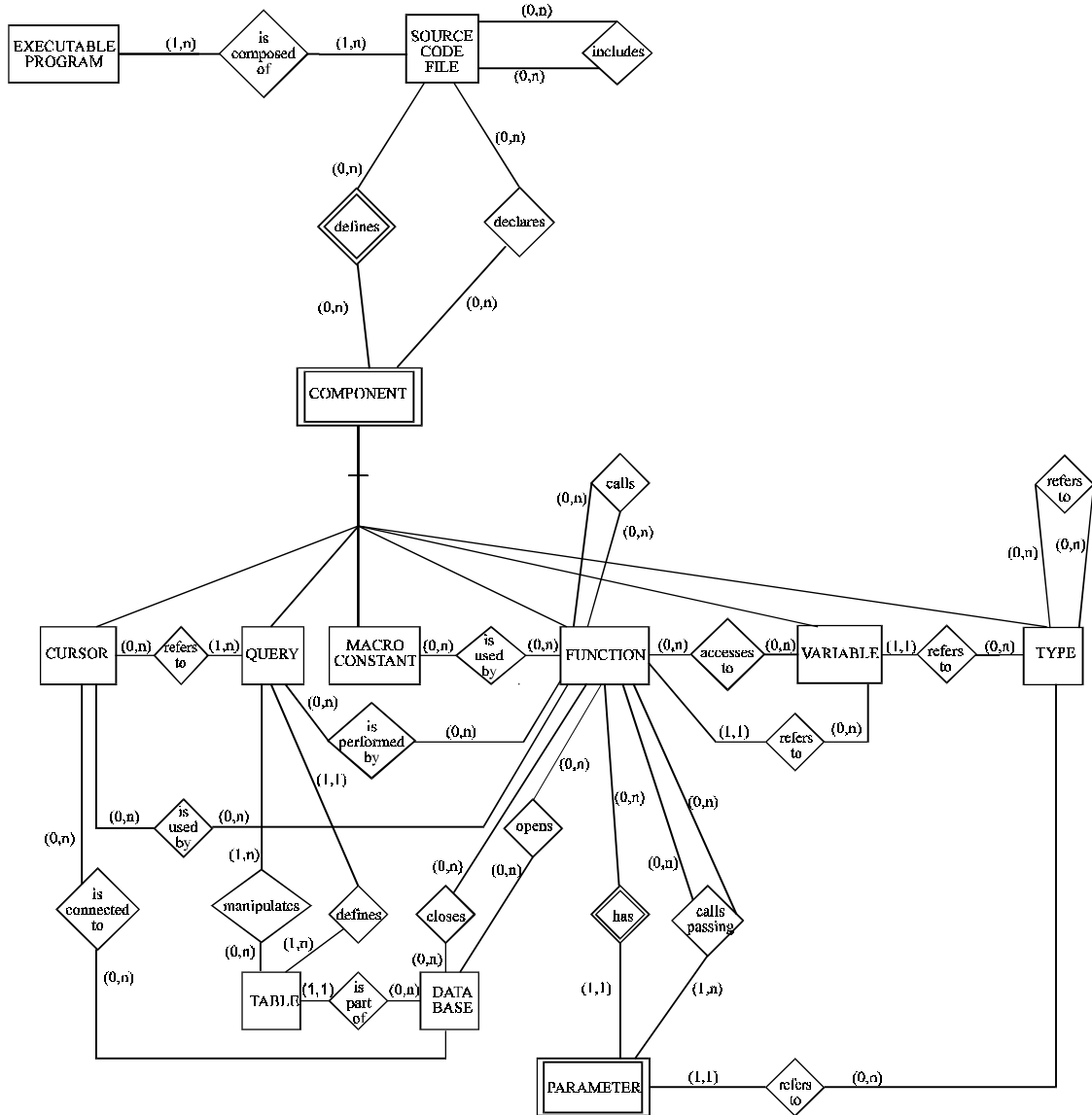
Figure 2. C+SQL ER schema

3. THE MODEL DEPENDENCY DESCRIPTOR

A *model dependency descriptor* is a directed acyclic graph $G = (N, E)$, where nodes are composed of three disjoint subsets:

$$N = S \cup T \cup D$$

S is the set of software objects in the source design model, T is the set of software objects in the target design model, and D represents the set of decisions which are involved in the transformation from the source model to the target model.

The graph can be applied at different levels of abstraction, each time a specification is transformed into an implementation through a problem solving process. For example, if

S refers to objects in a requirements specification and T to objects in an essential design, then D is the set of *design decisions*. In the same way, if S refers to objects in an essential design and T to objects in a language-oriented design, then D is the set of *implementation decisions*.

A decision *d* is described by a 6-tuple

$$d = (id, status, prob, Alt, sol, just,).$$

The first attribute, *id,* is an identification code which is different for each decision.

The *status* attribute states whether the decision is waiting, active, or old. The next section shows the decision life-cycle together with the events which drive the transitions. The attribute *prob* indicates a problem which requires some choice. It is described by a couple (desc, type), where *desc* captures the semantics of the problem in natural language and *type* classifies the problem according to a predefined taxonomy which derives from the characterization of design decisions in (Rugaber *et al.*, 1990). The taxonomy could be extended by the user to include new categories of problems arising from experience. *Alt* is the set of alternatives available for making the decision. Each alternative must be a viable solution for the related decision. An alternative *alt* is made up of (id, desc), where *id* is a relative identification of the alternatives for the same decision, while *desc* provides an informal description of the semantics of the potential solution. The attribute *sol* shows the alternative selected as solution. In other terms *sol* $\in$ *Alt*. The attribute *just* explains the rationale for the decision. Justification is made up of the couple (desc, type), where *desc* captures the semantics using natural language and *type* catalogues the justification according to a set of quality factors (maintainability, reliability, usability, etc.), design constraints (existing systems, lack of resources, company policy, legal restrictions, etc.) or previous decisions. In this way, the decisions can be selected automatically without reading the informal knowledge contained in the description field. This taxonomy must also be adaptable according to the user's experience.

Figure 3 shows the content of an implementation decision.

```
id:         d
status      active
prob
    desc:   "Implement the abstract data type Queue"
    type:   Representation
Alt
    id:   a1
    desc:   "array representation of lists"
    id:   a2
    desc:   "Circular array"
    id:   a3
    desc:   "Dynamic pointers"
sol
    id:   a2
    desc:   "Circular array"
just
    desc:   "In the array representation of lists, with a pointer to the last element, the Dequeue function
            has a linear complexity, because the entire queue must be shifted of one position in the
            array. On the contrary, both in the dynamic pointer and in the circular array
            implementation, the Dequeue function has a constant complexity.
            There is no need to manage a great number of items; so, the static representation is
            preferred because more easy to read."
    type:   Maintainability
```

Figure 3. An implementation decision

The edges *E* are divided in seven subsets:

$$E = Der \cup Inp \cup Out \cup Gen \cup Cause \cup Just \cup Succ$$

The subset *Der* represents *derivation links* between source and target software objects:

$$Der = \{ (s,t) \mid s \in S \wedge t \in T \wedge \textit{"t is derived from s"} \}$$

The derivation link is used only for describing transformations where there are no decisions to be stored because no alternatives have been considered. It is typical of life-cycle repositories where only the direct dependencies between artifacts are recorded. For example, *module* objects of the EDFD model are usually mapped directly to *function* objects in C language because C provides for a single mechanism for control abstraction. In Cobol, on the other hand, a selection must be made from Paragraph, Section and External Module because they are valid alternatives for implementing a functional component.

The subset *Inp* represents *input links* between source software objects and decisions:

$$Inp = \{ (s,d) \mid s \in S \wedge d \in D \wedge \textit{"s is input to d"} \}$$

The subset *Out* represents *output links* between decisions and target software objects:

$$Out = \{ (d,t) \mid d \in D \wedge t \in T \wedge \textit{"t is in output from d"} \}$$

For example, the decision in Figure 3, which selects a circular array to implement a queue, has an input link with the EDFD object "queue" (type Data Structure) and an output link with the Pascal object "Queue_Int" (type Variable).

