



The Influence of the Level of Abstraction on the Evolvability of Conceptual Models of Information Systems

JAN VERELST

jan.verelst@ua.ac.be

Department of Management Information Systems, University of Antwerp, Antwerp, Belgium

Editors: Forrest Shull and Natalia Juristo

Abstract. In today's dynamic environments, evolvability of information systems is an increasingly important characteristic. We investigate evolvability at the analysis level, i.e., at the level of the conceptual models that are built of information systems (e.g., UML models). More specifically, we focus on the influence of the level of abstraction of the conceptual model on the evolvability of the model. Abstraction or genericity is a fundamental principle in several research areas such as reuse, patterns, software architectures and application frameworks. The literature contains numerous but vague claims that software based on abstract conceptual models has evolvability advantages. Hypotheses were tested with regard to whether the level of abstraction influences the time needed to apply a change, the correctness of the change and the structure degradation incurred. Two controlled experiments were conducted with 136 subjects. Correctness and structure degradation were rated by human experts. Results indicate that, for some types of change, abstract models are better evolvable than concrete ones. Our results provide insight into how the rather vague claims in literature should be interpreted.

Keywords: Conceptual modelling, evolvability, maintainability, abstraction.

1. Introduction

Information systems support organizations that function in changing environments. Changes in for example legal obligations, market dynamics, internal reorganizations, mergers and acquisitions require timely responses if an organization is to survive in an increasingly competitive environment. In these circumstances, it is of increasing importance that information systems that support the organization, can be adapted to changing requirements of the organization and its end users. This adaptive quality is generally described with terms such as maintainability, flexibility or evolvability. IEEE, for example, defines maintainability as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changing environment” (IEEE, 1994).

Evolvability has received increasing attention over the last 10 years in the academic literature, both in theoretical and empirical work. However, most studies are aimed at evolvability of program code or software designs. This study focuses on evolvability of conceptual models built during the analysis phase of the development cycle. In an object oriented context, these models are likely to be Unified Modeling Language (UML)

models; in structured development Entity Relationship (ER) models and Data Flow Diagrams (DFD's).

In this study, our goal is to investigate whether the level of abstraction or genericity of a conceptual model influences the evolvability of the conceptual model. Abstraction or genericity is a fundamental principle in several research areas such as reuse, patterns, software architectures and application frameworks. Literature contains numerous claims that software based on abstract conceptual models has evolvability advantages. A typical example is Helm (1995) who introduces patterns as “idiomatic and recurring patterns and structures of communicating objects that solve particular design problems and make designs more flexible, elegant and ultimately reusable.” Although many claims of this type can be found in the software engineering literature, they are typically so vague that many open issues remain. For example, these claims do not specify if the advantages are situated at the analysis level; it is possible that these advantages are only substantial when the change is made at analysis-, design- and implementation levels. Second, it is unclear how these advantages materialize: possibilities include that changes can be implemented faster, that fewer errors are committed or that the resulting software is of a higher quality. Third, it is hard to determine for which types of changes advantages exist.

We conduct two controlled experiments in order to clarify the abovementioned claims. The experiments investigate the influence of abstraction on evolvability under different types of change (from simple to drastic changes). Our results confirm the claims in literature that there are evolvability advantages associated with abstract models, in terms of time needed to implement a change to a conceptual model. However, this is only true for certain types of changes. For other types, there are evolvability disadvantages. Additionally, there are disadvantages in terms of correctness: abstract models are changed in a less correct way than concrete models. The main contribution of this paper is to provide insight into how the rather vague claims in literature should be interpreted.

This paper is structured as follows. In Section 2, the literature related to this study is summarized. In Section 3, we discuss how we define abstraction and genericity, provide an example, explain the relationship between expertise and abstraction, and we discuss abstraction as a principle underlying several research areas in software engineering. Based on our definitions of abstraction and genericity in this section, we explain two ways in which abstraction can influence evolvability of conceptual models according to the literature. In Section 4, research questions and hypotheses are formulated. In Sections 5 and 6, we discuss the experiments we conducted. Section 7 discusses the results of the experiments in detail. Finally, validity threats are identified and conclusions are presented.

2. Related Work

The large amount of claims in software engineering literature about evolvability advantages of abstract or generic conceptual models contrasts sharply with the relatively limited amount of empirical research on software maintenance. Kemerer

(1995) investigated 3 leading journals and the proceedings of 2 conferences (IEEE International Conference on Software Engineering and the IEEE Conference on Software Maintenance) over a period of 10 years. He concluded that only about 2% of the publications were empirical research on maintenance or maintainability. Moreover, the available empirical studies mostly focus on the implementation of information systems. However, there is also a substantial amount of research on evolvability of designs (see, for example, Arisholm and Sjöberg, 2004; Briand et al., 2001; Briand et al., 1997).

Interest in studying evolvability of conceptual models has been growing recently (see, for example, Genero et al., 2003; Genero et al., 2004). However, very little of this empirical research deals with the influence of the abstraction level on the evolvability of conceptual models. One exception is Larsen and Naumann (1992) who conducted experiments to investigate the impact of concrete or abstract terminology in Data Flow Diagrams (DFD's) on requirements determination tasks. Their subjects were university level faculty who completed a graduate level course in systems analysis and were therefore familiar with DFD's, but were not experienced practitioners. The subjects were given a concrete or abstract DFD and were instructed to complete the DFD by asking questions to an end user. Next, the subjects were asked to modify their DFD's using the acquired information. Larsen and Naumann conclude from their experiments that subjects commit more errors when changing abstract models. They did not find significant differences in the time subjects needed to make the changes, but they claim this was due to an artificial time limit in the experiment.

A second relevant study is Prechelt et al. (2001). Their experiments compare the maintenance characteristics of design patterns (which are abstract models) to simpler alternatives (which were more concrete models). Their maintenance tasks show mostly positive results from using design patterns where the inherent flexibility of the pattern is achieved without requiring more maintenance time or maintenance time was reduced compared to the simpler alternatives. In a few cases they found negative effects: the design pattern was more error-prone or required more maintenance time. They conclude that, unless there is a clear reason to prefer the simple solution, it is probably wise to choose the flexibility provided by the design pattern. Prechelt et al. used professional software engineers as subjects. Our study is related to Prechelt et al., but is situated at the analysis level as opposed to implementation and design; moreover we use novices as subjects to avoid the confounding effect of expertise and use different ways of measuring constructs such as correctness and structure degradation.

Although the amount of studies specifically aimed at evolvability of conceptual models is limited, we argue that a number of reasons exist for studying this topic. First, conceptual models are the artefact closest to the origin of a change, i.e., the end user in an organisation. Therefore, conceptual models are often the first artefact to which a change is or should be applied. Design and implementation artefacts are then updated accordingly. Wand and Weber (2002) similarly emphasize the importance of research in conceptual modeling as these models facilitate early detection of development problems. They specifically point to business process reengineering and enterprise resource planning systems as areas where high-quality conceptual modeling can make significant contributions. In their research agenda, this paper would fit in the conceptual modeling

