Submitted to Behaviour & Information Technology

Reuse of Domain Abstractions During Requirements Engineering: an Explanation is Required

N.A.M. Maiden & D.E. Tyndale

Centre for Human-Computer Interface Design
City University, London UK.

Tel: +44-71-477-8412
E-mail: N.A.M.Maiden@city.ac.uk

Abstract

Requirements engineering is the most complex and error-prone phase of software development. Reusing domain abstractions representing the fundamental features of all instances of a class of software engineering domain has been shown to aid requirement specification, however domain abstractions are difficult to understand. A controlled experiment investigated the effectiveness of domain abstraction reuse on the performance of inexperienced software engineers during a requirements engineering task. Reuse was aided by presentation of spatial diagrams and prototypical examples, to examine their effect on recognition, understanding and adaptation of domain abstractions. Results were surprising and revealed that effective domain abstraction reuse may not be straightforward. Implications are reported for design of tools which encourage reuse during requirements engineering. The reported study demonstrates benefits which arise from evaluating designs of software engineering environments before their implementation.

Short Title:
Reuse of Domain Abstractions During Requirements Engineering

1: Introduction

Requirements engineering is complex, error prone (e.g. Roman 1985, Meyer 1985) and in need of intelligent assistance for capturing complete and consistent requirements specifications (e.g. Meyer 1985). One solution is to populate tools with domain abstractions to assist requirements specification (Reubenstein & Waters 1991, Maiden & Sutcliffe 1992). Domain abstractions represent the fundamental behaviour, structure, goals and constraints of all instances of a class of software engineering domain which can be reused when modelling new domains. This paper reports an experimental investigation of domain abstraction reuse. Findings have implications for design of AIR (the Advisor for Intelligent Reuse), an intelligent tool assisting reuse of domain abstractions during requirements engineering as part of the ESPRIT 6353 'Nature' basic research action (Jarke et al. 1993).

Domain abstraction reuse can benefit requirements engineering (Reubenstein & Waters 1991, Maiden & Sutcliffe 1993a), however abstractions are difficult to understand. Reuse of generic specifications by inexperienced software engineers (Sutcliffe & Maiden 1990) revealed poor understanding and adaptation of abstractions manifest as copying, a result also reported in mathematical (Novick 1988) and physics (Chi et al. 1988) problem solving. Therefore the effort to understand abstractions when modelling new domains can outweigh benefits from successful reuse. To ensure effective reuse, explanation is needed to aid domain understanding. Explanation strategies implemented in AIR include guided exposure to prototypical examples and visualisation of domain abstractions (Maiden & Sutcliffe 1993a). Unfortunately the effectiveness of these strategies on understanding and adaptation remains undetermined, so this paper reports a controlled study on the effectiveness of domain visualisation and prototypical examples during reuse of domain abstractions.

The paper is divided into four parts. First, a theoretical model of domain knowledge is outlined to provide the background for the reported study. Second, strategies for explaining domain abstractions are outlined. Third, a controlled experimental study of domain abstraction reuse is reported. Fourth, conclusions and implications for design of tools to aid domain abstraction reuse during requirements engineering are discussed.

2: A Theoretical Model of Domain Knowledge

A theoretical model of domain knowledge has been proposed to determine the semantic content, boundaries and granularity of software engineering abstractions (Maiden & Sutcliffe 1993b). Experienced software engineers tend to recall and reuse mental knowledge structures or abstractions (Robertson & Strunch 1992) which permit them to perceive meaningful patterns in domains (e.g. Rich 1981, Soloway & Enrlich 1984). The importance of reusing

mental abstractions was suggested in studies (Guindon et al. 1987) which identified that a lack of specialised design schemata adversely affected design performance. Therefore reuse of domain abstractions mimics expert software engineering behaviour (e.g. Reubenstein & Waters 1991). Despite this however, most domain modelling research has focused on domain instances rather than domain abstractions.

2.1: Domain Abstraction Versus Domain Modelling

Domain-specific software engineering environments have been proposed for the specification, design (e.g. Fischer 1992) and maintenance (Arango et al. 1993a) of software systems. Most environments are application-specific, thus supporting design evolution (Arango et al. 1993b) and maintenance. On the other hand, requirements specification, by its nature, cannot always assume existence of complete and formal domain models. One solution is repeated reuse of generic domain knowledge in many applications. This contrasts with application-specific domain models which are both difficult and expensive to derive (Neighbours 1984, Arango et al. 1993a). Indeed, reusing generic domain knowledge can also guide the requirements engineering process towards key domain features as well as providing kernel models for defining viewpoints held by stakeholders (e.g. Leite & Freeman 1991, Easterbrook 1991). The cost of this approach, however, is the need for the requirements engineer to understand and customise retrieved domain abstractions to the new problem. Understanding abstractions is difficult (e.g. Sutcliffe & Maiden 1990). However, the Nature project claims that the potential benefits from explaining domain abstractions (Jarke et al. 1993) can outweigh the cost of modelling single domains (Arango et al. 1993a). It investigates how requirement engineers understand and adapt retrieved domain abstractions to inform design of tools for effective explanation and reuse. First, the theoretical model of domain knowledge (Maiden & Sutcliffe 1993b) is described to inform the reported study.

2.2: Domain Abstractions

The theoretical model hypothesizes that domain knowledge is organised in classes that share general features. Most software engineering domains can be ascribed to one of a tractably small set of domain classes which are distinguished by a few key features. This domain space is an inheritance hierarchy which specialises high-level domain abstractions to more detailed abstractions (Maiden & Sutcliffe, 1993b). A formal domain modelling language has been constructed to define all domain abstractions. In broad terms, domain abstractions are differentiated by:

- state transitions with respect to structure objects which change the state of objects;
- states representing simple set-theoretic constructs;

- positive or invariant goal states to be achieved or avoided;
- stative pre/postconditions on state transitions;
- domain or time events which trigger or stop state transitions.

This knowledge meta-schema is demonstrated using the domain abstraction shown in Figure 1(a). This object inspection abstraction can be instantiated to the dental patient domain shown in Figure 1(b). It is composed of one inspection structure and one waiting structure both containing many objects. Domain behaviour is defined by two state transitions of objects between the inspection and waiting structures, the first of which is triggered by a periodic temporal event. The dental patient domain consists of one checkup structure and one customer structure both containing many patients. Two state transitions represent the behaviour of patients as they attend and leave the checkup. Patient checkups are triggered by six-month periods between checkups.

The theoretical model has been operationalised as a computational mechanism for retrieval of domain abstractions (Maiden & Sutcliffe 1991, 1993c). The current knowledge meta-schema has been implemented in the knowledge representation language Telos (Mylopoulos et al. 1990) and ConceptBase (Jarke 1992), an object-oriented deductive data base (Jarke et al. 1993, Maiden & Sutcliffe 1993b). Over 150 domain abstractions have been modelled (Maiden & Sutcliffe 1993a) and are stored in this data base.