The subset *Gen* represents *generalization links* between decisions:

$$\text{Gen} = \{ (d_g, d_i) \mid d_g \in D \land d_i \in D \land \textit{"}d_g \textit{ generalizes } d_i\textit{"} \}$$

The decision d_g is a generic kind of decision, in the sense that the problem, the alternatives and the solution are specified but without linking software objects. The decision d_i is defined by instantiation from d_g, because it inherits the attributes of the generic decision and adds the specific links with the software objects. For example, the generic decision to use only circular arrays for all the queues is connected by generalization links to specific decisions, one for each type of queue elements. The decisions instantiated from the generic decision link the EDFD object "queue" with the Pascal objects: "Queue_Int", "Queue_Char", "Queue_Rec1", etc.

The subset *Cause* represents cause links between decisions:

$$\text{Cause} = \{ (d_i, d_j) \mid d_i \in D \land d_j \in D \land \textit{"}d_i \textit{ causes } d_j\textit{"} \}$$

The solution chosen for d_i generates a problem which must be addressed by d_j. As a consequence of the decision in the previous example, a new problem arises regarding the passage of the input parameter "queue" (as value or reference) into the parameter list of the procedures. The two decisions are connected by a cause link because the existence of the second decision is tied to the choice made in the first decision. Deletion of the first decision or changes in the solution will cause the disappearance of the second decision.

The subset *Just* represents justification links between decisions:

$$\textit{Just} = \{ (d_i, d_j) \mid d_i \in D \land d_j \in D \land \textit{"}d_i \textit{ justifies } d_j\textit{"} \}$$

The description of the justification for d_j is not in textual form but is provided by the existence of d_i. For example, suppose that we have chosen an old user interface management system (UIMS) because it is already available. This decision justifies the choice of an interaction style for a menu area, consisting of a fixed menu with keyboard selection, although the solution is not user-friendly like the other alternatives analysed

(PopupMenu, PulldownMenus in a MenuBar, OptionMenu, all with mouse selection)
The UIMS decision is connected to the menu-type decision by a justification link. If the UIMS changes, the menu-type decision can continue to make sense, although its justification should be reviewed.

The subset *Succ* represents *successor links* between decisions:

$$\text{Succ} = \{\ (d_s, d_p)\ |\ d_s \in D \wedge d_p \in D \wedge \textit{"}d_s \textit{ is successor of } d_p\textit{"}\ \}$$

The successor link provide a record of decision history during software evolution. The decision d_s (successor) is a new version of the decision d_p (predecessor). Both the successor and the predecessor share the problem but differ as regards the solution. Alternatives may remain unchanged or be refined with the addition of new options. For example, if a programmer decides to implement the abstract data type queue with dynamic pointers replacing the circular array, the new decision should be linked to the old decision by a successor link.

4. THE TRACEABILITY SUPPORT SYSTEM

The model dependency descriptor has been implemented in the Traceability Support System (TSS). The TSS is a tool for capturing rationales. The captured decisions enable cognitive links to be established between the artifacts produced during the development process and stored in the design database.

The TSS is part of an integrated environment for software evolution management (IESEM). As shown in Figure 4, the IESEM consists of a design database, and a set of four major components: Translator, Extractor, the System Quality Monitor and the TSS itself. The Translator maps information between the repository of commercial CASE tools used for forward engineering and the design database. The Extractor populates a language-oriented schema starting from source code. The System Quality Monitor analyses intermodular and intramodular measures to evaluate the quality of the software objects. The TSS deals with both external traceability and internal traceability inside the design database. More details on the first three tools can be found in (Lanubile and

Visaggio, 1992a; Lanubile and Visaggio, 1992b; Cimitile *et al.*, 1992). In this section we exploit the architecture and functionalities of the TSS.

The TSS has a layered architecture consisting of:

     (1) the TSS core,

     (2) the services layer, and

     (3) the interaction layer.

The TSS core provides database schemes for recording decisions and linking them to the software objects. The services layer provides operators for manipulating decisions and links, and facilities for tracing objects inside the model dependency descriptor. The functions offered by the services layer are independent of the database schema and can be used by any application outside the TSS. The interaction layer provides a user interface to help the user complete tasks quickly and easily thanks to mouse-based navigation and windowing capabilities
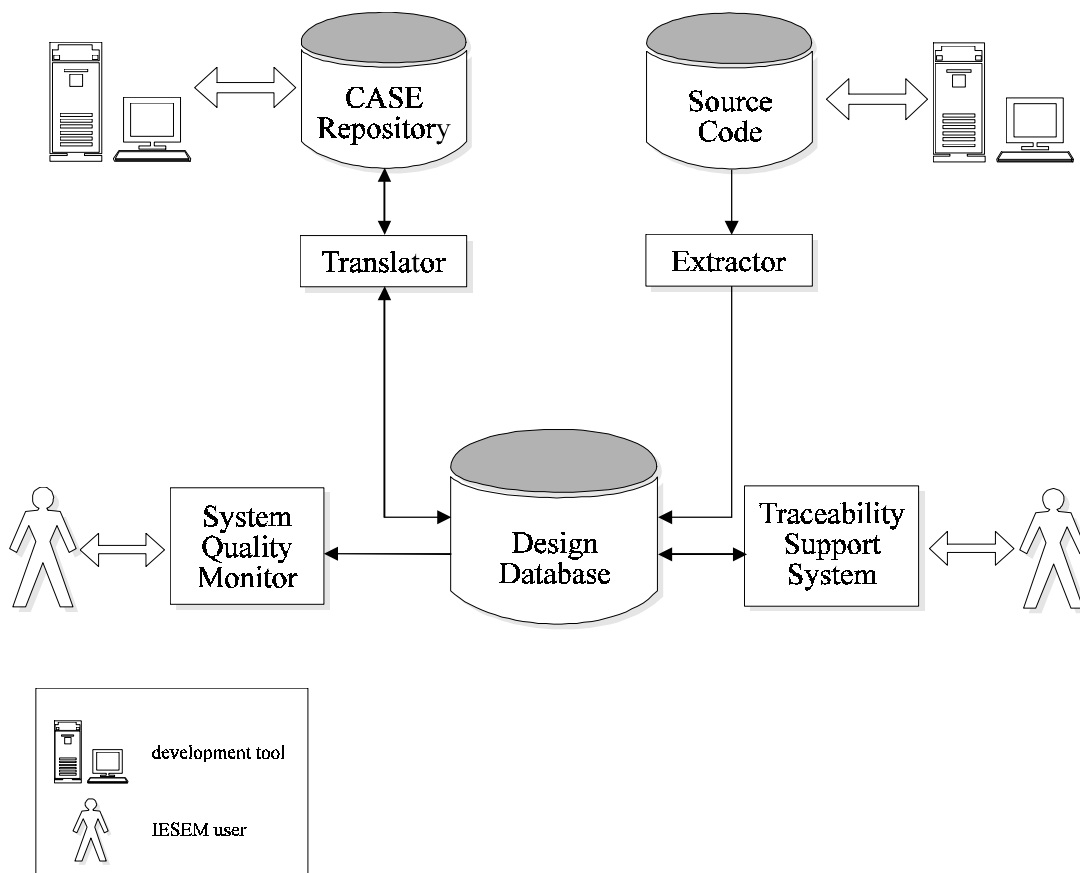

Figure 4. IESEM architecture

.

A hybrid prototyping/waterfall paradigm has been adopted for the development of the TSS. First, a prototype was built to clarify the characteristics and operation of the TSS by learning from a rapid application on the field. The prototype was built on an IBM-compatible personal computer, using SQLWindows, a graphical database application development tool which makes it possible to accelerate prototype development. After having collected experience in laboratory during the maintenance of middle size systems (from 2 to 10 KLOC), the TSS has been subjected to conventional development as a Motif application running on UNIX workstations.

4.1 The TSS core

The TSS core makes it possible to record the decisions and the links between decisions and software objects. First, the graph representation of the model dependency descriptor is converted into an ER diagram, as shown in Figure 5. The diagram is then implemented in an object-oriented database which is directly supported by the HP Iris DBMS. The definition and manipulation language is an object-oriented extension of SQL, the Object-SQL, which supports *is-a* hierarchies and recursive queries.

