

Cognitive learning efficiency through the use of design patterns in teaching

Gwendolyn Kolfshoten,
Delft University of Technology
g.l.kolfshoten@tudelft.nl

Stephan Lukosch
Delft University of Technology
s.Lukosch@tudelft.nl

Alexander Verbraeck,
Delft University of Technology,
University of Maryland
a.verbraeck@tudelft.nl

Edwin Valentin,
Delft University of Technology,
Accenture
edwinvalentin@accenture.com

Gert-Jan de Vreede,
University of Nebraska at Omaha,
Delft University of Technology
gdevreede@mail.unomaha.edu

ABSTRACT

Processes and systems in organizations become increasingly complex and dynamic. This requires managers of expert teams to quickly gain knowledge and insight outside their prime area of expertise. To transfer expert knowledge and to reuse design solutions design patterns can be used as building blocks for the development of systems and processes. The use of design patterns can increase the efficiency of design & implementation of solutions and in some cases it can enable automated implementation of design. This allows the expert to re-use components to accommodate new requirements in a more flexible way. However, the advantage of design patterns might go beyond re-use, design efficiency and flexibility. This paper argues that in addition to the benefits described above, there is a specific added value for the use of design patterns by novices to acquire design skills and domain knowledge. We propose that design patterns, due to their conceptual design, offer information in a way that enables the creation of better linkages between knowledge elements and improve the accessibility of the information in the memory. For this hypothesis we will analyze the literature on cognitive load and cognitive learning processes, and add to this three case study experiences in which novices and experts were offered design patterns to develop and implement systems and processes.

Keywords

Design patterns, cognitive load, design skills, learning efficiency , expertise reversal effect.

INTRODUCTION

Learning efficiency becomes increasingly important, both in educational setting and especially in organizational contexts. Our society has been coined a knowledge economy for decades, but increasing complexity of networked organizations and infrastructures demands that knowledge workers become life long learners. Artifacts that are developed today are the result of close collaboration between experts and professionals in various domains, based on the requirements of multiple actors with conflicting stakes. In this context it is critical that professionals share knowledge in a way that makes it ready applicable, and that they learn new knowledge in an efficient and effective way. We can define learning efficiency as the speed in which novices gain skills and knowledge that enables them to start performing equal to experts.

One way to foster learning efficiency and the reuse of expertise is the use of design patterns. The original pattern concept described by Alexander [Alexander 1979] has been widely adopted in the software engineering world after introduction by the gang of four [Gamma, Helm et al. 1995]. Alexander [1979] originally described his design patterns as re-usable solutions to address frequently occurring architectural problems. In Alexander's words: "a pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [p. x, Alexander, Ishikawa et al. 1977]." One of the key purposes of pattern languages is to use them to enable novices, 'anyone' to use the expertise captured in the language [Alexander 1979]. Design patterns seem especially useful to transfer tacit best practices and expertise. Patterns are captured in a specific structure, they convey a context, problem, an example, and a reference to related patterns. Further, patterns have catchy names and a picture to explain it.

When looking at the difference between novices and experts, the difference is often not the amount of information available. Information can nowadays be looked up or offered just in time on the job. However, expertise is characterized with the ability of a person to apply information to solve a problem, to make critical choices. In this paper we will explore the role of design patterns in the training of novice designers. We propose that design patterns do not only enable efficiency and flexibility of the design effort for novices, but also increase their understanding of the design process and the domain in which they design. Furthermore, we will explore the cognitive effect of offering knowledge in the shape of design patterns, and its implications for learning efficiency. We will therefore analyze the design pattern concept in the light of Cognitive Load Theory (CLT) [Sweller 1988]. CLT explains how information can be offered to users in a way that allows them to optimally use the capacity of the human brain for learning and comprehension. The remainder of this paper will first explain the design pattern concept and their use in design training and knowledge transfer. Secondly, we present a proposition on how the use of design patterns has an effect on cognitive processes and learning efficiency. Next, we present the result of three exploratory experiments in which the effect of design patterns on learning efficiency and cognitive learning effects. We use these results to evaluate our propositions and recommend further research.