3: Strategies for Explaining Domain Abstractions

Explanation strategies are needed to aid understanding and adaptation of domain abstractions. AIR (Maiden & Sutcliffe 1992a) uses guided exposure to prototypical examples and spatial diagrams to discourage copying and promote understanding. Studies of these tool-based strategies (Maiden & Sutcliffe 1993a) revealed some improvement in the completeness of requirement specifications from greater use of explanation strategies. However, these associations were not strong, and requirements engineers were not asked to adapt retrieved abstractions to fit the new problem. Therefore a controlled study is needed to identify recognition, comprehension and adaptation difficulties which arise. Mental processes during domain abstraction reuse provide a framework for this study and were derived from existing psychological and computational models of analogical reasoning (Gick & Holyoak 1983, Hall 1989). Effective reuse necessitates:

- recognition of similarities between the target domain and domain abstraction;
- comprehension of the domain abstraction;
- formation of analogical mappings with the domain abstraction;
- transfer of knowledge from the domain abstraction to the target domain.

3.1: Presentation of Prototypical Examples

Presentation of prototypical examples can aid recognition and understanding of domain abstractions. Empirical studies of analogical problem solving revealed that understanding of abstractions was aided by presentation of two or more analogical instances (Gick & Holyoak 1983). Evidence also suggests that new concepts are understood using prototypical examples (e.g. Riesbeck & Schank 1989, Rosch 1983). In software engineering, EXPLAINER (Redmiles 1993) allows software designers to explore example designs and implementation solutions for graphics programming and adapt these examples through analogy to developing solutions. AIR (Maiden & Sutcliffe 1993a) presents different forms of prototypical example, although this paper only reports the effect of prototypical software engineering examples.

3.2: Visualisation of Domain Abstractions

Visualisation (Lewis 1992) of domain abstractions can also aid understanding. Gick (1989) reported better learning of abstractions from spatial diagrams representing critical determinants during analogical problem solving. Fundamental aspects of computer programs were understood more quickly and accurately when represented graphically (e.g. Cuniff & Taylor 1987). Domain abstractions and analogical instances share a network of interrelated knowledge structures (Gentner 1983) which discriminate between domain classes. Spatial diagrams provide the best medium for representing such complex knowledge structures. As a consequence, AIR presented state transitions and states as simple blockworlds (Gupta 1991). Diagrams were annotated with text representing other knowledge types as shown in Figure 1.

3.3: Summary

To sum, it is hypothesized that domain abstraction reuse can benefit from presentation of prototypical examples and visualisation of domain abstractions, however this claim has not been investigated experimentally. Therefore, their effect during domain abstraction reuse was investigated in the reported study. Software engineering authors rarely evaluate the usability of their products, therefore this study represents an alternative model for designing software engineering environments.

4: An Experimental Investigation of Domain Abstraction Reuse

Thirty-two inexperienced software engineers modelled the requirements of a system to control recall of patients for dental checkups. Reuse of the domain abstraction for object

inspection was aided by visualising the domain abstraction and presenting a prototypical example of that abstraction.

4:1: Experimental Method

The 32 (23M, 9F) software engineers were final-year undergraduate and postgraduate students in software engineering with little systems analysis experience, although most had knowledge of structured analytic techniques such as Jackson Systems Development (JSD, Jackson 1983) and Structured Systems Analysis and Design (SSADM, Cutts 1987). The subjects, aged between 20 and 37 years, volunteered their services and received no financial reward, although they were given coffee and chocolate biscuits during the task.

4.1.2: The Experimental Task

All subjects read a problem statement describing the target dental patient recall domain. The short narrative described the domain and requirements of the computerised system for dental patient recall. Subjects were asked to develop a context- and level-0 data flow diagram (DFD) and entity-relationship diagram for the patient recall domain. They were divided into 4 groups of 8 as shown in Table 1:

group-A: a control group in which subjects were given the target problem statement only;
group-B: an experimental group in which subjects were given the target problem statement and a text description of the object inspection abstraction;
group-C: an experimental group in which subjects were given the target problem statement and text and diagrammatic representations of the object inspection abstraction;
group-D: an experimental group in which subjects were given the target problem statement and text and diagrammatic representations of the object inspection abstraction. In addition, they were given text and diagrammatic representations of a vehicle inspection domain as the prototypical example of the domain abstraction.

Subjects were balanced across groups according to their limited systems analytic experience. They were given 30 minutes to develop the entity-relationship and data flow diagrams for the target patient recall domain as a pilot study indicated that this was sufficient time to complete the task. Experimental instructions were read by the experimenter and subjects read the problem statement prior to undertaking the task. All subjects were informed of the time limit beforehand. At the end of the task, subjects were required to complete a retrospective questionnaire which captured reuse strategies and behaviour.

For those readers unfamiliar with structured analytic techniques, DeMarco defines a DFD as "a network of related functions showing all interfaces between components" (p342). DFDs are made up of only four basic elements: data flows represented by named arcs, processes represented by circles, data stores represented by parallel lines, and data sources and sinks represented by curved rectangles. DFDs show a road map of the input-process-output interconnections from sinks. The method follows a process of top-down functional decomposition. Each process fulfils a goal and is decomposed into lower-level diagrams with subprocesses.

Knight (1983) defines an entity-relationship diagram as a network of entities, attributes of entities and relationships between entities. These diagrams represent a static view of data in a system. Entities represent permanent objects in the domain which have attributes. Relationships can represent physical, logical and functional links between entities.

4.1.3: Experimental Material

The dental patient system was chosen so that subjects had sufficient background knowledge of the target domain to undertake the task. The object inspection abstraction and vehicle inspection example could both be reused to model most of this dental patient recall domain and its requirements. The vehicle inspection example consists of two structures which represent the garage and region around the garage. Two state transitions describe the behaviour of vehicles which drive to and from the garage for their inspection. Vehicle inspection is triggered by an annual period between inspections.

The domain abstraction and prototypical example were defined using text and spatial diagrams. The text and diagram contained equivalent information except for exception processing (e.g. rescheduling) which was only described in the text. The diagrammatic notation had no constructs for representing exception processing. Mappings between the domain abstraction, dental patient and vehicle inspection domains are shown in Table 2. All experimental materials and the retrospective questionnaire are given in Tyndale (1993) and summarised in Appendix A.

4.1.4: Solution Completeness

Completeness scores were allocated to each subject's requirements specification to determine their success or otherwise in solving the problem. In order to construct a marking scheme, an expert solution was developed by a requirements engineer with knowledge of the match with the object inspection abstraction and the vehicle inspection example. The marking scheme contained a list of components to be included in the specification. It focused on semantic

features of subjects' solutions rather than on the syntax of the data flow and entity relationship diagramming notation. These components included processes, system inputs and outputs, external entities and data store entities in the expert solution. Subjects received a score if a component was included in their data flow diagrams, and each subject's completeness score represented the number of required components included in their solution. The expert solution to the dentist patient recall domain is shown in Appendix A.

4.1.5: Experimental Hypotheses

Domain abstraction reuse was expected to aid requirements specification (Reubenstein & Waters 1991, Maiden & Sutcliffe 1993a). Greater explanation of the domain abstraction was expected to improve understanding and transfer, thus leading to more complete requirements specifications. Three experimental hypotheses were based on analogical reasoning behaviour reported by Gick & Holyoak (1983):

- provision of domain abstractions during requirements engineering would promote reuse and improve the completeness of requirements specifications developed by inexperienced software engineers;
- recognition, understanding and adaptation of the domain abstraction would be aided by diagrammatic representation of domain abstractions;
- recognition, understanding and adaptation of the domain abstraction would be aided by presentation of a prototypical example of the domain abstraction.