scripts category, where they observe a paucity of research. Second, the literature proposes a number of quality frameworks for conceptual models, indicating desirable qualities. Several of these frameworks refer to evolvability, extensibility, flexibility and stability as important characteristics (Assenova and Johannesson, 1996; Kesh, 1995; Levitin and Redman, 1995; Simsion, 1994; Batini et al., 1992; Roman, 1985; Moody and Shanks, 1994). However, these frameworks do not provide quantitative measures for these characteristics. Recently, measures for modifiability have been proposed by Genero et al. (2004). Our paper is a different but related attempt to measure the evolvability characteristic in these frameworks. Third, several metrics for requirements specifications have been proposed that attempt to measure aspects of evolvability (Davis et al., 1993; Costello and Liu, 1995). Fourth, Marche (1991, 1993) empirically investigates the stability of ER models of 7 information systems. His research indicates that these models were kept stable in artificial ways, i.e., changes in the requirements were implemented by implicitly changing the definitions of entities and attributes instead of changing the ER model itself. This artificial stability is probably suboptimal because the models do not have a 1-to-1 correspondence with the real world anymore. Marche concludes that “there should be greater concentration on the question of data model evolvability, and on the appropriate preservation of meaning across model versions, rather than on data model stability.” Fifth, several authors point out that there is a lack of guidelines for building high-quality conceptual models. Liu (1995) and Rolland and Cauvet (1992) observe specifically that there are surprisingly few guidelines regarding maintainability and robustness of conceptual models. Rosson and Alpert (1990) claim similarly that there is limited consensus on the factors that affect long-term evolvability at the level of a software design. Also, there are no clear guidelines on how objects in a Universe of Discourse (UoD)¹ should be identified, which is a crucial task in building conceptual models (Parsons and Wand, 1997; Wand et al., 1995). Studying evolvability at the level of conceptual models can lead to new and useful guidelines. Sixth, research areas such as analysis and design patterns, application frameworks and software architectures frequently make claims about improving the evolvability of information systems. These research areas are situated at analysis or high-level design and can therefore be interpreted as proposing parts of or guidelines for building conceptual models. Therefore, their claims about evolvability clearly illustrate the relevance of studying evolvability of conceptual models. Finally, current developments on code generation suggest long-term relevance for evolvability studies on conceptual models. Over the last 3 decades, the abstraction level at which software is built, has risen substantially: it has not only risen from implementation over design to analysis; there is also the recent interest in code generation frameworks such as the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG). Yeh (1990) discusses “an alternative paradigm for software evolution” in which software is maintained at the highest level of description rather than at the code level. In this paradigm, it seems likely that the factors that influence evolvability on the implementation level, such as naming of variables and badly structured program code, will become less relevant. Hence, the evolvability of information systems would be more and more determined by the evolvability of its conceptual models.

3. Abstraction/Genericity

3.1. Definition

In the conceptual modelling literature, abstraction is often associated with specific techniques such as inheritance (super- and subclasses) or meta-modelling (see below). However, abstraction is much broader than just these techniques. We define abstraction based on the concepts of genericity and validity. Parnas (1979) defines genericity as: “software can be considered ‘general’ if it can be used, without change, in a variety of situations.” IEEE (1994) defines genericity as “the degree to which a system or component performs a broad range of functions.” Rowe et al. (1998) similarly defines generality as “accommodating change within existing architectural design and implementation.” Therefore, genericity refers to the semantic validity of a model: a generic model is valid in more situations (or UoD’s) than a more concrete one.² Abstraction is defined as “a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information” (IEEE, 1994). For our purposes, we consider genericity and abstraction as very strongly related.

3.2. Example

Figures 1 and 2 show examples of concrete (model A) and more generic and abstract (model B) ER models.³ In Figure 1, the definition of discount is rather different in these 2 models: in model A a discount is defined as a fixed amount that can be deducted from a sale. In model B a discount is a fixed amount that can be deducted in case of different actions (a sale, a rent, a demo. . .). It is clear that according to Parnas’ definition, model B is more generic than model A as it does not have to be modified to incorporate for example a discount on a demo. Model B is valid for both types of discounts, whereas model A is semantically only valid for the sale. Figure 2 contains an example of meta-modelling (Hay, 1995; Van der Sanden et al., 1997). Again, adding a new type of employee requires the concrete model A to be modified, but not the more abstract model B.

3.3. Abstraction/Genericity as a Software Engineering Principle

Abstraction or genericity as we defined it in the previous section is a fundamental principle of several research areas in software engineering such as reuse, analysis and design patterns, software architectures and application frameworks. In our interpretation, all these domains are based on abstract models of UoD’s and software. To support this interpretation, we will briefly discuss each area in this section, including the role of abstract or generic conceptual models in each of them.

Reuse is one of the most important aims in software engineering. Both reuse of software processes and software products is possible, both domain-specific and domain independent (Mili et al., 1995). For our purposes, reuse of software products is the most relevant. At the analysis level, reuse of analysis patterns has been documented. At the design level, design patterns, application frameworks and software architectures have

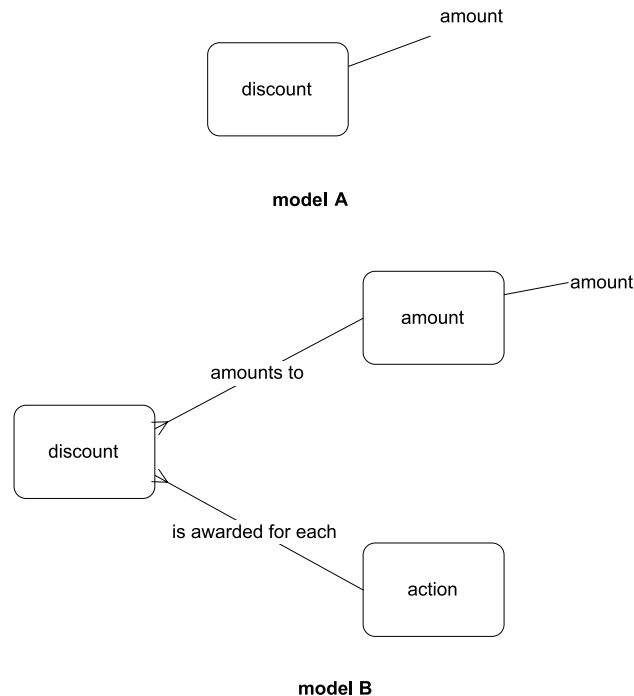


Figure 1. An example of a concrete (model A) and more abstract entity-relationship model (model B).

been proposed. At the implementation level, entire software libraries of reusable program code are commonly available.

All these types of product reuse at these different levels are fundamentally dependent on the characteristic of genericity, i.e., the fact that they are valid in a large amount of situations. This is illustrated by Prieto-Diaz (1990), who claims that “a more dramatic improvement of the reuse process results when we succeed, through domain analysis in deriving common architectures, generic models or specialized languages that substantially leverage the software development process in a specific problem area.”

Software architectures are high-level design models. They define the high-level structure of software in terms of components and the communication between these components. Shaw and Garlan (1996) define a number of architectural styles, which can be instantiated for building a specific piece of software. These architectural styles are inherently meant to be reused, which implies that they have to be relatively generic. The role of abstraction or genericity in software architectures is further illustrated by Tracz (1995), who defines domain-specific software architectures as “an assemblage of software components, specialized for a particular task (domain), generalized for effective use across that domain, composed in a standardized structure (topology) effective for building successful applications.”

Schmid (1997) defines an application framework as “a generic application that allows the creation of different applications from an application (sub) domain.” Both Schmid

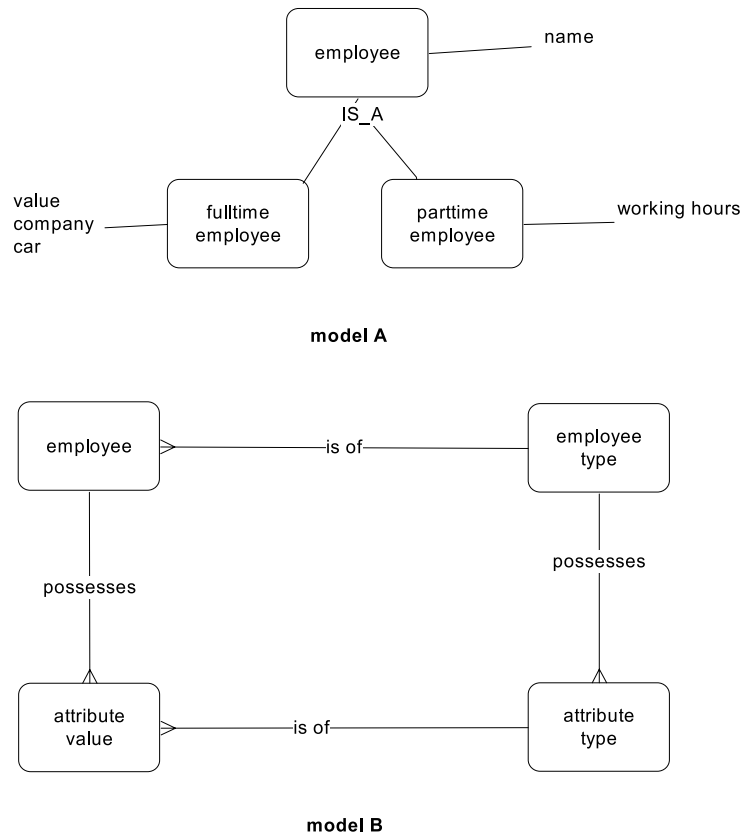


Figure 2. An example of a concrete model (model A) and an abstract model using meta-modelling (model B).