4.2 The services layer

The services layer provides the basic operations for building a model dependency descriptor and navigating along its paths. Table 1 summarizes the main functions for operating on decisions and links. The functions which are above the double line modify the state of a decision, while the functions below preserve the state. For the sake of brevity, in the following we only provide an informal description of these functions, as the services layer has been formally specified in (Visaggio *et al.*, 1993), using Z notation.

Figure 6 shows the state transition diagram which models the life cycle of a decision. The states that a single decision may occupy are *waiting*, *active*, and *old*. A service call may activate a transition between the states.
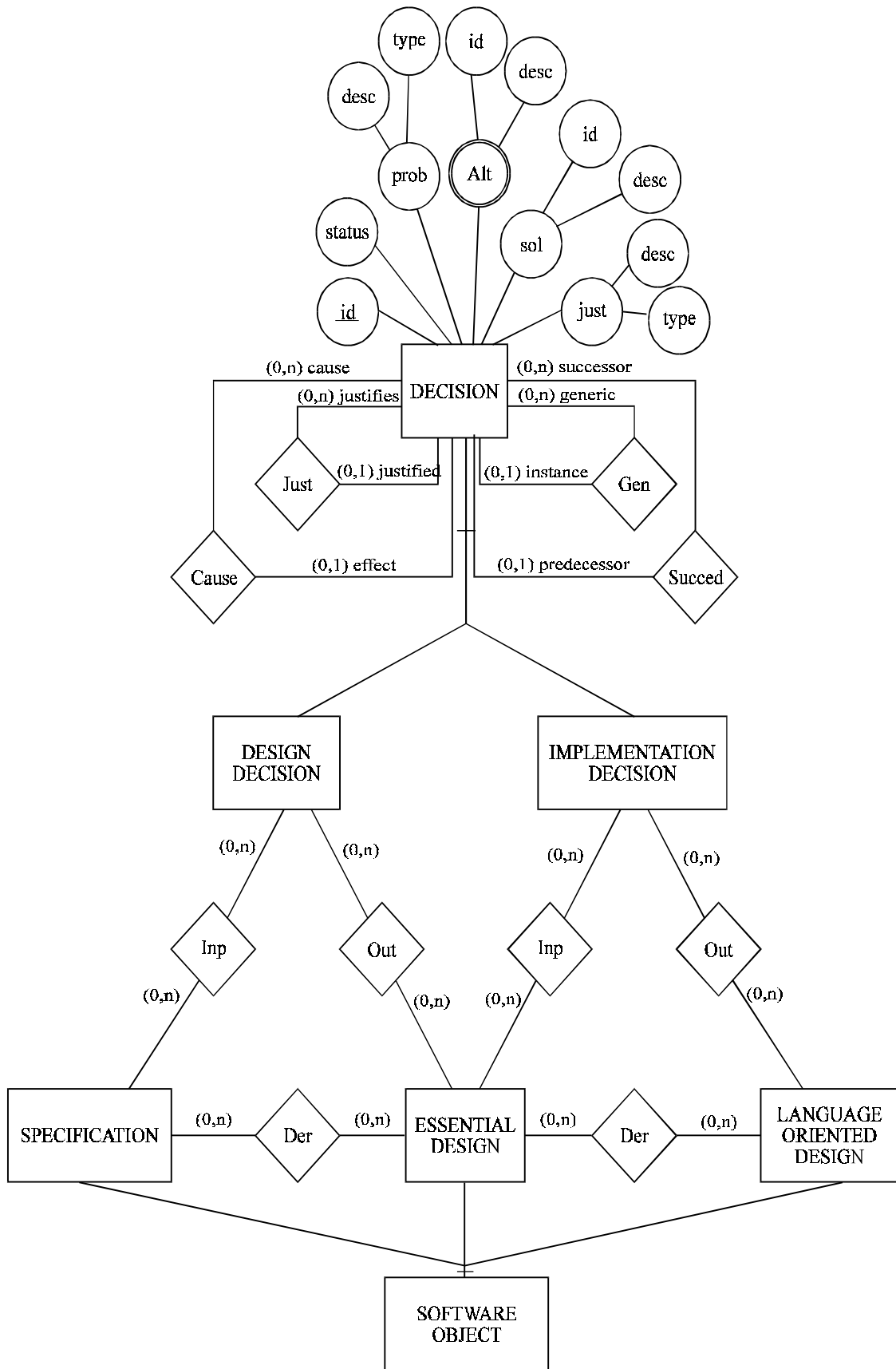
14

Figure 5. Model dependency descriptor ER schema

Table 1. The TSS services

Name	Description
Create	Originate a new decision
Instance	Instantiate a parameterized decision from a generic decision
Replace	Originate a new decision which takes the place of another decision
Connect	Make a decision operative
Disconnect	Make a decision ineffective
Delete	Remove a decision
Update	Change a decision
AddInpLink	Set an input link between a source node and a decision
AddOutLink	Set an output link between a decision and a target node
AddCauseLink	Set a cause link between two decisions
AddJustLink	Set a justification link between two decisions
AddDerLink	Set a derivation link between a source and a target node
RemoveLink	Eliminate a link between two nodes
WhichLink	Return the kind of link between two nodes
Next	Retrieve the heads of links from a node
Previous	Retrieve the tails of links from a node
History	Retrieve the predecessors of a node
TraceForward	Trace paths from a node, going forward
TraceBackward	Trace paths from a node, going backward

Initially a decision must be regarded as non-existent. Three events may occur. The first event takes place when a new decision has been edited and stored (*Create*). The second event, when a decision is instantiated from a generic decision (*Instance*). The third occurs when a new decision follows another *active* decision with the same problem and takes its place (*Replace*). As a consequence of the operation, the previous decision is no longer *active* but becomes *old*. When a decision is replaced, all the decisions dependent on it through Cause or Just links pass to *old* status too. However, new justified decisions are created through replication and given *waiting* status, because the justified decisions continue to make sense although the rationale should be reviewed. In this way, we preserve the software history during its evolution.

All the services bring a new decision to the state of *waiting*, where the decision exists but is isolated. In this state, a user may remove a decision from the design database by