All 32 subjects completed the task and answered the retrospective questionnaire about their requirements specification and reuse behaviour. Results are presented in the next section.

4.2: Results

Average completeness scores for all subjects are shown in Table 3. Subjects in groups-B and -C produced more complete requirements specifications than subjects in group-A, although this difference was not significant. One surprising finding was that group-D subjects developed requirements specifications which were less complete than subjects in group-A. Therefore, presentation of domain abstractions did not improve the completeness of requirements specifications. Indeed, presentation of domain abstractions with most explanation resulted in the least complete requirements specifications. Subjects claimed during retrospective questioning to find the task easier rather than difficult to complete (average 4.19 on a scale of 1-7 where 1 = very difficult, 7 = very easy), therefore poor analytic performance did not appear to be linked to a lack of knowledge about the target domain.

Subjects' solutions were examined for inclusion of components which were cued in the domain abstraction and prototypical example but not requested in the problem statement. The total number of components recognised by group is shown in Table 4. Subjects from groups-B and -C modelled more unrequested components than subjects from groups-A and -D, indicating that improved requirements specification performance may have been achieved by some domain abstraction reuse. However as before, inter-group differences were not significant.

4.2.1: Differences in Group Behaviour

Subjects' responses to the retrospective questionnaire were examined for differences in behaviour and attitudes to reuse. Subjects were requested to rate answers on a scale of 1-7 to permit statistical analyses of responses. The following questions revealed several significant inter-group differences.

What level of similarity did you find between the dentist domain and the domain abstraction text ?
Group-C and -D subjects found the domain abstraction to be more similar to the target dentist domain than did group-B subjects, see Table 5 (Z-score approximating t-test for non-normal distribution, group-C subjects $Z=-5.203$ $p<0.001$, group-D subjects $Z=-5.169$ $p<0.001$). The low average similarity rating claimed by group-B subjects (average rating 1.63/7) indicates that the text-only representation of domain abstractions did not appear to aid recognition of similarities.

How useful was the domain abstraction text for specifying the dentist domain ?
Group-C and -D subjects claimed to find the domain abstraction text more useful than did group-B subjects, see Table 5 (Z-score approximating t-test, group-C subjects $Z=-3.617$ $p<0.005$, group-D subjects $Z=-2.505$ $p<0.05$). Therefore, the domain abstraction text appeared to be more useful to subjects when the domain abstraction diagram was also provided.

How easy did you find the domain abstraction text to understand ?
Group-C and -D subjects claimed to understand the text description of the domain abstraction more easily than did group-B subjects, see Table 5 (Z-score approximating t-test, group-C subjects $Z=-3.813$ $p<0.005$, group-D subjects $Z=-2.909$ $p<0.02$). That is, subjects claimed to understand the domain abstraction text more easily when also given the domain abstraction diagram.

Therefore, responses to retrospective questions indicated the following general patterns of behaviour:

- subjects given the domain abstraction diagram claimed to find the domain abstraction and target domain more similar than subjects who did not receive the diagram;
- subjects given the domain abstraction diagram claimed to find the domain abstraction text more useful for modelling the target domain. However, this usefulness was not manifest as more complete requirements specifications for all subjects;
- subjects given the domain abstraction diagram claimed to find the domain abstraction text easier to understand.

That is, some subjects seemed to exploit the text but needed the diagram to recognise similarities with and understand the text. There was not effect from provision of the prototypical example.

4.2.2: Reusers and Non-Reusers in Groups-B, -C and -D

The poor effect from reuse indicated that many subjects may not have exploited the domain abstraction. Each subject in groups-B, -C and -D was defined as a *reuser* or *non-reuser* according to retrospective claims about their overall level of reuse from the domain abstraction text. Subjects who claimed high levels of reuse (minimum score 5 out of 7) were classified as reusers while subjects who claimed low levels (3/7 or less) were classified as non-reusers. The one remaining subject, who claimed a level of reuse of 4/7, was classified as a non-reuser due to her failure to produce any requirements specification, as discussed in 4.2.4.

Only 6 subjects (2 group-B and 4 group-C) were classified as reusers, as shown in Table 6. Therefore, poor analytic performance may have been due to the low levels of reuse which occurred. Requirements specifications produced by reusers were significantly more complete than those of non-reusers (Z-score approximating t-test $Z=4.565$ $p<0.001$) and group-A subjects (Z-score approximating t-test $Z=2.79$ $p<0.02$) while requirements specifications produced by non-reusers in groups -B, -C and -D showed no improvement over specifications developed by group-A subjects. Therefore, successful reuse of domain abstractions improved the completeness of requirements specifications produced by subjects in this study, however most subjects did not reuse. There appeared to be many reasons for failure to reuse.

4.2.3: Successful Reusers

Reusers included more components in their specifications which were cued in the domain abstraction but not the problem statement, see Table 7. Four reusers claimed to reuse the domain abstraction text to transfer functions and processes. Indeed the group-C subject, who produced the most complete requirement specification, claimed that the text "*helped identify components which would otherwise have been missed*", such as data flows in the data flow diagram and entities in the entity relationship diagram. Therefore, successful reuse may have led to the inclusion of some additional components in the requirements specification. However, these components did not account for all differences between reusers' and non-reusers' specifications, so reuse may also have had an indirect effect on subjects' behaviour.

Retrospective questioning revealed that the 2 group-B reusers claimed a weaker understanding of the domain abstraction text than the 4 reusers in group-C (group-B average = 3/7, group-C average = 4.75/7). Therefore, as reported for all subjects, the diagram may have aided domain understanding by group-C reusers. On the other hand, reusers and non-reusers in group-C claimed equal average levels of understanding of the domain abstraction. Therefore, better understanding did not appear to be a factor during reuse of the domain abstraction. Indeed, retrospective questioning also revealed that even 3 reusers found the domain abstraction text difficult to understand and required several attempts to comprehend it.

4.2.4: Unsuccessful Reusers

The behaviour of non-reusers was examined more closely to determine reasons for reuse failure. Psychological models of analogical reasoning (e.g. Gick & Holyoak 1983, Ross 1987) suggest at least 3 possible reasons for failure to reuse:

- failure to recognise similarities between the domain abstraction and target domain;
- failure to form sufficient mappings with the domain abstraction;
- failure to transfer the domain abstraction to the target domain.

Each non-reuser was allocated to one of these reasons according to responses to 2 retrospective questions. These questions captured subjects' perceived degree of similarity with the domain abstraction and understanding of the abstraction. Reasons for failure are shown in Table 8. All non-reusers except one group-C subject were classified according to the scheme. The remaining group-C subject's behaviour was unique because she mapped to and understood the domain abstraction but preferred to transfer the abstraction diagram rather than develop the requested structured diagrams. This strategy coincided with improved domain abstraction understanding but led to an incomplete requirements specification in the time allowed.

Recognition Failure

Four subjects in group-B did not recognise any similarities with the domain abstraction. Two of these subjects did not see any practical relevance of the abstraction and rejected it. Another subject claimed that the terminology in the domain abstraction text was too complex and did not warrant the effort to form mappings with it. Therefore, text-only representation of the domain abstraction did not appear to promote recognition of similarities by 50% of subjects.