(1997) and Codenie et al. (1997) state that frameworks are a form of consolidated domain knowledge, that was acquired in previous projects.

Simsion (1994) distinguishes between 2 kinds of patterns: standard structures and generic models, defining the latter as “a model covering a complete business or major business activity, usually at a high level of generalization. We might, for example, have a generic accounting or asset management model.”

Finally, it is considered good software development practice to take into account future changes that an information system should be able to adapt to. Rettig (1993), for example, states that “part of the process of programming is to anticipate the kinds of things users will need to know about objects and will want to do with them.” Lubars et al. (1992) add that anticipating changes is also important for building conceptual models; this practice according to them should increase the evolvability of the conceptual model. Ecklund et al. (1996) provides a useful tool to this extent: he proposes to model likely future changes in terms of “change cases.”

Anticipating future changes can be done in several ways. One strategy is to completely include the anticipated functionality in the conceptual model or the information system.

This clearly implies an increase in the level of genericity or abstraction of the conceptual model or information system, because it is valid not only for the set of current requirements but in addition for a set of future requirements. This strategy is typical for application packages such as Enterprise Resource Planning systems. These systems contain an extensive set of functionality; parameters are used to activate the necessary functionality. A second strategy can be found in the literature on software patterns and application frameworks. Schmid (1997) proposes to include so-called “hot spots” at places in the framework where future changes are likely. These hot spots are described as superclasses. These superclasses allow easy implementation of changes: a change in the requirements will be implemented by adding a subclass to the superclass. Also in this case, the application framework and the way in which anticipated changes are dealt with, can be considered generic.

3.4. Abstraction and Expertise

The nature of expertise is extensively studied in cognitive psychology. Expertise can be defined as “a high level of skill or ability resulting from extensive experience and a large body of domain specific knowledge” (Storey et al., 1995). Johnson et al. (1987) explicitly mention the generic character in their definition of expertise, “a kind or operative knowledge” characterized by “generality or the ability to act in new situations, and by power, or the capacity to achieve problem solutions”.

In the context of software development, expertise consists both of problem-solving knowledge, i.e., knowledge of the methodologies and modelling notations for conceptual modelling, and domain-specific knowledge (Storey et al., 1995). Vessey and Conger (1993) indeed conclude from their experiments that both types of knowledge contribute to specifying information requirements, and that interaction effects exist.

It is in the domain-specific knowledge that the relationship between abstraction and expertise for our purposes is most relevant. The role of domain-specific knowledge in conceptual modelling has been investigated by several authors. Batra and Davis (1992) compare the behaviour of 5 experts and 4 novices in conceptual database design. They note that experts spend more time building a holistic understanding of the problem before building the conceptual model. They use domain-related experience to understand the requirements and ask for clarifications. Additionally, they observed that experts were able to quickly relate parts of the problem to knowledge they already possessed, i.e., they reused chunks that they already knew from experience. These chunks have a generic character according to Shanks (1997).

Shanks (1997) concludes that ER models of experts are more innovative and flexible. Innovative is defined as the extent to which “new concepts”, i.e., concepts that were not literally described in the description of the UoD, featured in the ER models. Shanks claims that this is due to reuse of previous experience. Novices build ER models that are more literal translations of the description of the UoD. Flexibility is defined as the ease with which changes in the UoD can be implemented without changing the ER model. Shanks concludes that the experts build more flexible models because they use more generic concepts and because of the modelling of business rules as instances of entities,

making the ER model more abstract. Kao and Archer (1997) investigate the influence of domain knowledge on conceptual model design. They conclude that domain experts build models with more high-level concepts, and that these high-level concepts tended to guide the entire design. Similar use of domain knowledge at the design level has been observed by Jeffries et al. (1981), Adelson and Soloway (1985) and Guindon (1990).

3.5. Abstraction and Its Influence on Evolvability

On the one hand, the academic literature contains numerous claims about the positive influence of abstraction on evolvability (see for example Garlan and Perry (1995) for claims about evolvability advantages of software architectures; Fayad and Schmidt (1997) and Schmid (1997) for frameworks; Prechelt et al. (2001), Helm (1995) and Cline (1996) for patterns). Typically, most of these claims are very vague, such as in Helm (1995) who introduces patterns as “idiomatic and recurring patterns and structures of communicating objects that solve particular design problems and make designs more flexible, elegant and ultimately reusable.” About the vagueness: first, it is unclear if these advantages are situated at the analysis level. For example, it is possible that these advantages are only substantial when the change is made at analysis-, design- and implementation levels. Second, it is unclear how these advantages materialize: possibilities include that changes can be implemented faster, that fewer errors are committed or that the resulting software is of a higher quality. Third, it is hard to determine for which types of changes advantages exist.

On the other hand, the literature contains many claims concerning the difficulty of comprehending abstract models or software (see for example, Barker (1989), Veldwijk et al. (1994), Shanks (1997)). Additionally, Fayad and Schmidt (1997) claim that validating changes to generic components is more difficult than to concrete ones, because the changes to generic components have to be validated for all possible instances. Dikel et al. (1997) warn that building in too many anticipated changes increases the complexity of software, so that it becomes increasingly difficult to maintain.

To our knowledge, there are only two empirical studies that address the influence of abstraction on evolvability, i.e., Larsen and Naumann (1992) and Prechelt et al. (2001). We summarized their findings in Section 2.

4. Research Question and Hypotheses

Our goal is to gain insight into the claims in software engineering literature about the effect of abstraction or genericity on the evolvability of conceptual models. Our research question can therefore be stated as follows:

What is the influence of the level of abstraction of a conceptual model on the evolvability of the conceptual model?

As described in the introduction, we are interested in studying the ease with which information systems can be adapted to the changing requirements of an organization and

its end users. We argued that this type of changes has high empirical relevance in today's increasingly volatile environments in which information systems function. In such environments, a high volume of perfective and adaptive maintenance can be expected. In this study, we limit evolvability to perfective maintenance-i.e., changes expanding the existing functional requirements of the information system; corrective, preventive and adaptive maintenance tasks (in the sense of changes implied by moving the software to a different hardware or software platform) are excluded (for definitions, see Lientz and Swanson (1980), Takang and Grubb (1996)). We only consider evolvability of the conceptual model; we do not include the evolvability of the information system as a whole (i.e., including designs and program code).

We are interested in studying whether the influence of abstraction on evolvability depends on the types of change in the UoD. From a theoretical point of view, it would be ideal if an empirically-based typology would be available and a representative set of changes could be selected for controlled experiments. However, the question is whether this is feasible; in any case, an extensive literature survey revealed no such typology. Therefore, we decided on the following approach: a UoD consists of concepts and properties. A change is defined in terms of the concepts and properties in the UoD (additions, modifications and deletions). We included the different types of changes in our study in the independent variable "complexity of change." We explain in following sections how the changes were made operational.

From a theoretical perspective, evolvability has at least three aspects: time, structure degradation and correctness.

Time needed to implement a change is used in most scientific experiments on evolvability. We differentiate between two separate time measurements: first, time to read the UoD and the description of the required change; second, time to implement the change to the conceptual models. To investigate the difference between concrete and abstract models of the same UoD, only the second is relevant and will be reported in our discussion.

Only measuring time needed, provides a limited measure for evolvability. Structure degradation refers to different ways in which a certain change can be applied to a conceptual model. When a change is applied as a "quick fix," the time used to apply the change is low, but future preventive maintenance is more likely to be needed. When a change is applied in a "structurally correct" way, the change does not cause significant preventive maintenance. It is even possible that a certain change to the model improves the structure so that preventive maintenance is less likely to be needed in the future.⁴ We define structure degradation as the difference in evolvability between the initial and the changed conceptual model. If a certain change is applied to an abstract and a concrete conceptual model in the same amount of time, but with different structure degradation (and therefore, with different future preventive maintenance), there is *ceteris paribus* a difference in the evolvability of the two conceptual models. Therefore, structure degradation is a relevant aspect of evolvability. Constructs such as structure degradation and quick fixes are not defined in a precise, formal way in literature (Harrison, 1996). There are some indications of how they should be measured at the design and implementation level, but not at the analysis level. Few, if any, experiments measure structure degradation. We used a human expert to rate structure degradation on an 11-point scale and thereby investigate the feasibility of this approach.