invoking the Delete operator, or modify the content of a decision by calling the Update operator, or link the decision to another node by using a linking function (AddInpLink, AddOutLink, AddCauseLink, AddJustLink), or eliminate an undesired link by employing the RemoveLink function.
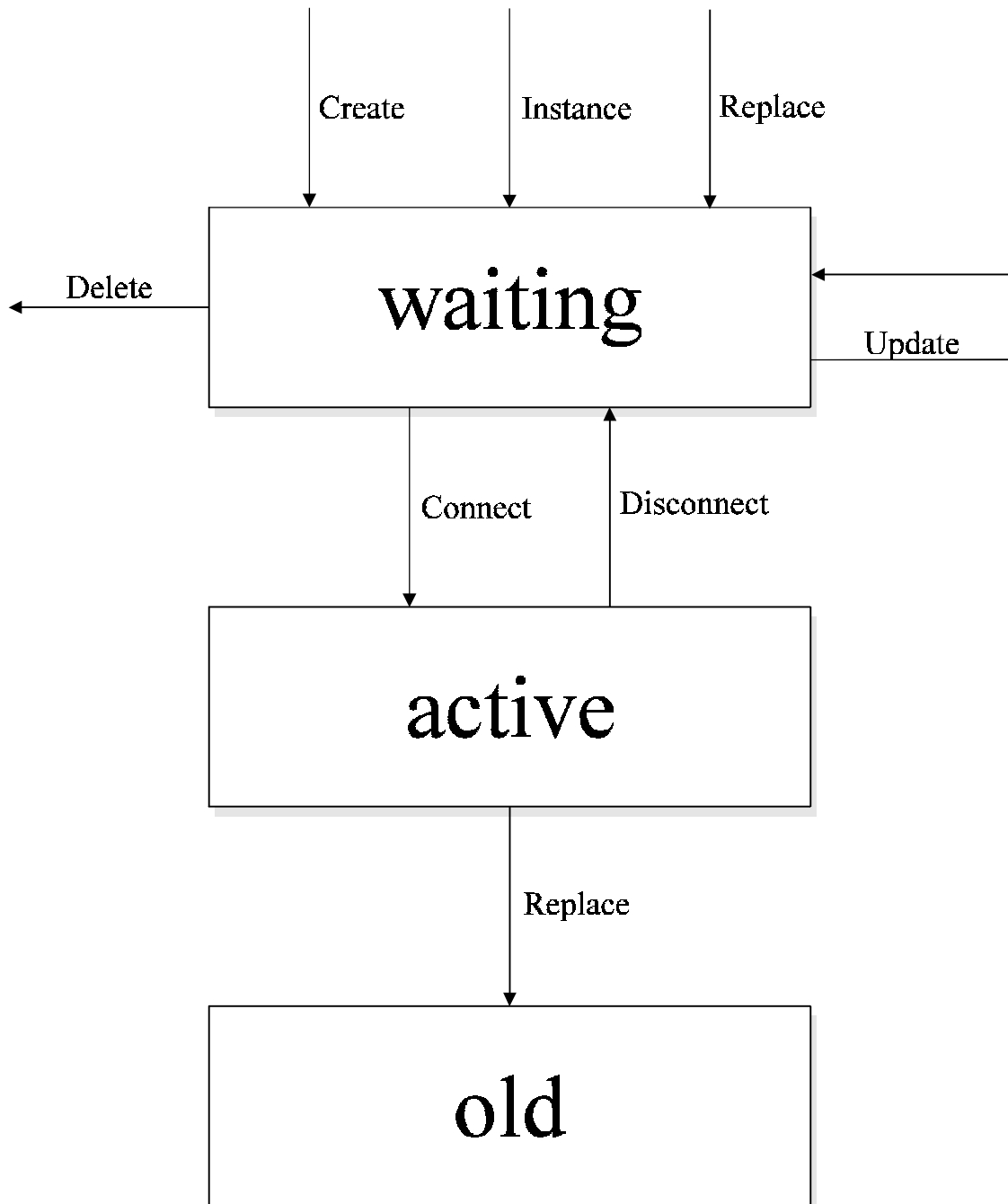

Figure 6. Decision lifecycle

When all the links have been specified, the decision is ready to become a cognitive link for the design database by means of the Connect operation. The Connect service carries the decision to *active* status. On the contrary, the Disconnect operator carries the decision back from *active* to *waiting* status. While *waiting* status characterizes incomplete specifications during constructive design, active status represent a stable situation where the decision is baselined and used for supporting the maintenance process. For this reason, the navigation services (Next, Previous, History, TraceForward, TraceBackward) ignore the *waiting* decisions.

The remaining services are used to examine the model dependency descriptor. The WhichLink service verifies the existence of a link between two nodes before adding or removing operations. The Next and Previous services return to the immediate adjacent nodes respectively in the digraph G and in the inverse G^{-1}. They allow a user to navigate step by step in the model dependency descriptor to verify its content. The History service runs on the transitive closure of the successor link and elicits all the historical ancestors of a decision. The TraceForward and TraceBackward services work like the Next and Previous services but in the graph resulting from the transitive closure of the model dependency descriptor. The TraceForward and TraceBackward functions are used in impact analysis to identify the changing artifacts, and in program comprehension to capture the rationales along the traceability paths. The operations have been implemented as C functions and use the IRIS C language interface for accessing the design database.

4.3 The interaction layer

The interaction layer provides a graphical user interface for editing, selecting and manipulating decisions and links in the model dependency descriptor. The mouse-driven interface presents information and allows data to be entered using menus (pull-down, pop-up, option), dialog and list boxes, scroll bars, buttons (push, cascade, option, radio) and other graphic items.

Figure 7 shows the initial screen when the user enters in the TSS. The user must select the abstraction level: design or implementation decisions. If design decisions are chosen

then the tool will refer to the specification objects as source nodes and to the essential design objects as target nodes. If the implementation decisions have been selected then the essential design objects are considered as source nodes and the language-oriented objects as target nodes. In the Figure, the user has selected implementation decisions between essential design via functional decomposition and language-oriented design in C+SQL.
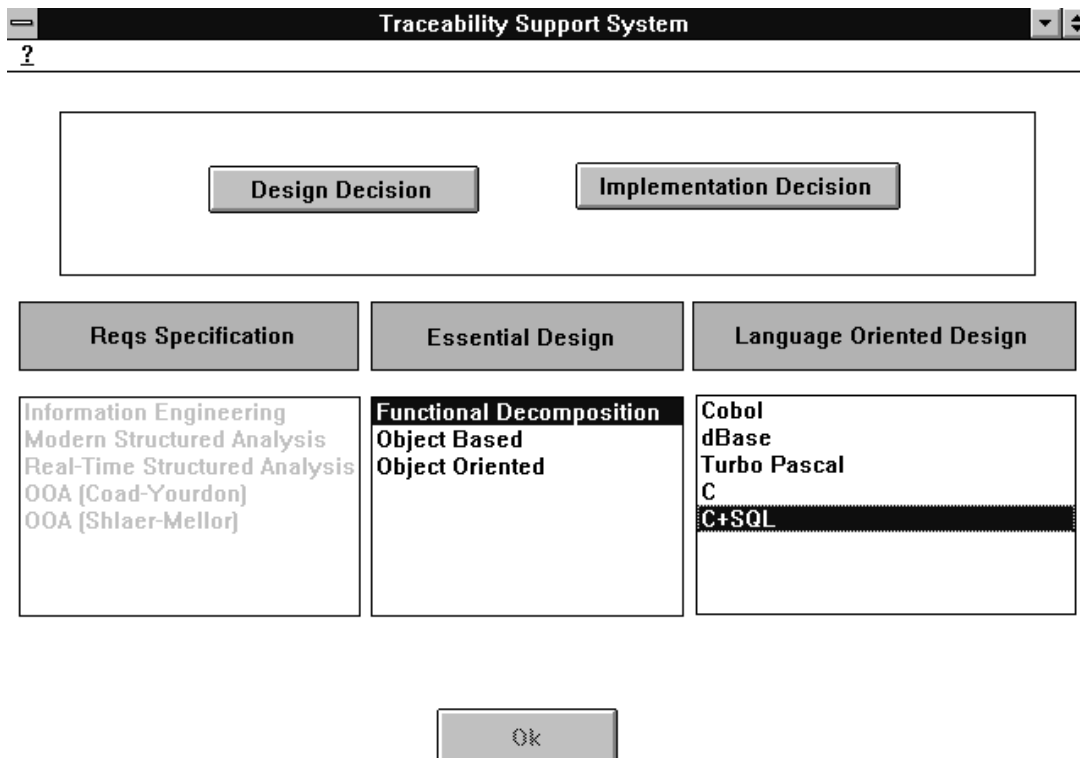
Figure 7. The initial TSS screen