Mapping Failure

Five subjects from groups-B, -C and -D recognised some similarities with the domain abstraction. These similarities appeared to lead to minor levels of reuse. For example, 2 subjects reused the domain abstraction to model part of their entity-relationship diagrams. Overall, 6/8 subjects in group-B recognised no or some similarities with the domain abstraction, therefore reuse again may have failed because of poor recognition of similarities with the domain abstraction text.

Transfer Failure

Two group-C subjects and 6 group-D subjects failed to transfer the domain abstraction despite understanding the abstraction and forming mappings between it, the prototypical example and the target domain. The retrospective questionnaire identified 4 possible reasons for failure to transfer:

- *insufficient analogical mapping*. One group-C subject claimed that the domain abstraction did not provide 'full' analogical mappings with the target domain, so it was ignored. Another group-D subject failed to recognise key mappings (e.g. patient/object) despite mapping a lower-level process between the two domains;
- *complexity of understanding.* One group-D subject ignored the domain abstraction and prototypical example because he did not want to 'cloud the mental picture of the dentist that was being formed';
- *requirements modelling notation*. The same group-D subject claimed that diagrammatic representation discouraged direct reuse because it could not be transferred to the requirements specification;
- *attitudes towards reuse.* Two group-D subjects claimed that domain abstraction reuse was unnecessary for modelling the target domain. Such claims have implications about reusers' attitudes about the complexity of requirements engineering and benefits from domain abstraction reuse.

Four of the 6 group-D subjects attributed to transfer failure also listed between-domain mappings during retrospective questioning, however they defined mappings with the prototypical example rather than the domain abstraction. Indeed, 2 subjects claimed that difficulties in understanding the domain abstraction text were alleviated by the prototypical example which helped to understand the abstraction diagram. Therefore, subjects appeared to prefer to form mappings with prototypical examples, with implications for strategies for effective mapping formation and domain understanding.

4.2.5: Summary of Subject Behaviour

This study revealed that domain abstraction reuse, assisted by domain visualisation and prototypical examples, was complex and variable. In particular:

- presentation of domain abstractions did not improve the completeness of requirements specifications, due to failure to reuse by most subjects;
- successful reuse could be linked to more complete requirements specifications, however reuse often failed due to poor recognition of similarities with, understanding and transfer of the domain abstraction;
- successful reuse appeared to occur from the domain abstraction text;
- the domain abstraction diagram appeared to improve understanding of the text, suggesting a cumulative effect on comprehension of text and diagram;
- presentation of the domain abstraction text without the diagram led to poorer recognition of similarities;
- subjects who understood the domain abstraction failed to transfer due to perceived insufficient mappings, complexities in understanding the domain abstraction, the lack of a transferable notation and failure to perceive benefits from reuse;
- there was no significant effect from the prototypical example, however subjects defined analogical mappings with the example rather than the abstraction.

These findings are discussed in the remainder of the paper.

4.3: Discussion

Results were surprising and suggested that domain abstraction reuse by inexperienced software engineers was more complex and difficult than hypothesized. Referring back to the first hypothesis, successful domain abstraction reuse improved requirement specification completeness, however successful reuse was problematic. For the second hypothesis, diagrammatic representation aided recognition of similarities and understanding but not

transfer of domain abstractions. Third, there was no effect from the prototypical example, so the final hypothesis was rejected. Reuse necessitates difficult mental tasks to recognise, map and transfer on top of complex requirements engineering tasks including structuring, scoping and formation of complex mental models of the domain (e.g. Guindon 1990, Sutcliffe & Maiden 1992). Therefore, the difficulty of reuse during requirements engineering may have been underestimated, and it is not surprising that software engineers sometimes avoid reuse and select strategies requiring less mental effort.

Visualisation appeared to aid understanding and recognition of similarities with the domain abstraction (e.g. Gick & Holyoak 1983). Analogies are defined by shared, interrelated knowledge structures (Gentner 1983, Holyoak & Thagard 1989, Maiden & Sutcliffe 1992), so spatial diagrams which draw attention to structure may have improved recognition and understanding. Cumulative effects on understanding from text and diagram have been recognised during design of hypermedia systems (e.g. Bottrill & Lock 1993). Studies (e.g. Hegarty & Just 1993) have shown that conjunction of text and pictures facilitated comprehension of mechanical systems with causal, physical behaviour. Readers integrated information in text and graphics in local units which were then combined at more global levels (p. 739). Causal and physical behaviour are also modelled in domain abstractions. Therefore, as results indicate, integrated text and graphics would seem to be an effective medium for explanation. Furthermore, diagrammatic representation has been shown for effective recall of spatial information (e.g. Taylor & Tversky 1992), indicating that diagrammatic representations could promote deep understanding of domain abstractions. Deep understanding of abstractions will be investigated in subsequent studies.

Fundamental aspects of computer programs were also understood more quickly and accurately when represented graphically (e.g. Cuniff & Taylor 1987). Graphic illustrations also draw attention to key component features such as data structures and operations in programs to act as beacons for program understanding (Gellenbeck & Cook 1991). However, Green et al. (1991) report from studies of comprehension with visual programming languages that '*what a programmer sees is largely a matter of training*'. Therefore, software engineers can benefit from training with the visual notation prior to reuse. Furthermore, integrating this notation with more familiar structured notations such as data flow diagrams might assist comprehension, although further work is needed in this direction.

Domain abstraction reuse necessitates analogical reasoning from one unfamiliar, generic analog (e.g. Gick 1989). There is strong evidence that analogical recognition is difficult between semantically remote domains such as the dental patient recall domain and its abstraction (e.g. Keane 1987, Ross 1987, 1989). Results from presentation of the domain abstraction text only support this assertion. Therefore, strategies such as domain visualisation

are imperative for improvement of similarity recognition. Analogical comprehension is also problematic (Gick 1989). Issing et al. (1989) report that complex pictoral analogies must be understood, through support for decoding of the analogy in the correct order. It is possible that the software engineers imposed order on decoding the domain abstraction picture using the text, a finding reported by Hegarty & Just (1993).

Furthermore, Issing et al. (1989) report that pictorial analogies should be taken from the natural world of things rather than be artificial or constructed, to ensure a cognitive function for analogical reasoning. Domain abstractions and the car testing example are both artificial constructs. Therefore, prototypical examples of abstractions which are drawn from the natural world might aid comprehension and hence mapping. Further empirical studies will investigate this hypothesis.

Previous studies investigated analogical reasoning to solve simple problems (e.g. Gick & Holyoak 1983, Ross 1989). Requirements engineering, on the other hand, is a complex problem solving task which distinguishes the problem and solution spaces. Spatial diagrams represented fundamental domain features but not the solution space. This can explain poor transfer despite effective recognition and understanding of the abstraction, a finding also reported by Novick & Holyoak (1991). Distinguishing between problem and solution spaces also has implications for case-based design such as architectural (Voss & Schmidt-Belz 1993) or industrial design engineering (Kruger 1993). Designers need to recognise similarities, understand and transfer between both the problem and solution spaces. However, support for these cognitive activities during case-based design is lacking.