Most experiments measure the number of syntactic errors, i.e., errors against the modelling formalism in use. For our purposes, these are of secondary importance. We define correctness in terms of the UoD and the description of the change. More specifically, our definition of correctness emphasizes that each concept in the UoD should be represented in the conceptual model in a semantically correct way. Again, we used a human expert to rate correctness on a 7-point scale.

The hypotheses tested are (see Figure 3):

H1a: the **abstraction level of a conceptual model** influences the **time** needed to apply a change to the conceptual model

H2a: an interaction effect exists of the **abstraction level of a conceptual model** and the **complexity of a change** on the **time** needed to apply a change to the conceptual model

H1b: the **abstraction level of a conceptual model** influences the **correctness** with which a change is applied to the conceptual model

H2b: an interaction effect exists of the **abstraction level of a conceptual model** and the **complexity of a change** on the **correctness** with which a change is applied to the conceptual model

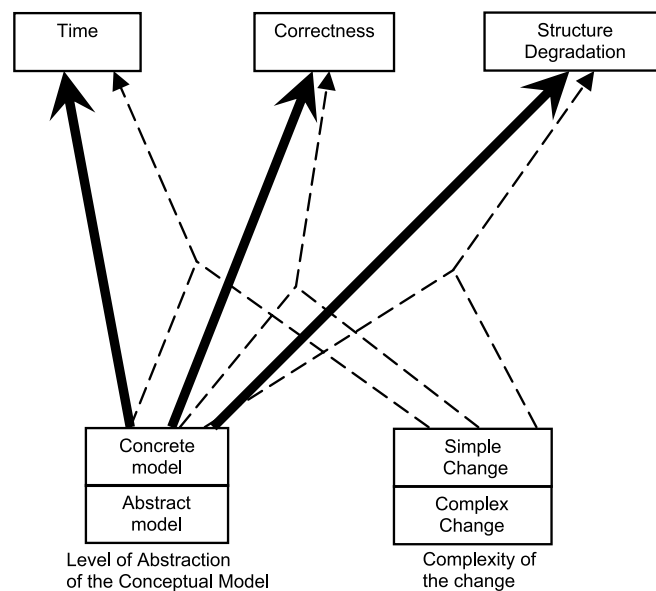


Figure 3. Summary of the tested hypotheses: main effects are indicated in bold; interaction effects in dashed lines.

H1c: the **abstraction level of a conceptual model** influences the **structure degradation** of the conceptual model caused by applying the change

H2c: an interaction effect exists of the **abstraction level of a conceptual model** and the **complexity of a change** on the **structure degradation** of the conceptual model caused by applying the change.

5. Experiment 1

5.1. Subjects

In experiment 1, we used 74 students following an Information Systems Analysis course, which is an exceptionally large number for this kind of experiments. The students were in the fourth year of the MIS-specialisation track of a Business Administration degree at the University of Antwerp. The students had received 15 hours training in Entity Relationship models as a start to database design. The training consisted mainly of supervised exercises. Attention was paid not to bias the students towards abstract or concrete models. They also had knowledge of structured programming techniques.

It is common to use students instead of professional analysts for this kind of experiments (see, for example, Briand et al. (1997), Daly et al. (1996), Abbatista et al. (1994)). For our purposes, advantages are that:

- the literature indicates that abstraction and expertise are related. Batra and Davis (1992), for example, find that experts use domain knowledge to understand conceptual models. By using students, we study the influence of abstraction on evolvability without the confounding effects of expertise. In other words, we can study the true “cognitive effort” required to deal with abstract models.
- the prior knowledge of the students is rather homogeneous.
- the availability of a large number of subjects.

5.2. Experimental Design

We used a 2X2 factorial design, with abstraction level as between-subjects factor and complexity of the change as within-subjects factor.

Abstraction level was treated as a between-subjects factor in order to avoid learning effects. Learning effects can occur when a subject applies a change to a concrete and abstract model of the same UoD. It is possible that subjects will copy parts of the abstract model to the concrete model. Even when confronted with different UoD's, it is possible that there is a bias: seeing an abstract model can influence the way in which a subject will apply the change in a concrete model. To neutralize these learn-

Table 1. Overview of the experimental design and allocation of the experimental groups to the experimental tasks.

Abstraction level of the CM (between-subjects)	Concrete CM	Abstract CM
Complexity of change (within-subjects)		
Simple change	Group A (1) N = 9	Group E (1) N = 9
Human resources UoD	Group B (2) N = 8	Group F (2) N = 9
Complex change	Group A (2) N = 9	Group E (2) N = 9
Transport UoD	Group B (1) N = 9	Group F (1) N = 9
Simple change	Group C (2) N = 9	Group G (2) N = 10
Transport UoD	Group D (1) N = 9	Group H (1) N = 10
Complex change	Group C (1) N = 9	Group G (1) N = 10
Human resources UoD	Group D (2) N = 9	Group H (2) N = 10

ing effects, we used different groups of subjects for the concrete and abstract conceptual models.

Subjects were confronted with both levels of the second independent variable, complexity of the change. Each subject applied both a simple and a complex change. This allowed us to increase the number of observations from the available subjects.

Table 1 illustrates the allocation of the subjects to the experimental tasks. Subjects were divided up into 8 groups, labelled A to H. Each group consisted of 8 to 10 subjects. The order in which subjects from a group received the experimental tasks is indicated in brackets. To neutralize learning effects, the sequence of the changes was counter-balanced: one subject applied a simple change to UoD1 and a complex one to UoD2; the next subject did the same tasks in different order. Also, the UoD used in the experimental task is indicated (for details, see next paragraph).

5.3. *Experimental Task*

Two UoD's were used: a Human Resources (HR) UoD about hiring, promoting and firing employees and a Transport UoD about selling and renting cars and trucks. The chosen UoD's were by no means unusual or exceptional. They were also sufficiently different from any UoD's the students had seen before in their university courses.

As discussed before, the changes were defined in terms of concepts and properties that were added, edited or deleted in the UoD. However, there is no precise way of measuring if there is a difference in adding, for example, a new type of customer or a new type of invoice, so it is difficult to argue that one would be a more complex change than the other. Therefore, we operationalized the difference between simple and complex in terms of the number of additions and deletions to the UoD. The simple change consisted of 2–3 new properties that were added to the UoD. The complex change consisted of all the

additions in the simple change, plus a number of additional ones (both concepts and properties). So, for this experiment, the difference between the simple and complex change lies in the number of additions. The additions themselves were not very complicated.

The conceptual models consisted of 3 parts: an Entity Relationship model, a definition of the entities with some examples of instances and 1 to 3 procedures. The model, the definitions and the procedures each fitted on one page. We opted for the Entity Relationship notation instead of UML because of the prior knowledge of the students. The procedures were written in pseudocode.

The conceptual models of the HR UoD consisted of about 7 and 8 entities; the models of the Transport UoD of 8 entities (concrete model) and 15 entities (abstract model). The abstract models contained much meta-modelling (as in Figure 2). Also the corresponding procedures/functions were typical for meta-modelling. Meta-modelling was explained to the subjects in the lectures before the experiments.⁵

5.4. *Experimental Procedure*

The experiment started with a 15-minute introduction in which the steps of the experiment were explained. The students were then divided up according to their study orientation, because their study orientation influenced their prior knowledge. Next, the students of each orientation were randomly distributed over the experimental groups. The experiment was carried out in 4 steps: in the first step each subject received a natural language description of the UoD with a change. Subjects were asked to read this page carefully, after which they had to signal a supervisor. At this point, the supervisor wrote down the time used for reading the UoD. In the second step, the subject received the conceptual model of the UoD. When subject had applied the change, the supervisor was signalled again and a second time measure was taken. Next, the subject received the natural language description of the second, different UoD with a change (step 3). After carefully reading this, and signalling the supervisor, the subject received the conceptual model (step 4). The subjects had more than 2 hours time; no subject reached this time limit.