Figure 8 shows the screen which is typically used during constructive design for working with decisions. The data fields reflect the structure of a decision. The screen is not limited to editing, as the user can invoke the operations of the services layer through the push buttons.
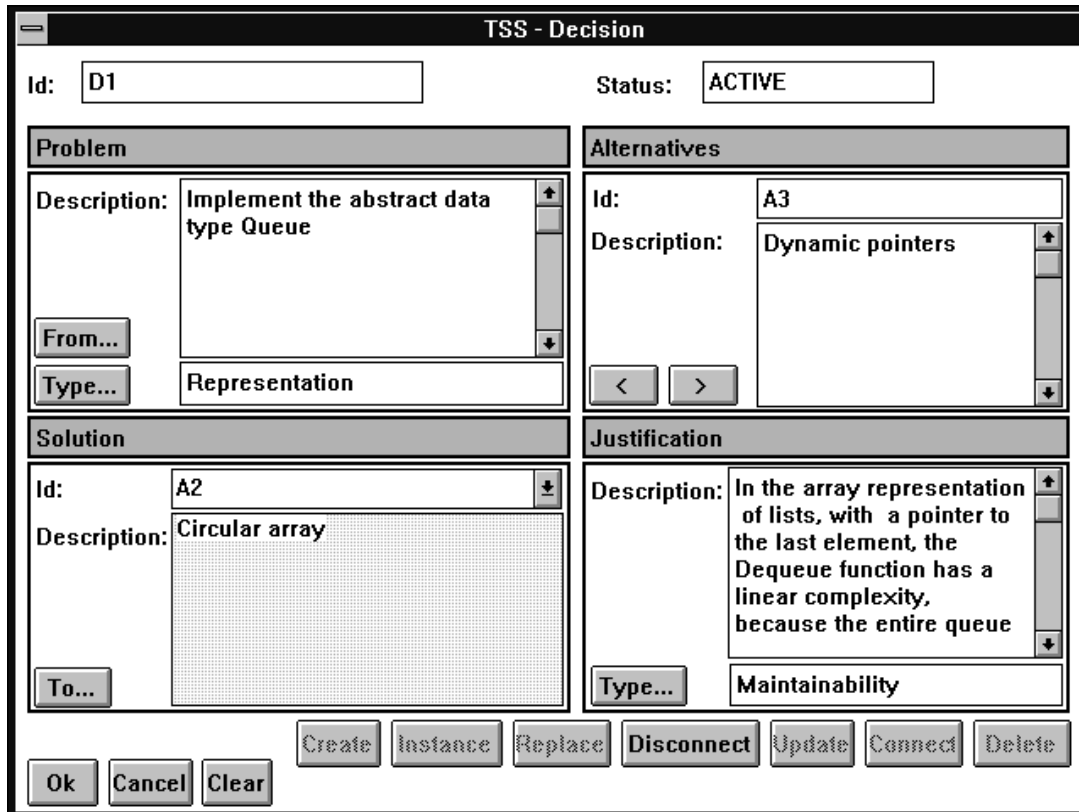
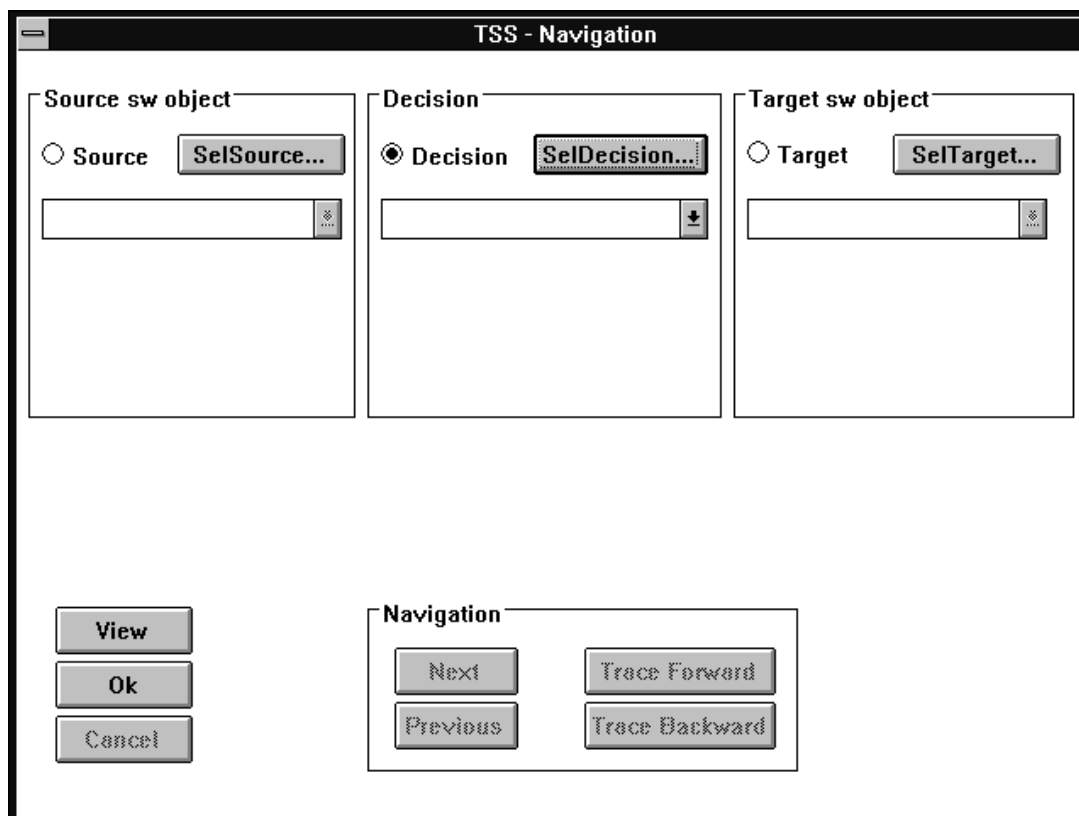*Figure 8. The TSS screen to work with decisions*

Figure 9. The TSS screen to navigate into the model dependency descriptor

Figure 9 shows the screen for navigating in the model dependency descriptor. The SelSource and SelTarget buttons retrieve source and target software objects respectively, while the SelDecision button retrieves decisions through three different mechanisms: identification, keyword, and pattern matching. The keyword option searches for decisions using the structured part of a decision (status, prob type, just type), while the pattern matching option looks for strings inside the descriptive fields (prob desc, alt desc, sol desc, just desc).

The interaction layer uses both the storage layer, to select objects and decisions in the design database, and the services layers which are invoked by push buttons.

5. HOW TO USE THE TSS - AN EXAMPLE

A scenario is used to show a dynamic description of the role of the TSS in a maintenance task. The software to be maintained is the reverse engineering tool Extractor. We have focused our attention on the subsystem which analyses TurboPascal programs. The subsystem is designed by functional decomposition, written in C and uses embedded SQL to interface a relational DBMS. There are 98 C functions, 15 SQL tables are used, and the size is 4638 LOC.

The design database was constructed incrementally. First the specification schema was imported using Translator. Then the first decisions on design issues were created and linked to the specification objects using TSS. The Translator was then used again, this time for importing the essential design schema. The missing links were added to the model dependency descriptor, using the linking services of TSS, and at this stage the design decisions became active. The process was then repeated, this time for the language-oriented design. Primary implementation decisions were edited but left hanging. After the program was codified, its language-oriented schema was imported using the Extractor for C programs with embedded SQL. Then, using the TSS again, the remaining links and decisions were added and the implementation decisions connected.

A real problem which arose when executing the Extractor with Pascal programs of 4000 LOC, was the duration time of the last phase (14 hours) with respect to the first four phases (45 minutes). The last phase uses the structural information already stored in the design database to compute information flow metrics and updates the design database with the results. The performance degradation affected usability, because the computational feature was avoided when selecting the starting options of Extractor. A modification was required with the object of improving the performance of the structure metrics computation and in particular the information flow metrics.

Starting from the specification objects (SelSource) dealing with the information flow metrics, we applied the TraceForward primitive of the services layer to retrieve all the related design decisions and the essential design objects affected. The result was a decision which separates the acquisition of input data from the design database into two distinct parts. The former extracts the information flows according to Henry and Kafura's definition (1981) and the latter, in accordance with the variant proposed by Ince and Shepperd (1989). Because the bottleneck could be in the database access, we proceeded to investigate the implementation decisions. A query was built (SelSource) to retrieve all those modules, inside the scope of control of the selected objects, which used some global data structure. The TraceForward was applied again, this time starting from the retrieved modules, to collect all the implementation decisions and the language-oriented objects.