The lack of diagrammatic representations of the solution space appeared to limit transfer. In light of this, software engineers guided transfer of the domain abstraction text using its sequence. Previous studies revealed that software engineers exploit the structure of analogical specifications to guide transfer (e.g. Maiden & Sutcliffe 1992b), although this can be lead to comprehension avoidance manifest as copying (Sutcliffe & Maiden 1990, Lange & Moher 1989). Experienced software engineers interleaved similarities recognition, domain understanding and solution transfer during analogical reuse (Maiden & Sutcliffe 1992b). Therefore, integrated assistance which mimics experienced software engineering practice is one solution for recognition, understanding and transfer. This assistance is reported in Maiden (1992).

Software engineers who received the prototypical example exhibited least reuse. This contrasts with the effective exploitation of examples in program comprehension (Neal 1989) and reuse (Redmiles 1993). Evidence indicates that software engineers formed mappings with and understood the example but this had no effect on transfer (e.g. Novick & Thagard 1991).

This result was unexpected. Analogical instances have been shown to assist induction of mental abstractions (Gick & Holyoak 1983), and prototypical examples provided an intermediate stage during mapping formation with domain abstractions. However, results indicate the need for additional procedural guidance during explanation to exploit prototypical examples.

Results inform design of AIR (the Advisor for Intelligent Reuse) which aids domain abstraction reuse within the ESPRIT 'Nature' project (Jarke et al. 1993). A computational mechanism for analogical reasoning known as the domain matcher retrieves domain abstractions (Maiden & Sutcliffe 1991, 1993c) and infers mappings for explanation. Results here inform design of the requirements critic (Maiden & Sutcliffe 1994) to aid recognition, understanding and transfer of retrieved abstractions. The critic is one of the few software engineering environments designed from findings in cognitive science and empirical studies of software engineering practice.

The critic explains domain abstractions using integrated text and graphics, unlike the passive explanation examined in this paper. Gradual exposure of text directs the reader's attention to graphics. This can overcome some of the reported poor domain understanding. Dialogue control (Maiden & Sutcliffe 1994) contrasts with existing browsers for software reuse which leave the initiative with the software engineer (e.g. Redmiles 1993, Green et al. 1992). Furthermore, explanation of domain features is interleaved with presentation of analogical specification parts to guide knowledge transfer (Maiden & Sutcliffe 1992b).

Tool-based guidance supporting explanation of prototypical examples can aid formation of mappings to domain abstractions. Tabular notations encourage object-pair mappings as demonstrated by experienced software engineers (Maiden & Sutcliffe 1992b). Provision of prototypical examples from the natural world of things ensures that example has a cognitive as well as a motivating function (Issing et al. 1989). Presentation of more than one prototypical example can promote better mental abstraction (Gick & Holyoak 1983). The requirements critic is defined in more detail in Maiden & Sutcliffe (1994).

5: Conclusions

Results revealed that domain abstraction reuse is complex and problematic without effective explanation. Individual strategies for explanation and guidance can improve domain abstraction reuse, however the real payoff is expected from integrating these strategies into the requirements critic (Maiden & Sutcliffe 1994) to explain domain abstractions and guide transfer, as part of the ESPRIT 6353 Nature basic research action (Jarke et al. 1993). We believe that this intelligent cooperation will enable effective reuse of domain abstractions as

an alternative to expensive and time-consuming, application-specific domain modelling (e.g. Arango et al. 1993a).

Implications for tool design demonstrate benefits from empirical evaluation of design concepts prior to implementation. The results reported in this paper were not foreseen by the authors, with implications for poor tool design if the study had not been undertaken. Software engineering has emerged as a cooperative task (Andriole & Freeman 1993) which defies earlier predictions of its automation (Balzar et al. 1983), therefore further empirical evaluation of tool designs is needed to ensure that the genuine requirements of software engineers are met. The authors hope that this paper will inspire software engineering authors to validate designs through empirical evaluation, thus improving future software engineering practice.

Acknowledgements

The authors extend their thanks to all students in the Department of Business Computing, City University who gave their time to take part in the study. This also wish to thank Alistair Sutcliffe and Peter Faraday for their comments and assistance.

References

Andriole S.J. & Freeman P.A., 1993, 'Software systems engineering: the case for a new discipline', *Software Engineering Journal* **8**(3), 165-179.

Arango G., Schoen E. & Pettengill R., 1993a, 'Design as Evolution and Reuse', Proceedings of 2nd International Workshop on Software Reusability (Advances in Software Reuse), IEEE Computer Society Press, 9-18.

Arango G., Schoen E., Pettengill R. & Hoskins J., 1993b, 'The graft-host method for design change', Proceedings of ICSE-15, Baltimore 17-21 May 1993, IEEE Computer Society Press, 243-254.

Arango G., Schoen E. & Pettengill R., 1993c, 'A process for consolidating and reusing design knowledge', Proceedings of ICSE-15, Baltimore 17-21 May 1993, IEEE Computer Society Press, 231-242.

Balzer R., Cheatham T.E. & Green C., 1983, 'Software technology in the 1990's: using a new paradigm', *IEEE Computer*, **November 1983**, 39-45.

Bottrill J. & Lock R., 1993, 'Do students learn more from pictures of from text? - a pilot study', *School Science Review*, **74**(269), 109-112.

Cheng P.W. & Holyoak K.J., 1985, 'Pragmatic reasoning schemas', *Cognitive Psychology* **17**, 391-416.

Chi M.T.H., Bassok M., Lewis M.W., Reimann P. & Glaser R., 1989, 'Self-explanations: how students study and use examples in learning to solve problems', *Cognitive Science* **13**, 145-182.

Cunniff N. & Taylor R.P., 1987, 'Graphical vs textual representation: an empirical study of novices' program comprehension', in *Empirical Studies of Programmers, Second Workshop*, eds G. Olsen, S. Sheppard and E. Soloway, Ablex, Norwood NJ, 114-131.

Cutts G., 1987, 'SSADM - Structured systems analysis and design methodology', Paradigm Publishing.

DeMarco T., 1978, 'Structured systems analysis and specification', Prentice-Hall International.

Easterbrook S., 1991, 'Handling conflict between domain descriptions with computer-supported negotiation', *Knowledge Acquisition* **3**, 255-289.

Fischer G., 1992, 'Domain-oriented design environments', Proceedings 7th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, 204-215.

Gellenbeck E.M. & Cook C.R., 1991, 'An investigation of procedure and variable names as beacons during program comprehension', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson, Ablex, Norwood NJ, 65-81.

Gentner D., 1983, 'Structure-mapping: a theoretical framework for analogy', *Cognitive Science* **5**, 121-152.

Gick M.L., 1989, 'Two functions of diagrams in problem solving by analogy', Knowledge Acquisition from Text and Pictures, ed. H Mandi & J.R. Levin, Elsevier Science Publishers B.V. (North-Holland) 1989, 215-231.

Gick M.L. & Holyoak K.J., 1983, 'Schema induction and analogical transfer', *Cognitive Psychology* **15**, 1-38.

Green T.R.G., Gilmore D.J., Blumenthal B.B., Davies S. & Winder R., 1992, 'Towards a cognitive browser for OOPS', *International Journal on Human-Computer Interaction* **4(1)**, 1-34.