5.5. *Ratings by Experts*

Each subject's solution was rated by 5 experts on correctness and structure degradation. The 5 experts were practitioners with minimum 2 years experience as a professional information systems analyst from 3 different and large software companies. We preferred practitioners over academic experts as practitioners should have more relevant empirical experience in terms of how future evolvability is impacted in a certain model. Raters from the same company were explicitly asked not to communicate with each other; all of them confirmed this after the experiment. Every expert rated all observations in the experiment. As this was quite time consuming, the experts were paid 250 euro for their collaboration to improve their motivation. The experts were not given a "correct" solution. In their written instructions, they received the following definition of

correctness: “Correctness refers to the extent in which the changed CM is a correct representation of the changed UoD.” Additionally, in accordance with this definition, we explicitly asked not to focus on syntactical correctness. Correctness was rated on a 7-point scale.

For structure degradation, their written instructions contained the definition discussed above, i.e., that the way in which a change is applied to the CM can influence the future evolvability in a positive or negative way. They were asked to rate structure degradation on an 11-point scale, from “−5: future evolvability has decreased to a large extent” over “0: future evolvability was not influenced” to “+5: future evolvability has increased to a large extent.”

5.6. Results and Analysis

5.6.1. Anomalies

One subject in group B only completed his first experimental task. The second task was by mistake of the supervisor not presented to the subject. The completed experimental task was included in the analysis where possible.

5.6.2. Time

Table 2 shows that simple changes were applied faster to the concrete model than to the abstract model. This is interesting because implementing the change required additions of entities and relationships to the concrete model whereas the abstract model merely required changes in the instances. In previous sections, we pointed out that for these changes, many authors expect evolvability advantages from abstract implementations of information systems, taking into account changing the models and the implementations (program code, databases...). Here we observe that, if only the conceptual model is considered and for small changes, these advantages do not exist; on the contrary. The reason could be disadvantages in comprehensibility. We observe that for these very simple changes, subjects still need on average 14 to 16 minutes (excluding reading the UoD and the description of the change). Novices possibly use the majority of this time to

Table 2. Experiment 1: Descriptive statistics for time.

		Abstraction level	
		Concrete	Abstract
Simple change	Avg.	0:14:06	0:16:46
	St. dev.	0:05:55	0:08:11
	N	35	38
Complex change	Avg.	0:23:08	0:19:57
	St. dev.	0:08:52	0:08:38
	N	35	38

Table 3. Experiment 1: ANOVA results for time.

Source	Variation (Sum of squares-SS)	df	Variance (mean square)	F	Probability
Between-subjects					
Abstraction level	8877.899	1	8877.899	0.045	0.83
Error	13993297	71	197088.696		
Within-subjects					
Complexity	4891369.7	1	4891369.7	18.456	0.00*
Abstraction \times Complexity	1120147.8	1	1120147.8	4.227	0.04*
Error	18816628	71	265022.923		

* $p < 0.05$.

understand the model, and, in case of such a simple change, only a limited portion actually modifying the model. In the case of simple changes by novices, it seems that the disadvantages such as comprehensibility are bigger than the advantages of adding instances as compared to adding entities and relationships.

As mentioned before, the complex change consisted of all the changes to the UoD in the simple change, plus a number of additional ones. This means that an even larger number of changes in the UoD could be implemented as changes to instances in the abstract model that required additional entities, relationships and attributes in the concrete model. Therefore, the potential for evolvability advantages for the abstract models is even larger in this case, than in the case of simple changes.

Table 2 shows that for complex changes, abstract models are indeed modified quicker than concrete models. But the magnitude of the time advantages is in our opinion modest with respect to the total time spent implementing the change, i.e., changing a number of instances by novices requires on average about 20% less effort than changing entities, relationships and attributes. Again, this could be due to novices using a large part of their time understanding the model, which makes the evolvability advantages relative to total time spent on the modification, rather small. But for complex changes, the disadvantages are smaller than the evolvability advantages.

Repeated measures ANOVA was used to determine the statistical significance of these effects. Table 3 shows the interaction effect is indeed significant ($p < 0.05$).⁶

Table 4. Experiment 1: Inter-rater reliability for correctness.

	Cronbach's α
Simple change	0.82
Simple change/concrete	0.75
Simple change/abstract	0.86
Complex change	0.82
Complex change/concrete	0.85
Complex change/abstract	0.82

Table 5. Experiment 1: Descriptive statistics for correctness.

		Abstraction level	
		Concrete	Abstract
Simple change	Avg.	4.53	4.00
	St. dev.	1.0414	1.4807
	N	35	38
Complex change	Avg.	4.72	4.26
	St. dev.	0.9979	1.2171
	N	35	38

5.6.3. Correctness

All observations were rated by 5 human experts on correctness. First, we check the inter-rater reliability to determine how consistent the results of the raters were with each other. For this purpose, we use Cronbach's α -coefficient (Cronbach, 1951). This coefficient is frequently used in the IS literature for these purposes (see for example Shanks (1997)).

Results of 0.7 and above are considered reliable to very reliable. Therefore, we can conclude from Table 4 that the correctness ratings are sufficiently reliable to justify further analysis.

Table 5 shows that the results for all 4 cells in the experimental design are very close to the middle of the rating scale: ratings vary between 4.0 and 4.72. Interestingly, the averages for correctness are higher for the complex change than for the simple change. Also, the subjects should have been able to apply these changes correctly, so we expected a higher level of correctness, certainly for the simple changes. A possible explanation is that the experimental tasks were slightly larger but not otherwise more difficult than the models used during pre-experiment training. This size was, however, necessary to be able to define the complex change as a change to more concepts in the UoD than the simple change. A second explanation could be that we did not provide detailed guidelines to the raters regarding what was to be considered "very correct." This may have led them to score more towards the middle of the scale.

Table 6. Experiment 1: ANOVA results for correctness.

Source	Variation (Sum of Squares-SS)	df	Variance (mean square)	F	Probability
Between-subjects					
Abstraction level	8.897	1	8.897	4.072	0.04*
Error	155.138	71	2.185		
Within-subjects					
Complexity	1.859	1	1.859	2.562	0.11
Abstraction \times Complexity	5.068 ^E - 02	1	5.068 ^E - 02	0.070	0.79
Error	51.522	71	0.726		

*p < 0.05.

Table 7. Experiment 1: Inter-rater reliability results for structure degradation.

	Cronbach's α
Simple change	0.85
Simple change/concrete	0.88
Simple change/abstract	0.87
Complex change	0.72
Complex change/concrete	0.86
Complex change/abstract	0.65

Concrete models were modified more correctly than abstract ones, both in the case of simple and complex changes. Table 6 shows that the main effect of abstraction level on evolvability is statistically significant. This result is consistent with the findings of Larsen and Naumann (1992) in their experiments with Data Flow Diagrams: also their subjects committed more errors on the abstract models. A possible explanation could be the comprehension difficulties of the abstract models, but no direct evidence to support this was available from this experiment.

5.6.4. Structure Degradation

All observations were rated by 5 human experts on structure degradation. Table 7 shows that the inter-rater reliability results for the simple changes are both above 0.8 and are therefore very good. The coefficient for complex change to the abstract model is slightly below par (0.65). This was caused by the results of one rater; when this rater is excluded from the Cronbach's α calculations, the reliability rises to 0.86. We conclude therefore that the ratings are sufficiently consistent to justify further analysis.

The average score for all 4 cells in the experimental design lies between -0.5 and -1 (see Table 8). This means that, on average, little structure degradation was incurred. This seems logical from the point of view that the changes in the experiment did not require the model to be fundamentally changed. All changes could be applied by adding a limited number of instances, entities, relationships or attributes to the model. It is possible that more drastic changes would induce more structure degradation.

Table 8. Experiment 1: Descriptive statistics for structure degradation.