Figure 10 shows a fragment of the result in graphical form. The decision *d0*, shown in Figure 11, was to use a nested SQL query for retrieving the indirect flows of a given module. The class of the justification was maintainability because the query could strictly match with that in the detailed module specification. Decision *d0* is instantiated by a number of implementation decisions, one for each application of the generic decision. In particular, decision *d01* is connected in input to the module "GetIndFlowSrc", and in output to the function "GetIndFlowSrc" and to the query "Indirect".
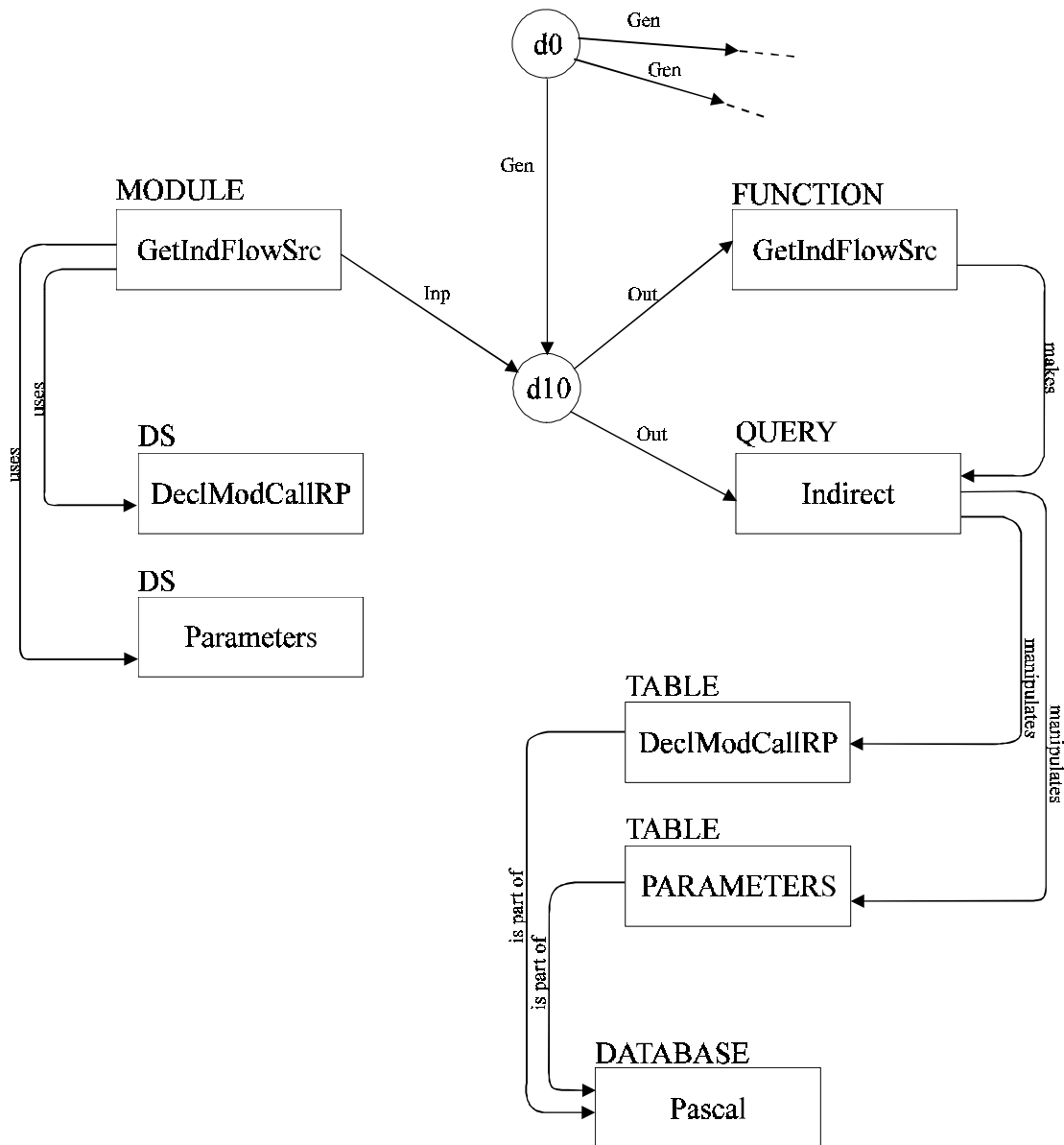
Figure 10. Model dependency descriptor before the modification

id: d0
status   active
prob
   desc: "Implement a nested SQL query"
   type: Composition/Decomposition
Alt
   id: a1
   desc: "Use a nested SQL query"
   id: a2
   desc: "Use two distinct SQL queries. For each tuple from the first query, a procedure iteratively
               activates the second query. The duplicated tuples are purged at the end of the iteration."
sol
   id: a1
   desc: "Use a nested SQL query"
just
   desc: "The resulting code strictly reflects the detailed module specification."
   type: Maintainability

Figure 11. The decision d0

23

id:	d1
status	active
prob	
desc:	"Implement a nested SQL query"
type:	Composition/Decomposition
Alt	
id:	a1
desc:	"Use a nested SQL query"
id:	a2
desc:	"Use two distinct SQL queries. For each tuple from the first query, a procedure iteratively activates the second query. The duplicated tuples are purged at the end of the iteration."
sol	
id:	a2
desc:	"Use two distinct SQL queries. For each tuple from the first query, a procedure iteratively activates the second query. The duplicated tuples are purged at the end of the iteration."
just	
desc:	"Operation revealed a response time ratio (nested/distinct) of 20:1."
type:	Efficiency
id:	d2
status	active
prob	
desc:	"Implement the interface between the two queries obtained from the decomposition of a nested SQL query"
type:	Representation
Alt	
id:	a1
desc:	"Use an SQL cursor"
id:	a2
desc:	"Use a pointer implementation of queue"
id:	a3
desc:	"Use a circular array implementation of queue"
sol	
id:	a1
desc:	"Use an SQL cursor"
just	
desc:	"The SQL cursor is made available to programmers by the DBMS, so it comes already tested."
type:	Reliability
id:	d3
status	active
prob	
desc:	"Implement the purge of duplicated tuples, resulting from the decomposition of a nested query"
type:	Representation
Alt	
id:	a1
desc:	"Use an SQL cursor defined on a temporary table"
id:	a2
desc:	"Use a pointer implementation of queue"
id:	a3
desc:	"Use a circular array implementation of queue"
sol	
id:	a1
desc:	"Use an SQL cursor defined on a temporary table"
just	
desc:	d2
type:	Decision