Green T.R.G., Petre M. & Bellamy R.K.E., 1991, 'Comprehensibility of visual and textual Programs: a test of superlativism against the 'mismatch' conjecture', in *Empirical Studies of Programmers, fourth Workshop*, ed. J. Koenemann-Belliveau, T.G. Moher & S. Robertson, Ablex, Norwood NJ, 121-146.

Guindon R., 1990, 'Designing the design process: exploiting opportunistic thoughts', *Human-Computer Interaction* **5**, 305-344.

Guindon R., Krasner H. & Curtis B., 1987, 'Breakdowns and processes during the early activities of software design by professionals', in *Empirical Studies of Programmers, Second Workshop*, eds G. Olsen, S. Sheppard and E. Soloway, Ablex, Norwood NJ, 65-82.

Gupta N. & Nau D.S., 1991, 'Complexity results for blocks-world planning', Proceedings of AAAI'91/MIT Press, 629-633.

Hall R.P., 1989, 'Computational approaches to analogical reasoning: a comparative analysis', *Artificial Intelligence* **39**, 39-120.

Hegarty M. & Just M.A., 1993, 'Constructing mental models of machines from text and diagrams', *Journal of Memory and Language* **32**, 712-742.

Issing L.J., Hannemann J. & Haack J., 1989, 'Visualisation by Pictorial Analogies in Understanding Expository Text', Knowledge Acquisition from Text and Pictures, ed. H Mandi & J.R. Levin, Elsevier Science Publishers B.V. (North-Holland) 1989, 195-214.

Jackson M., 1983, 'Systems development', Prentice-Hall International.

Jarke M., 1992, ConceptBase V3.1 user manual, RWTH-Aachen, Aachen, Germany.

Jarke M., Bubenko Y., Rolland C., Sutcliffe A.G. & Vassiliou Y., 1993, 'Theories underlying requirements engineering: an overview of NATURE at genesis', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press, 19-31.

Keane M., 1987, 'On Retrieving Analogues When Solving Problems', The Quarterly Journal of Experimental Psychology 39A, 29-41.

Knight B., 1983, 'A mathematical basis for entity analysis', Entity-Relationship Approach to Software Engineering', edited by C.G. Davis, S. Jajodia, P.A. Ng and R.T. Yeh, Elsevier Science Publishers B.V. (North-Holland), 81-90.

Kruger C., 1993, 'Cognitive aspects of reuse in industrial design engineering', Proceedings of Workshop: Reuse of Designs: An Interdisciplinary Approach (IJCAI-13), Technical Report of Institut National de Recherche en Informatique et an Automatique.

Lange B.M. & Moher T.G., 1989, 'Some strategies of reuse in an object-oriented programming environment', Proceedings of CHI'89, ed. K. Bice & C. Lewis, ACM Press, 69-73.

Leite J.C.S.P. & Freeman P.A., 1991, 'Requirements validation through viewpoint resolution', *IEEE Transactions on Software Engineering* **17**(12), 1253-1269.

Lewis M., 1992, 'Situated visualisation: building interfaces from the mind up', *Multimedia Review*, **Winter 1991/Spring 1992**, 23-39.

Maiden N.A.M., 1992, 'Analogical specification reuse during requirements analysis', PhD. Thesis, Department of Business Computing, City University, July 1992.

Maiden N.A.M. & Sutcliffe A.G., 1994, 'Requirements critiquing using domain abstractions', Proceedings 1st International Conference on Requirements Engineering, IEEE Computer Society Press.

Maiden N.A.M. & Sutcliffe A.G., 1993a, 'Requirements engineering by example: An empirical study', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press 104-112.

Maiden N.A.M. & Sutcliffe A.G., 1993b, The domain theory: object system definition', Nature Technical Report CU-93-00A, June 1993, Department of Business Computing, City University.

Maiden N.A.M. & Sutcliffe A.G., 1993c, 'Architecture and algorithms of the domain matcher', Nature Technical Report CU-93-00D, June 1993, Department of Business Computing, City University.

Maiden N.A.M. & Sutcliffe A.G., 1992a, 'Exploiting reusable specifications through analogy', *Communications of the ACM*, **34**(5), April 1992, 55-64.

Maiden N.A.M. & Sutcliffe A.G., 1992b, 'Analogously-based reusability', *Behaviour & Information Technology* **11(2)**, 79-98.

Maiden N.A.M & Sutcliffe A.G., 1991, 'Analogical matching for specification retrieval', Proceedings 6th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, 108-116.

Meyer B., 1985, 'On formalism in specifications', *IEEE Software*, **January 1985**, 26.

Mylopopoulos J., Borgida A., Jarke M. & Koubarakis M, 1990, 'Telos: representing knowledge about information systems', *ACM Transactions on Office Information Systems* **8(4)**, 325.

Neal L.R., 1989, 'A system for example-based programming', Proceedings of CHI'89, ed. K. Bice & C. Lewis, ACM Press, 63-68.

Neighbors J.M., 1984, 'The Draco approach to constructing software from reusable components', *IEEE Transactions on Software Engineering* **10**(5), 564-574.

Novick L.R., 1988, 'Analogical transfer, problem similarity, and expertise', *Journal of Experimental Psychology: Learning, Memory and Cognition* **14**(3), 510-520.

Novick L.R. & Holyoak K.J., 1991, 'Mathematical Problem Solving by Analogy', Journal of Experimental Psychology: Learning, Memory, and Cognition 17(3), 398-415.

Payne S.J., Chesworth L. & Hill E., 1992, 'Animated demonstrations for exploratory learners', *Interacting With Computers* **4**(1), 3-22.

Rich C., 1981, 'Inspection methods in programming', Technical Report TR-604, Cambridge MA, Artificial Intelligence Laboratory, MIT.

Redmiles D.F., 1993, 'Reducing the variability of programmers' performance through explained examples', Proceedings of INTERCHI'93, ed. S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel & T. White, ACM Press, 67-73.

Reubenstein H.B. & Waters R.C., 1991, 'The Requirements Apprentice: automated assistance for requirements acquisition', *IEEE Transactions on Software Engineering* **17**(3), 226-240.

Riesbeck C.K. & Schank R.C., 1989, 'Inside case-based reasoning', Lawrence Erlbaum Associates, Hillsdale NJ.

Roman G., 1985, 'A taxonomy of current issues in requirements engineering', *IEEE Computer*, **April 1985**, 14-22.

Rosch E., 1983, 'Prototype classification and logical classification: the two systems', New Trends in Conceptual Representation: Challenges to Piaget's Theory, edited K. Scholnick, Lawrence Erlbaum Associates, Hillsdale NJ.

Ross B.H., 1989, 'Distinguishing Types of Superficial Similarities: Different Effects on the Access and Use of Earlier Problems', Journal of Experimental Psychology: Learning, Memory and Cognition 15(3), 456-468.

Ross B.H., 1987, 'This is like that: the use of earlier problems and the separation of similarity effects', *Journal of Experimental Psychology: Learning, Memory and Cognition* **13**(4), 629-639.

Soloway E. & Enrlich K., 1984, 'Empirical studies of programming knowledge', *IEEE Transactions on Software Engineering* **10(5)**, 595-609.

Sutcliffe A.G. & Maiden N.A.M., 1992, 'Analysing the novice analyst: cognitive models in software engineering', *International Journal of Man-Machine Studies* **36**, 719-740.