		Abstraction level	
		Concrete	Abstract
Simple change	Avg.	-0.53	-0.97
	St. dev.	1.0105	1.1965
	N	35	38
Complex change	Avg.	-0.75	-0.83
	St. dev.	0.9629	0.9189
	N	35	38

Table 9. Experiment 1: ANOVA results for structure degradation.

Source	Variation (Sum of squares-SS)	df	Variance (mean square)	F	Probability
Between-subjects					
Abstraction level	2.459	1	2.459	1.533	0.22
Error	113.911	71	1.604		
Within-subjects					
Complexity	5.940E-02	1	5.940 ^E -02	0.115	0.74
Abstraction × Complexity	1.213	1	1.213	2.357	0.13
Error	36.547	71	0.515		

*p < 0.05.

Second, the results indicate that the abstract models incurred more structure degradation than the concrete models, both for simple and complex changes. Table 9 shows, however, that this effect is not statistically significant.

6. Experiment 2

A second experiment was conducted for 2 reasons: first, to test different kinds of changes; second, to confirm the results from the first experiment to improve reliability.

The first change was much more complex than the ones in experiment 1 (a “drastic” change); this drastic change invalidated large parts both of the concrete and the abstract conceptual model. The second was the complex change already used in experiment 1. The design of the experiment was the same; also the same UoD’s were used. 62 new subjects were used with the same background as in the first experiment.

6.1. Time

The descriptive statistics in Table 10 for the complex change confirm the results of experiment 1: the abstract model needed again less time to be modified than the concrete one. The subjects needed considerably more time, though, to complete the two experimental tasks than in the first experiment.

Table 10. Experiment 2: Descriptive statistics for time.

		Abstraction level	
		Concrete	Abstract
Complex change	Avg.	0:31:15	0:24:08
	St. dev.	0:12:33	0:10:53
	N	31	31
Drastic change	Avg.	0:42:35	0:47:05
	St. dev.	0:14:30	0:13:30
	N	31	31

Table 11. Experiment 2: ANOVA results for time.

Source	Variation (Sum of Squares-SS)	df	Variance (mean square)	F	Probability
Between-subjects					
Abstraction level	190480.645	1	190480	0.27	0.60
Error	41759158.065	60	695985		
Within-subjects					
Complexity	32805551.613	1	32805551	64.4	0.00*
Abstraction \times Complexity	3762580.645	1	3762580	7.39	0.00*
Error	30521467.742	60	508691		

*p < 0.05.

Interestingly, the results of the drastic change are in the opposite direction: the abstract model needed more time than the concrete one. It should be noted that the drastic change could not be implemented by merely changing instances in the abstract model; entities and relationships had to be added or modified. Apparently, this was more difficult for abstract than concrete models. Table 11 shows that the interaction effect is statistically significant.

6.2. Correctness

All results were rated by 3 of the 5 human experts used in the first experiment; each expert was from a different software company. They were again paid € 250 for their cooperation.

As in the first experiment, the inter-rater reliability for the complex change was good; Cronbach's α coefficients were 0.70 or higher (see Table 12). However, for the drastic change applied to the abstract model, Cronbach's α coefficient dropped to 0.42, which is insufficient. This time, we could not trace the reliability problems down to a specific rater. Results for correctness for the drastic changes to abstract models have to be interpreted with caution.

The descriptive statistics show virtually no difference in the correctness between abstract and concrete models (see Table 13). We note, however, that the correctness of the complex changes was good, but the correctness of the drastic changes was low. In retrospect, although the pre-experiment training should have been sufficient to apply the drastic change correctly, it is fair to say that the drastic changes were a severe test of the capabilities of subjects with their level of pre-experiment training.

Table 12. Experiment 2: Inter-rater reliability for correctness.

	Cronbach's α
Complex change	
Complex change/concrete	0.79
Complex change/abstract	0.70
Drastic change	
Drastic change/concrete	0.76
Drastic change/abstract	0.42

Table 13. Experiment 2: Descriptive statistics for correctness.

		Abstraction level	
		Concrete	Abstract
Complex change	Avg.	5.3333	5.2903
	St. Dev.	0.95839	1.13118
	N	31	31
Drastic change	Avg.	4.1935	3.8280
	St. Dev.	1.13129	1.02513
	N	31	31

6.3. Structure Degradation

The inter-rater reliability results show reliability problems in the case of the drastic and the complex change applied to the concrete models (see Table 14). Therefore, results have to be interpreted with caution. For the abstract models, the descriptive statistics show that some structure degradation was incurred, according to the experts, especially in the case of the drastic change (see Table 15).

7. Discussion

Our results suggest that abstraction does have evolvability advantages at the level of conceptual models, excluding design and implementation. Moreover, it is interesting to note that these advantages are visible even in relatively small conceptual models.

However, advantages in maintenance time are dependent on the type of change. A small, easy change is applied slower to abstract models than to concrete models. The same is true for a very drastic change that invalidates large parts of the existing conceptual model. Interpretations for the drastic change should be made with some caution, as there were inter-rater reliability issues and relatively low correctness. Nevertheless, our result seems consistent with the conclusion of Prechelt et al. (2001). They found in their experiments at design and implementation level that there were circumstances where the

Table 14. Experiment 2: Inter-rater reliability for structure degradation.

	Cronbach's α
Complex change	
Complex change/concrete	0.60
Complex change/abstract	0.85
Drastic change	
Drastic change/concrete	0.23
Drastic change/abstract	0.70

Table 15. Experiment 2: Descriptive statistics for structure degradation.

		Abstraction level	
		Concrete	Abstract
Complex change	Avg.	0.1613	-0.7957
	St. dev.	0.85145	1.19467
	N	31	31
Drastic change	Avg.	-0.2688	-1.9140
	St. dev.	0.73746	1.04693
	N	31	31

additional flexibility of design patterns was not beneficial and the patterns added to the complexity of the model which resulted in increased maintenance times.

For a change consisting of a large number of small, easy changes, the abstract models do perform better. Our confidence in this result is high, as this effect was visible in both experiments. This effect makes abstraction useful in areas where the future changes to the requirements can be predicted to a certain extent. These anticipated changes can then be “built in” in the abstract model using so-called hot spots as suggested by, for example, Schmid (1997). Therefore, this study confirms the practice of using abstraction for evolvability advantages which is already well-known and widespread. It even suggests the usefulness of this practice in the long-term, i.e., in code generation environments based on the Model Driven Architecture (MDA). However, anticipating changes and building them in also has its limitations. First and most obvious, the difficulty is to predict which changes will actually take place in the future. It is interesting to note that Codenie et al. (1997) remark while describing their experiences implementing application frameworks, that customizing their framework for different customers required more flexibility than their hot spots offered. The framework itself had to be adapted more than they expected. Second, whether the advantages are of the same magnitude as at the implementation level, remains to be investigated.

Summarizing, the three types of changes taken together, taking into account the validity issues mentioned in the next paragraph, hint at the relationship between abstraction and evolvability shown in Figure 4.

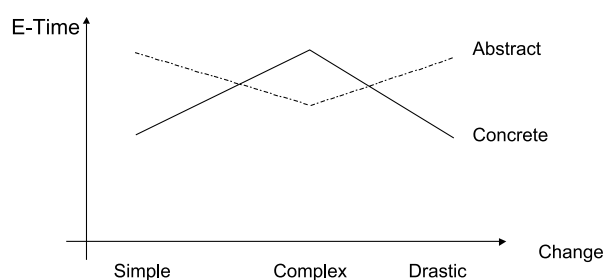


Figure 4. A hypothetical generalization of the relationship between evolvability (time) and type of change, based on our results.

In this context, it is relevant to point out the possible influence of domain knowledge on modification times. The literature on expertise clearly suggests a relationship between abstraction and expertise (see above), but the implications for maintenance tasks remain unknown. We would expect that domain knowledge of professional analysts would reduce the time required to understand a conceptual model and perform impact analysis, thereby making the advantages of abstraction in implementing the change on the conceptual model more visible. One of the few large-scale empirical studies using both novices, intermediates and experts is Arisholm and Sjöberg (2004): they found that the maintainability of object-oriented software depends to a high extent on the level of skill of the developers.