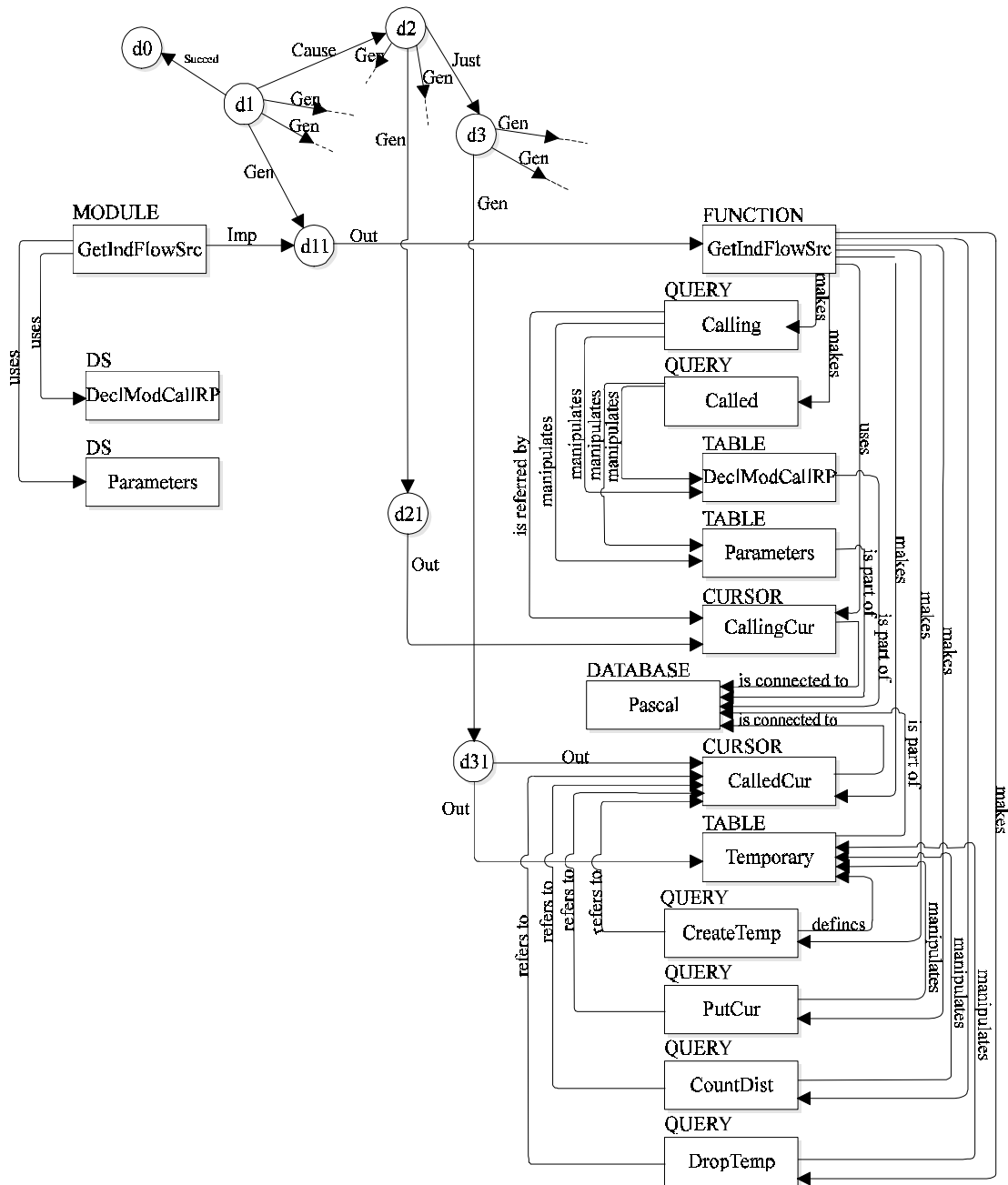*Figure 12. The decisions d1, d2 and d3*

Figure 13. Model dependency descriptor after the modification

We discovered, by running the nested query interactively, that this decision was very time-consuming. Because the decision was tied to the other ones by a generalization link, its effect on program performance was onerous. We therefore decided to select the second alternative as solution, consisting of breaking the query down into two parts and using more procedural code to process the results. A new decision, shown in Figure 12, was taken for solving the problem of how to implement the interface between the two resulting queries. This modification brought the response time to 16 minutes, compared with 220 minutes before the change. The gain in efficiency, stored as justification type

of the new decision, was obtained at the expense of maintainability, because the resulting code was more complex due to the presence of the SQL cursors and control structures, which had been absent before. Figure 13, showing a portion of the new model dependency descriptor, bears evidence to the increasing complexity; there are more nodes, edges and paths which must now be tracked to manage the new version of the program. There are three generic decisions; *d1* and *d2* are connected by a cause link and *d2* and *d3* by a justification link. The decision d1 is the successor of the old decision d0. Each generic decision has instances connected by a generalization link. In particular, *d11* has the module "GetIndFlowSrc" in input and the function "GetIndFlowSrc" in output. The instance d21 has the SQL cursor "CallingCur" linked in output, and the instance *d31* the cursor "CalledCur" and the table Temporary.

6. AN EMPIRICAL STUDY

A preliminary investigation has been conducted to assess our approach to design recording. The Goal/Question/Metric (GQM) paradigm (Basili and Weiss, 1984; Basili and Rombach, 1988) was been invoked to set the goal and refine it into quantifiable questions that specify metrics. The data collected are validated and analysed in the context of the structure resulting from the GQM paradigm. The goals are specified in terms of purpose, perspective and environment. In our study, the following goal was set:

Purpose: to analyze software products for the purpose of characterization,

Perspective: with respect to the model dependency descriptor from the maintainer's point of view.

Environment:

The context is a replicated project study in an academic software engineering course, where thirty student teams (3-5 people) each developed a business application program starting from the same software requirements specification. Students followed a phased development process made up of design, implementation and testing. Design was conducted according to functional decomposition. Implementation had to be done on

IBM-compatible personal computers with MS-DOS operating system. No constraints were set for the programming language. The resulting programs ranged from 3 to 10 KLOC. Students were asked to render the problem-solving process explicit in the form of design and implementation decisions and to link decisions to software objects.

The goal has been broken down into three questions:

Question 1: What are the decisions?

Question 2: How are decisions structured?

Question 3: How are decisions linked?

Data were collected from the deliverables of the development work. Since software requirements were common to all the development teams, the resulting thirty software products can be seen as variants of the same software application. All questions have been answered by carefully reviewing the produced documentation. All differences reported here have a significance level of 0.05. The statistical test used to assess the significance of differences between design and implementation decisions was the nonparametric Mann-Whitney U-test. The data collected and results of the analysis are presented in terms of the questions related to the stated goal.

Tables 2, 3 and 4 answer the first question which characterizes decisions. In Table 2, design decisions are significantly more numerous than implementation decisions (76.79% versus 23.21%), confirming the central role of the design phase in the problem solving process. Most of the decisions are instantiated (72.96%) from generic decisions (7.11%). This show that decisions are usually based on general principles and then applied to specific cases.

Table 2. Decisions by type

Decision type	Design decisions		Implementation decisions		All	
	N	*%*	*N*	*%*	*N*	*%*
Generics	111	7.18	32	6.85	143	7.11
Instances	1125	72.82	343	73.45	1468	72.96
Others	309	20.00	92	19.70	401	19.93
All	1545	76.79	467	23.21	2012	100

Table 3. Decisions by problem type

Problem type	Design decisions		Implementation decisions		All	
	N	*%*	*N*	*%*	*N*	*%*
Encapsulation/Interleaving	121	29.02	18	14.52	139	26.69
Generalization/Specialization	133	31.89	43	34.68	176	32.53
Data/Procedure	16	3.84	7	5.65	23	4.25
Representation	71	17.03	39	31.45	110	20.33
Function/Relation	0	0.00	0	0.00	0	0.00
Composition/Decomposition	76	18.23	17	13.71	93	17.19

Table 4. Decisions by justification type

Problem type	Design decisions		Implementation decisions		All	
	N	*%*	*N*	*%*	*N*	*%*
Quality Factors	*406*	*97.13*	*107*	*86.29*	*513*	*94.65*
Correctness	51	12.20	5	4.03	56	10.33
Reliability	104	24.88	12	9.68	116	21.40
Efficiency	39	9.33	26	20.97	65	11.99
Integrity	10	2.39	6	4.84	16	2.95
Usability	25	5.98	24	19.35	49	9.04
Maintainability	143	34.21	25	20.16	168	31.00
Testability	0	0.00	2	1.61	2	0.37
Flexibility	12	2.87	4	3.23	16	2.95
Portability	1	0.24	0	0.00	1	0.18
Reusability	21	5.02	3	2.42	24	4.43
Interoperability	0	0.00	0	0.00	0	0.00
Others	0	0.00	0	0.00	0	0.00
Constraints	*1*	*0.24*	*5*	*4.03*	*6*	*1.11*
Existing systems	0	0.00	1	0.81	1	0.18
Lack of time	0	0.00	4	3.23	4	0.74
Lack of resources	0	0.00	0	0.00	0	0.00
Company policy	1	0.24	0	0.00	1	0.18
Legal restriction	0	0.00	0	0.00	0	0.00
Others	0	0.00	0	0.00	0	0.00
Decision	*11*	*2.63*	*12*	*9.68*	*23*	*4.24*

Decisions are also characterized according to the problem type. Table 3 shows that
e*ncapsulation/interleaving* decisions are frequent (25.69%) but significantly decrease
from the design to the implementation phase (29.02% versus 14.52%). One reason may
be that information hiding has been applied since the design phase so as to lower the
need for encapsulation in the implementation phase. The *representation* decisions, on

the other hand, significantly increase from the design to the implementation phase (17.03% versus 31.45%) owing to the effort of adapting the essential design to the programming environment. We can also observe that there are no *function/relation* decisions and few *data/procedure* decisions (4.25%), probably due to a deficiency in the problem analysis. The zero percentage in the *other* field shows how the classification of problems has been able to cover all the considered decisions.

Another characterization of decisions is the type of justification, shown in Table 4. Decision rationales are mostly quality-based (94.65%). The quality factors which are often considered in the design decisions are: maintainability, reliability and correctness (respectively 34.21%, 24.88%, 12.20%), while in the implementation decisions they are: efficiency, maintainability, and usability (respectively 20.97%, 20.16%, 19.35%). Maintainability is high in both phases because we teach students to develop software with change in mind. The increasing weight of efficiency is coherent with the indication of dealing with performance only as an optimisation of the final program. Usability increases in implementation because it is mainly applied in terms of user-friendliness of the human-interface. The taxonomy of justifications has been successfully applied, as the zero percentages in the *other* fields testify.