Sutcliffe A.G. & Maiden N.A.M., 1990, 'How specification reuse can support requirements analysis', Proceedings of Software Engineering'90, edited by P. Hall, Brighton UK, July 24-27 1990, 489-509.

Taylor H.A. & Tversky B., 1992, 'Descriptions and depictions of environments', *Memory and Cognition* **20(5)**, 483-496.

Tyndale D.E., 1993, 'Understanding domain abstractions during requirements engineering', Technical Report, Department of Business Computing, City University.

Voss A. & Schmidt-Belz B., 1993, 'Case-oriented knowledge acquisition for architectural design', Proceedings of Workshop: Reuse of Designs: An Interdisciplinary Approach (IJCAI-13), Technical Report of Institut National de Recherche en Informatique et an Automatique.

Illustrations and Captions

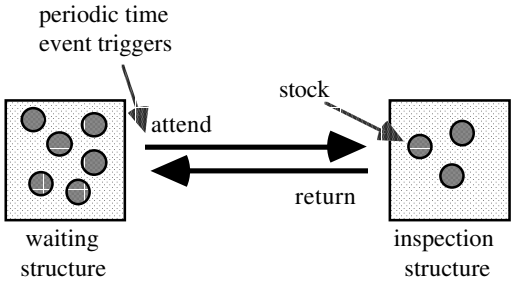


Figure 1(a) - diagrammatic representation of the periodic object recall domain abstraction

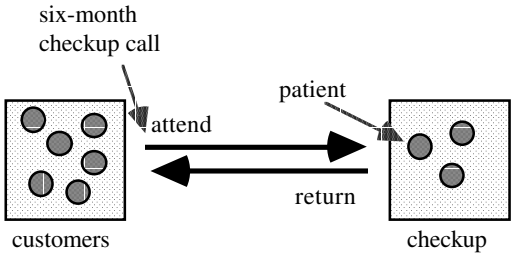


Figure 1(b) - diagrammatic representation of the dental patient checkup domain

experimental material given	group-A	group-B	group-C	group-D
problem statement for the target dental patient domain	¦	¦	¦	¦
periodic object recall domain abstraction text		¦	¦	¦
periodic object recall domain abstraction diagram			¦	¦
prototypical example - vehicle inspection domain text and diagram				¦

Table 1 - experimental material presented to subjects, by group

target domain objects	domain abstraction objects	prototypical example objects
patient	object	vehicle
dentist	inspector	mechanic
dentist surgery	inspection space	garage
dentist catchment area	object space	garage catchment area
dental checkup	object inspection	vehicle inspection

Table 2 - fundamental analogical mappings between objects in the target domain, domain abstraction and prototypical example

	group-A	group-B	group-C	group-D
average completeness score as a %age of expert model	36.1	41.1	43.6	32.5

Table 3 - average completeness scores by subjects in group, as a percentage of the expert's requirements specification

	group-A	group-B	group-C	group-D
average completeness score for components cued by the domain abstraction only	2.25	3.13	3.25	2.5

Table 4 - average completeness scores for components cued in the domain abstraction only by subjects in group, as a percentage of the expert's requirements specification

retrospective questions	group-B	group-C	group-D
perceived degree of similarity with the domain abstraction	1.625	4.837	5
representation on domain abstraction understanding	2.5	4.75	4.625
degree of reuse from domain abstraction text	2.25	3.875	2

Table 5 - average response scores to retrospective questions by subjects in group

	reusers	non-reusers
average completeness score as a %age of expert model	19.83	11.82
number of subjects	6	17

Table 6 - average completeness scores by subjects classified as reusers and non-reusers, as a percentage of the expert's requirements specification

requirements components	reusers	non-reusers
checkup cancellation	50%	11%
perform dental checkup	67%	55%
reschedule dental checkup	83%	11%

Table 7 - percentage of reusers and non-reusers who included processes in their requirements specifications. Processes were cued in the domain abstraction but were not requested in the target problem statement.

categories of reuse failure	no of subjects		
	group-B	group-C	group-D
failure to recognise similarities (level of similarity=1)	4	0	0
recognition of some similarity (level=2 or 3) but poor understand ing of domain abstraction text (level of understanding Š3)	2	1	2
recognition of similarities (level •4) and understanding of domain abstraction text (level•4)	0	2	6

Table 8 - categories of reuse failure showing numbers of subjects allocated to each category by subject group

Appendix A: Key Experimental Materials Given to Subjects

A1: Problem Statement

Doctor Brian Smith is a qualified dental surgeon. He established his practice in Beckingham over twenty years ago. He runs this practice with his partner - Doctor Jane Wakeham and the secretary - Miss Tracey Long. Until six months ago they ran this practice via an appointment book; an index card system of patient name and address; and folders with dental records. All this information was converted to a computerised data base. The system stores the name and address of patients and records dental records. It is not concerned with patients joining and leaving the practice. Its main use is to record patient checkups and issue six monthly calls for patients.

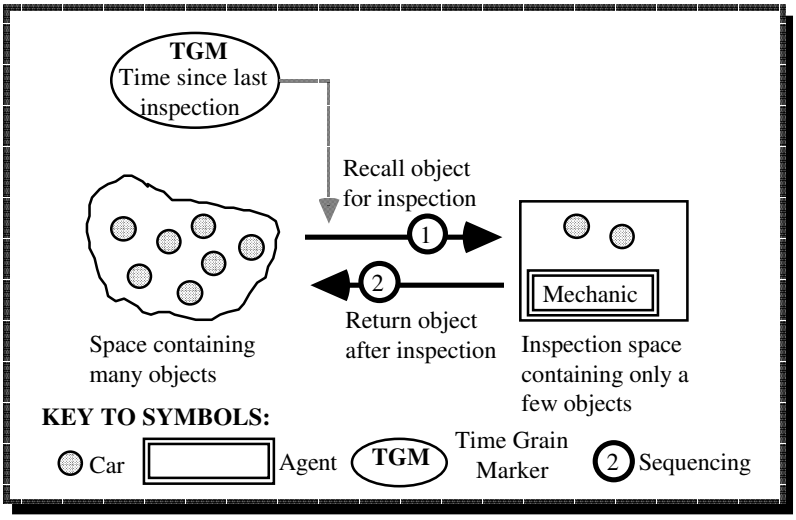
A2: Reusable Domain Abstraction

The reusable domain abstraction was composed of four modules presented to subjects depending on their experimental group. Group-B subjects received the domain abstraction text only, group-C subjects also received the domain abstraction diagram and group-D subjects also received the prototypical example text and diagram.

A2.1: Domain Abstraction Text

Many objects exist in a space and most are periodically called to a space for inspection or other purposes. The system records the existence of the object, a call for inspection of the object and a record of that inspection once it had occured. The objects are notified when to attend the space. Inspection periods are most often temporal or linked to usage of the object. The inspection itself is conducted by one agent operating in the space. One or many objects can be inspected at any given time but this is managed to ensure the effective inspection of all objects. The inspection may be postponed and rearranged if the initial inspection cannot be attended. Objects may be admitted for inspection at short notice. In exceptional circumstances the object may fail to attend the initial inspection.