In terms of correctness, our results suggest that abstract models are modified less correctly than concrete models for all types of changes. This is consistent with the results of Larsen and Naumann (1992). It would be interesting to investigate the influence of domain-independent methodological knowledge, i.e., knowledge of conceptual modelling (languages), on correctness, because this type knowledge could improve correctness of modifications to abstract models, specifically in the case of meta-modelling.

The results on time and correctness suggest that there is a certain trade-off between them when using abstraction for evolvability. However, the only correlation between time and correctness that was statistically significant was in the case of the complex change applied to concrete models. On aggregate, the correlation was not significant.

Our results suggest the following considerations for software engineering practice. First, the literature on expertise suggests that during development time, building abstract models requires more knowledge and maybe other resources than concrete models. Building abstract models is no easy-to-use trick for novices; it requires both domain-specific and methodological expertise and probably more time and other resources. During evolution, abstraction seems to have evolvability advantages even for novices for some types of changes, but at a price in terms of correctness. In real-world settings, we expect correctness to be of higher importance than time. When experts are used to perform maintenance on the conceptual models, the evolvability advantages could be more pronounced both in terms of time and correctness, but this again requires additional resources. This seems to require a change in current practices: Briand et al. (2001) remark that programmers with little object-oriented training and experience are commonly found on maintenance projects. Summarizing, abstraction is certainly no panacea for evolvability; it comes only under certain circumstances and it comes at a certain price.

8. Validity

This type of experiments intrinsically has a number of external validity issues. First, the experimental tasks were performed by novices. One of the main reasons why we used novices was that expertise would make the cognitive effort required by abstract models, less visible. Nevertheless, a replication with professional analysts would be interesting. We already discussed the potential influence of experts with methodological and domain-specific knowledge on our results in the previous paragraph. Second, the experiments used relatively small conceptual models. The experimental materials comprised only

3 pages. It is possible that the effects of abstraction are different when using larger, more realistic models. Third, results would probably be very different when also considering design and implementation. For example, our results show that, at the analysis level, some modifications can be applied faster to abstract models than to concrete models. It is very likely that additional time gains can be found at the design and implementation level because abstract models imply different software architectures than concrete models. An example is that changing an instance in a database is much easier than adding an entity type, but at the analysis level, as the experiments have shown, the difference is not that big.

In terms of internal validity, the following issues have to be taken into account. First, selection effects should not have occurred after random assignment of the 136 subjects used, also taking into account their relatively homogeneous background. There are some indications of instrumentation effects: analysis shows that the results we reported are clearly more pronounced in one of the UoD's we used. Although we took great care that the UoD's and the changes contained no unusual or abnormal features, it is useful to replicate our experiments with different UoD's. Finally, as we pointed out before, our approach to measuring correctness and structure degradation using ratings from human experts has reliability issues. Our results indicate that this approach is possible for models to which small changes were applied; in these cases, inter-rater reliability was good to very good. This made it possible to check whether time measurements were influenced by trade-offs with correctness and structure degradation in the first experiment. However, when drastic changes were required to the models, inter-rater reliability was insufficient to use statistical testing on the rating results. This poses problems, especially for measuring structure degradation, where reliable and practical alternative measures are difficult to find.

Table 16. Overview of the hypothesis and experimental results.

	Experiment 1	Experiment 2
H1a: the abstraction level of a conceptual model influences the time needed to apply a change to the conceptual model	Not supported	Not supported
H2a: an interaction effect exists of the abstraction level of a conceptual model and the complexity of a change on the time needed to apply a change to the conceptual model	Supported	Supported
H1b: the abstraction level of a conceptual model influences the correctness with which a change is applied to the conceptual model	Supported	
H2b: an interaction effect exists of the abstraction level of a conceptual model and the complexity of a change on the correctness with which a change is applied to the conceptual model	Not supported	
H1c: the abstraction level of a conceptual model influences the structure degradation of the conceptual model caused by applying the change	Not supported	
H2c: an interaction effect exists of the abstraction level of a conceptual model and the complexity of a change on the structure degradation of the conceptual model caused by applying the change	Not supported	

9. Conclusions and Future Research

Abstraction or genericity is one of the most important mechanisms that literature proposes to improve evolvability of information systems. In this study, we investigate to which extent the claims about evolvability advantages are situated at the analysis level, thereby excluding the design and implementation levels. In other words, we investigate whether abstract conceptual models are better evolvable than concrete ones.

Our results, summarized in Table 16, show that abstraction or genericity is indeed effective in terms of improving evolvability, but only for some types of changes. A small, easy change is implemented slower to abstract models than to concrete models. The same is true for a very drastic change that invalidates large parts of the existing conceptual model. For a change consisting of a set of small, easy changes, the abstract models do perform better. This makes abstraction useful in areas where future changes to the requirements can be predicted to a certain extent. Therefore, this study confirms a practice which is already well-known and widespread at the design and implementation level, anticipating changes and “building them in” using abstraction. However, reservations have to be made in terms of the correctness of changes to abstract conceptual models: results of experiment 1 indicate, consistent with other research, that abstract models are changed less correctly than the concrete ones. Table 16 does not include results for correctness and structure degradation from experiment 2, as the inter-rater reliability for these variables was doubtful.

We conclude that the decision to use abstraction for evolvability should be taken carefully. Abstraction is related to expertise. Building abstract models requires more expertise and probably other resources than concrete ones. Modifying abstract models by novices seems to have advantages for “built in” changes, but for other changes the results are less positive. We expect that experts would be able to exploit the evolvability advantages to a larger extent, but this expertise again increases the resources needed to achieve the benefits.

Many issues concerning evolvability of conceptual models are still open and need to be investigated before we fully understand the relationship between abstraction and evolvability. We already mentioned interesting future research when discussing the role of methodological and domain-specific knowledge. It would also be interesting to replicate these experiments for types of abstract conceptual models not using meta-modelling to ensure that some of the effects are not specific to this technique. Finally, different changes in the UoD generate different results in our experiments. A larger set of changes would further increase insight into the relationship between abstraction and evolvability.

Notes

1. The Universe of Discourse (UoD) is the ‘slice’ of reality to be modelled (Hirschheim, Klein and Lyytinen, 1995), or in other words, the part of reality that is to be represented by a conceptual model.

2. Abstraction and genericity are relative and not absolute: a conceptual model is not abstract or generic in itself, but more abstract or generic than another model. However, for the sake of brevity, the relative formulation is not used in this paper unless needed.
3. The definitions of the entities, as well as the procedures/functions for these examples, have been omitted for the sake of brevity.
4. Strictly speaking, we should use the term "Structure degradation and/or improvement." For the sake of brevity, we will use "Structure degradation."
5. The experimental materials are available on <http://www.ua.ac.be/jan.verelst>.
6. The relevant statistical assumptions about normality and homoscedasticity were checked. No substantial problems were found.

References

- Abbatista, F., Lanubile, F., Mastelloni, G., and Vissagio, G. 1994. An experiment on the effect of design recording on impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pp. 253–259.
- Adelson, B., and Soloway, E. 1985. The role of domain experience in software design. *IEEE Transactions on Software Engineering* 11(11): 1351–1360.
- Arisholm, E., and Sjöberg, D. I. K. 2004. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on software engineering* 30(8): 521–534.
- Assenova, P., and Johannesson, P. 1996. Improving quality in conceptual modelling by the use of schema transformations. In *Proceedings of the International Conference on the Entity Relationship Approach (ER'96)*, pp. 277–291.
- Barker, R. 1989. *CASE*Method: Entity Relationship Modelling*. Addison Wesley.
- Batini, C., Ceri, S., and Navathe, S. B. 1992. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings.
- Batra, D., and Davis, J. G. 1992. Conceptual data modelling in database design: similarities and differences between expert and novice designers. *International Journal of Man-Machine Studies* 37: 83–101.
- Briand, L. C., Bunse, C., Daly, J., and Differding, C. 1997. An experimental comparison of the maintainability of object-oriented and structured design documents. *Journal of Empirical Software Engineering* 2(3): 291–312.
- Briand, L. C., Bunse, C., and Daly, J. W. 2001. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering* 27(6): 513–530.
- Cline, M. P. 1996. The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM* 39(10): 47–49.
- Codenie, W., De Hondt, K., Steyaert, P., and Vercammen, A. 1997. From custom applications to domain-specific frameworks. *Communications of the ACM* 40(10): 71–77.
- Costello, R. J., and Liu, D.-B. 1995. Metrics for requirements engineering. *Journal of Systems and Software* 29: 39–63.
- Cronbach, L. J. 1951. Coefficient alpha and the internal structure of tests. *Psychometrika* 16(3): 297–334.
- Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. 1996. Evaluating inheritance depth on the maintainability of object-oriented software. *Journal of Empirical Software Engineering* 1: 109–132.
- Davis, A. M., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., and Theofanos, M. 1993. Identifying and measuring quality in a software requirements specification. In *Proceedings of the 1st International Software Metrics Symposium*, pp. 141–152.
- Dikel, D., Kane, D., Bass, L., Ornburn, S., Loftus, W., and Wilson, J. 1997. Applying software product-line architecture. *IEEE Computer* 30(8): 49–55.
- Ecklund, E. F., Delcambre, L. M. L., and Freiling, M. J. 1996. Change cases: Use cases that identify future requirements. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pp. 342–358.