The decision structuredness raised by question 2 is characterized in this study by Table 5, which shows how many alternatives per decision have been taken into account. Most decisions (82.66%) have only two alternatives and there are very few decisions with more than three alternatives (2.40%). Because the decisions have not been affected by reworking due to maintenance activities, we expect that the number of alternatives will grow with exposure to operation in the real world.

Table 5. Alternative rate

	Number of alternatives			
	2	3	4	5
N	448	81	8	5
%	82.66	14.94	1.48	0.92

The Tables 6, 7 and 8 answer question 3, by describing the model dependency descriptor as they result from the case-studies.

Table 6. Links by type

Link type	Design decisions		Implementation decisions		All	
	N	%	N	%	N	%
Inp	994	21.75	455	7.96	1449	14.08
Out	1721	37.65	462	8.08	2183	21.21
Cause	5	0.11	4	0.07	9	0.09
Der	713	19.93	4447	84.48	5160	50.15
Cause	1125	24.61	343	6.00	1468	14.27
Just	13	0.28	8	0.14	21	0.20

Table 7. Link heads by type

Head type	Design decisions		Implementation decisions		All	
	N	%	N	%	N	%
Source obj	1707	93.08	4902	99.17	6609	97.52
Generic dec	5	0.27	3	0.06	8	0.12
Causing dec	111	6.05	32	0.65	143	2.11
Justifying dec	11	0.60	6	0.12	17	0.25

Table 8. Link tails by type

Tail type	Design decisions		Implementation decisions		All	
	N	%	N	%	N	%
Target obj	2434	68.08	4909	93.24	7343	83.07
Instance dec	4	0.11	4	0.08	8	0.09
Caused dec	1125	31.47	343	6.51	1468	16.61
Justified dec	12	0.34	9	0.17	21	0.24

Table 6 describes the distribution of links according to their type. There are few *cause* links (0.09%) and *justification* links (0.20%) with respect to the other ones. This means that above all the graph assigns a rationale to the transformation from source to target object or to the specific application of general principles. *Derivation* links prevail among the implementation decisions (84.48%) and this is coherent with the

considerations made for Table 2. There are no *succession* links because our programs have not yet been subject to change after operation.

Tables 7 and 8 describe the distributions of nodes in the model dependency descriptor according to the heads or tails of links respectively. Data show that decisions mainly involve source objects (97.52% in Table 7) and target objects (83.07% in Table 8). The wide coverage of software objects supports our confidence in the usefulness of cognitive links for impact analysis.

7. CONCLUSIONS

In this paper we have discussed an approach to design recording which uses the decisions taken during the software life cycle to assure traceability among all the deliverable items of a software system. The approach is technologically supported by a tool, the TSS, which is in its first prototype version. Since the prototype is not at a full functional level, we have not yet made experiments with industrial applications. However, empirical work has been done with medium programs developed by university student teams.

The results of this first experimentation reveal that in a model dependency descriptor it is possible to recognize the project goals and techniques for achieving these goals and to verify their coherence with the focus of the software engineering course attended by students. This means that the intentions of the original developer are recognisable in the decision web in the same way as the present behavior of developers and maintainers can be detected. This supports our confidence in the usefulness of this approach for the maintenance of large applications.

The experimentation has also shown that the taxonomies for problems and justifications are well grounded and we think the classification is now sufficiently mature to be applied in the industrial field without excessive modifications.

Future work will consist of experimenting the effects of the TSS on software maintenance tasks. We intend to work both on small-scale experiment programs with university students, in order to work in a controlled environment, and on large-scale

experiences with industrial projects, in order to gain confirmation of our thesis in the real world. A necessary condition for working with industrial programs is to evolve the TSS prototype so that information recording will not constitute a burden for maintainers.

Acknowledgements

References

Arango, G., Bruneau, L., Cloarec, J. and Feroldi, A. (1991) 'A tool shell for tracking design decisions', *IEEE Software*, March, pp.75-83.

Basili, V.R. and Weiss, D.M. (1984) "A methodology for collecting valid software engineering data", *IEEE Transactions on Software Engineering*, 10, (6), pp.728-738.

Basili, V.R. and Rombach, H.D. (1988) "The TAME project: towards improvement-oriented software environments", *IEEE Transactions on Software Engineering*, 14, (6), pp.758-773.

Brooks, F.P. (1987) 'No Silver Bullet: Essence and Accidents of Software Engineering', *Computer*, April, pp.10-19.

Canfora, G., Cimitile, A. and De Carlini, U. (1992) "A logic-based approach to reverse engineering tools production", *IEEE Transactions on Software Engineering*, 18, (12), pp.1053-1064.

Chen, Y., Nishimoto, M. and Ramamoorthy, C.V. (1990) "The C Information Abstraction System", *IEEE Transactions on Software Engineering*, 16, (3), pp.325-334.

Cimitile, A., Lanubile, F. and Visaggio G. (1992) 'Traceability based on design decisions', *Proceedings of Conference on Software Maintenance 1992*, Orlando, Florida, IEEE Computer Society Press, pp.309-317.

Desclaux, C. (1992) 'Capturing design and maintenance decisions with MACS', *Software Maintenance: Research and Practice*, 4, pp.215-231.

Garg, P.K. and Scacchi, W. (1990), "A hypertext system to manage software life-cycle documents", *IEEE Software*, May, pp.90-98.

Georges, M. (1992) 'MACS: maintenance assistance capability for software', *Software Maintenance: Research and Practice*, 4, pp.199-213.

Henry, S.M. and Kafura, D. (1981) 'Software Structure Metrics based on Information Flow', *IEEE Transactions on Software Engineering,* 7, (5), pp.510-518.

Ince, D.C. and Shepperd, M.J. (1989) 'An Empirical and Theoretical Analysis of an Information Flow-Based System Design Metric', *Proceedings of the 2nd European Software Engineering Conference*, Springer-Verlag, pp.86-99.

Klein, M. (1993) 'Capturing design rationale in concurrent engineering teams', *Computer*, January , pp.39-47.

Lanubile, F. and Visaggio, G. (1992a) 'Software maintenance by using quality levels', *Proceedings of Workshop on Software Quality: Measurement and Practice*, Capodichino (Naples), Italy, July, CUEN, pp.43-51.

Lanubile, F. and Visaggio, G. (1992b) 'Maintainability via structure models and software metrics', *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, June, IEEE Computer Society Press, pp.590-599.

Lanubile, F. and Visaggio, G. (1993) 'TSS specifications', *Internal Report*, available from authors, pp.1-22.

Lee, J. and Lai, K.Y. (1991) 'What's in design rationale?', *Human-Computer Interaction*, 6, (3-4), pp.251-280.

Lehman, M.M. (1989) 'Uncertainty in computer application and its control through the engineering of software', *Software Maintenance: Research and Practice*, 1, pp.3-27.

Lehman, M.M. and Belady, L.A. (1985) *Program Evolution - Processes of Software Change*, Academic Press.

Linos, P., Aubet, P., Dumas, L, Helleboid, Y, Lejeune, P. and Tulula, P. (1993) "CARE: an environment for understanding and re-engineering C programs", *Proceedings of Conference on Software Maintenance 1993*, Montreal, Quebec, Canada, IEEE Computer Society Press, pp.130-139.

Ramamoorthy, C.V., Usuda, Y., Prakash, A. and Tsai, W.T. (1990) "The Evolution Support Environment system", *IEEE Transactions on Software Engineering*, 16, (11), pp.1225-1234.

Rugaber, S., Ornburn, S. and LeBlanc, R.J. Jr. (1990) 'Recognizing design decisions in programs', *IEEE Software*, January, pp.46-54.

Simmonds, I. (1989) "Configuration management in the PACT software engineering environment", *ACM SIGSOFT, Software Engineering Notes*, 17, (7), pp.118-121.

Wild, C., Maly, K. and Liu, L. (1991) 'Decision-based software development', *Software Maintenance: Research and Practice*, 3, pp.17-43.