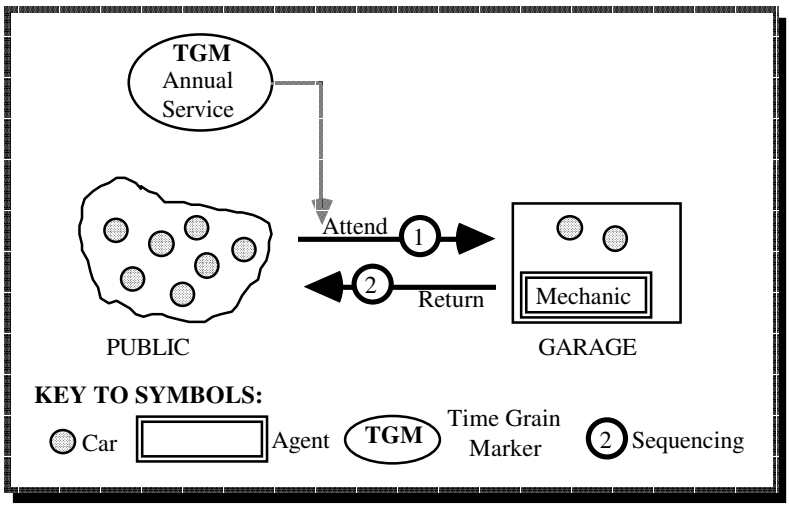
A2.2: Domain Abstraction Diagram



A2.3: Prototypical Example Text

In the car servicing domain, customers are informed that their car is due for its annual MOT service by telephone or letter. In the garage a number of cars are being serviced. The service is conducted by a mechanic and a record of the service is kept. Once the service is complete, the car is returned to the customer. If the original service date is inconvenient, the service is rescheduled. Some customers forget to keep the appointment resulting in the allocated time being used for other repair/maintenance work. Customers will therefore send their car to the garage at some other time.

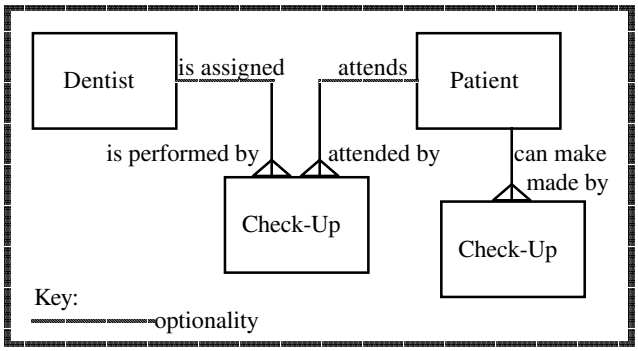
A2.4: Prototypical Example Diagram



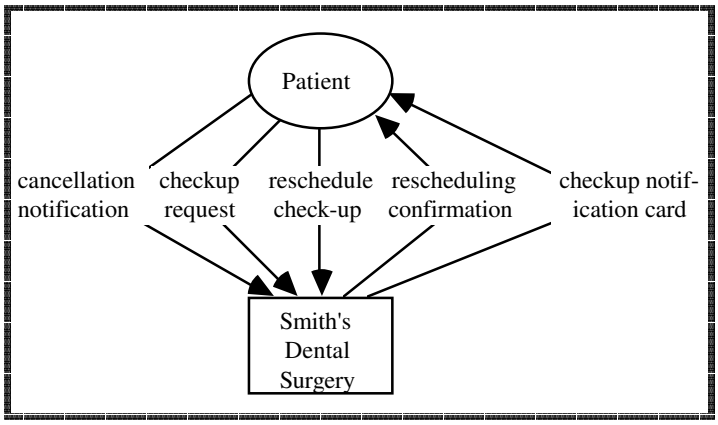
A3: Expert Solution to the Requirements Modelling Problem

The expert solution consisted of one entity-relationship diagram and two data flow diagrams requested for the task. These solutions provided the basis for calculating the completeness of requirements models.

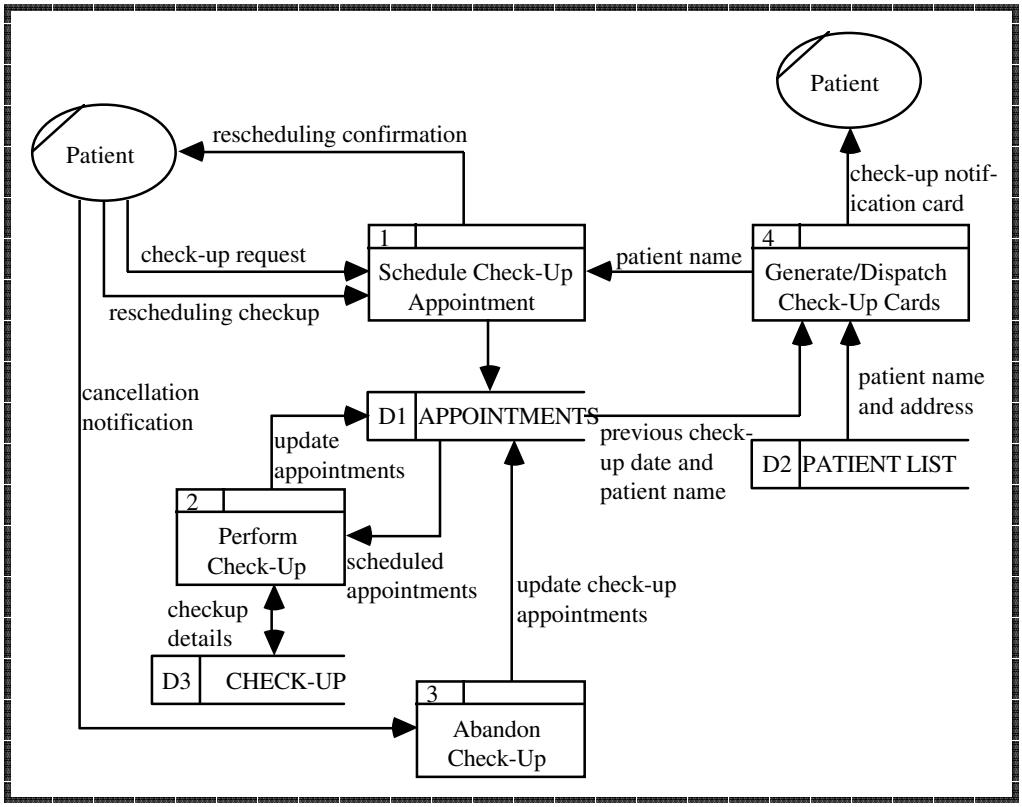
A 3 . 1 : E n t i t y - R e l a t i o n s h i p D i a g r a m :



A 3 . 2 : T h e C o n t e x t D a t a F l o w D i a g r a m :



A 3 . 3 : T h e L e v e l - 0 D a t a F l o w D i a g r a m :


Patient
rescheduling confirmation
Patient
check-up notif-
ication card
check-up request
1
Schedule Check-Up
Appointment
patient name
4
Generate/Dispatch
Check-Up Cards
rescheduling checkup
cancellation
notification
D1 APPOINTMENTS
patient name
and address
update
appointments
previous check-
up date and
patient name
D2 PATIENT LIST
2
Perform
Check-Up
scheduled
appointments
update check-up
appointments
checkup
details
D3 CHECK-UP
3
Abandon
Check-Up