- Fayad, M. and Schmidt, D. G. 1997. Object-oriented application frameworks. *Communications of the ACM* 40(10): 32–38.
- Garlan, D. and Perry, D. E. 1995. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering* 21(4): 269–274.
- Genero, M., Poels, G., and Piattini, M. 2003. Defining and validating metrics for assessing the maintainability of entity-relationship diagrams. Working paper 2003/199, University of Ghent (Belgium).
- Genero, M., Piattini, M., and Manso, E. 2004. Finding “early” indicators of UML class diagrams understandability and modifiability. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 207–216.
- Guindon, R. 1990. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies* 33: 279–304.
- Harrison, W. 1996. Change-prone modules, limited resources and maintenance. In *Proceedings of the 1st International Workshop on Empirical Studies of Software Maintenance (WESS'96)*, pp. 145–150.
- Hay, D. C. 1995. *Data Model Patterns: Conventions of Thought*. Dorset House.
- Helm, R. 1995. Patterns in practice. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pp. 337–341.
- Hirschheim, R., Klein, H. K., and Lyytinen, K. 1995. *Information Systems Development and Data Modeling: Conceptual and Philosophical Foundations*. Cambridge University Press.
- IEEE. 1994. IEEE Standard 610.12-1990: IEEE Standard glossary of software engineering terminology. *IEEE Standards Collection: Software Engineering*.
- Jeffries, R., Turner, A. A., Polson, P. G., and Atwood, M. E. 1981. The processes involved in designing software. In J. R. Andersen (ed.), *Cognitive Skills and Their Acquisition*. Erlbaum, pp. 255–283.
- Johnson, P. E., Zulkernan, I., and Garber, S. 1987. Specification of expertise. *International Journal of Man-Machine Studies* 26: 161–181.
- Kao, D., and Archer, N. P. 1997. Abstraction in conceptual model design. *International Journal of Man-Machine Studies* 46: 125–150.
- Kemerer, C. F. 1995. Software complexity and software maintenance: A survey of empirical research, In O. Balci (ed.), *Annals of Software Engineering*. Baltzer Science Publishers, pp. 1–22.
- Kesh, S. 1995. Evaluating the quality of entity relationship models. *Information and Software Technology* 37(12): 681–689.
- Larsen, T. J., and Naumann, J. D. 1992. An experimental comparison of abstract and concrete representations in systems analysis. *Information and Management* 22: 29–40.
- Levitin, A., and Redman, T. 1995. Quality dimensions of a conceptual view. *Information Processing and Management* 31(1): 81–88.
- Lientz, P. B., Swanson, E. B. 1980. *Software Maintenance Management*. Addison-Wesley.
- Liu, L. 1995. Adaptive schema design and evaluation in an object-oriented information system. In *Proceedings of the International Conference on Entity-Relationship approach (ER'95)*, pp. 21–31.
- Lubars, M., Meredith, G., Potts, C., and Richter, C. 1992. Object oriented analysis for evolving systems. In *Proceedings of the International Conference on Software Engineering (ICSE'92)*, pp. 173–185.
- Marche, S. 1991. *Models of Information: The Feasibility of Measuring the Stability of Data Models*. PhD Dissertation, London School of Economics and Political Science, 184 p.
- Marche, S. 1993. Measuring the stability of data models. *European Journal of Information Systems* 2(1): 37–47.
- Mili, H., Mili, F., and Mili, A. 1995. Reusing software: issues and research directions. *IEEE Transactions on Software Engineering* 21(6): 528–561.
- Moody, D. L., Shanks, G. 1994. What makes a good data model? Evaluating the quality of entity relationship models. In *Proceedings of the 13th International Entity Relationship Conference (ER'94)*, pp. 94–111.
- Parnas, D. L. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* 5(2): 128–138.
- Parsons, J., and Wand, Y. 1997. Choosing classes in conceptual modeling. *Communications of the ACM* 40(6): 63–69.
- Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., and Votta, L. G. 2001. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering* 27(12): 1134–1144.

- Prieto-Diaz, R. 1990. Domain Analysis: An introduction. *ACM SIGSOFT Software Engineering Notes* 15(2): 47–54.
- Rettig, M. 1993. Extended Objects. *Communications of the ACM* 36(8): 19–24.
- Rolland, C., and Cauvet, C. 1992. Trends and perspectives in conceptual modelling. In Loucopoulos, P. and Zicari, R. (eds.), *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*. Wiley.
- Roman, G.-C. 1985. A taxonomy of current issues in requirements engineering. *IEEE Computer* 18(4): 14–22.
- Rosson, M. B., and Alpert, S. R. 1990. The cognitive consequences of object-oriented design. *Human-Computer Interaction* 5(4): 345–379.
- Rowe, D., Leaney, J., Lowe, D. 1998. Defining systems evolvability—A taxonomy of change. In *Proceedings of the 11th International Conference on the Engineering of Computer Based Systems (ECBS'98)*.
- Schmid, H. A. 1997. Systematic framework design by generalization. *Communications of the ACM* 40(10): 48–51.
- Shanks, G. 1997. Conceptual data modelling: An empirical study of expert and novice data modellers. *Australian Journal of Information Systems* 4(2): 63–73.
- Shaw, M., and Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.
- Simsion, G. C. 1994. *Data Modeling Essentials: Analysis, Design and Innovation*. International Thompson Publishing.
- Storey, V. C., Thompson, C. B., and Ram, S. 1995. Understanding database design expertise. *Data and Knowledge Engineering* 16: 97–124.
- Takang, A. A., and Grubb, P. A. 1996. *Software Maintenance: Concepts and Practice*. International Thomson Computer Press.
- Tracz, W. 1995. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes* 20(3): 49–62.
- Van Der Sanden, W., Campschroer, J., and Dejel, F. 1997. Flexibele systemen met generieke software. *Informatie* 39(1): January, 40–47.
- Veldwijk, R. J., Boogaard, M., and Spoor, E. R. K. 1994. Assessing the software crisis: Why information systems are beyond control. *Information Sciences* 81(1–2): 103–116.
- Vessey, I., and Conger, S. A. 1993. Learning to specify information requirements: The relationship between application and methodology. *Journal of Management Information Systems* 10(2): 177–201.
- Wand, Y., and Weber, R. 2002. Information systems and conceptual modeling—A research agenda. *Information Systems Research* 13(4): 363–376.
- Wand, Y., Monarchi, D. E., Parsons, J., and Woo, C. C. 1995. Theoretical foundations for conceptual modelling in information systems development. *Decision Support Systems* 15: 285–304.
- Yeh, R. T. 1990. An alternative paradigm for software evolution. In P. A. Ng, R. T. Yeh (eds.), *Modern Software Engineering*. Van Nostrand Reinhold, pp. 7–22.



Jan Verelst is currently working at the Department of Management Information Systems (MIS) at the Faculty of Applied Economics of the University of Antwerp. He received his Ph.D. in MIS from the University of Antwerp in 1999. His research interest include aspects of systems analysis and design such as conceptual modeling, software evolvability/maintainability. He is a member of the ACM and the IEEE.