DESIGN PATTERNS FOR TRAINING OF DESIGN SKILLS AND DOMAIN KNOWLEDGE

Pattern languages are used to gather and share reusable solutions to recurring problems. Pattern languages are more than a compendium of best practices. A pattern language is a coherent set of design patterns. A design pattern is defined by Alexander as a re-usable solution to address a frequently occurring architectural problem. Design patterns serve several purposes [Alexander 1979]; they providing a convenient common language for communication, they help inspiring and designing new or improved patterns, they offer a basis to design larger systems based on individual patterns, they support teaching, capturing, and sharing expert design knowledge, they are described in a simplistic way to enable use by novices, they encompass quality to ensure they are used to create designs that improve the quality of life and, combined in the pattern language they create a coherent system.

After their introduction in software engineering [Gamma, Helm et al. 1995] they have been adopted in various fields such as knowledge management [May and Taylor 2003], human-computer interaction [Borchers 2001], computer-mediated interaction [Schümmer and Lukosch 2007], communication software [Rising 2001], productive software organizations [Harrison and Coplien 1996], hypermedia design [Rossi, Garrido et al. 1995], e-learning [Niegemann and Domagk 2005] and pedagogy [Eckstein, Manns et al. 2001]. While the first patterns of the gang of four were focusing on designers only, later pattern languages stated patterns in a way so that they are understandable by developers and end-users. These patterns can serve as a Lingua Franca for design [Erickson 2000] that helps end-users and developers in communication.

Design patterns help software engineers to prevent common errors and obstacles [Monroe, Kompanek et al. 1997]. Patterns as a focus of study, as opposed to tools, show designers the structure of the solution instead of the objects in the solution [Coplien 1997]. Furthermore, patterns empower users to find working and established proven solutions. Patterns allow experts to describe their knowledge as rules of thumb. These rules include a problem description, which highlights a set of conflicting forces and motivate a proven solution, which helps to resolve the forces. Thus, patterns can serve as an educational resource, as patterns make the core of a problem and the corresponding solution explicit by explaining what happens to the inner forces. In that line, [Johnson 1997] and [Brugali, Menga et al. 1997] pointed out that the power of pattern languages lies in its potential for serving as an educational and communicative vehicle.

To understand how patterns contribute to learning efficiency we will look at the cognitive implications of the use of design patterns to communicate knowledge. For this we need a general understanding of the cognitive mechanisms involved in learning. For this we look at cognitive load theory.

COGNITIVE LOAD THEORY

Cognitive load can be defined as *the cognitive effort made by a person to understand and perform his task*. It has both a task-based dimension (mental load) and a person-based dimension (mental effort) [Sweller, Merriënboer et al. 1998]. Cognitive load theory (CLT) is based on the assumption that our short-term or working memory is limited to seven plus or minus two information elements [Miller 1956]. This is the information that we can process at a certain moment.

Besides working memory the model assumes that we have a long-term memory in which information is stored, in so called schemas¹ [Sweller 1988]. To learn we need to consciously combine individual elements of information to build schemas. Schemas can be handled by our working memory as an individual component. The schema is not just a storage frame; information in the schema can be retrieved unconsciously. An example of this is reading. Experienced readers for instance do not process every character they read anymore; they recognize entire words, or even parts of sentences [Sweller, Merriënboer et al. 1998]. Therefore, the larger the schema, the more information we can process in the same time in our working memory, the faster we can gain new understanding and combine schemas to find solutions or answers to problems.

The availability of schemas determines the difference between experts and novices in several ways [Sweller 1988]: An expert, compared to a novice does not have more schemas, but larger schemas. A second difference is that an expert recognizes patterns of problems from previous experience, and combined these in his schema with solution-directions, while novices do not possess such schema and thus have to solve the problem from scratch. This lack of sophisticated schemas causes another difference between novices and experts. Experts categorize their knowledge based on different solution modes, while novices do not yet see the direct relation between problems and solutions, they can only structure their schemas based on surface structures such as shared objects.

The cognitive load theory explains how we use our cognitive capacity to construct schemas. There are 3 types of cognitive load [Sweller 1988]:

- Intrinsic cognitive load, is the cognitive load that is inherent to the task, and that is defined by the intrinsic task complexity.
- Extraneous cognitive load, is the cognitive load caused by the presentation and transition method of the information.
- Germane cognitive load, is the cognitive load instrumental to building the schemas and storing them in the long term memory.

Reduction of the intrinsic or extraneous cognitive load makes capacity available within the working memory to increase the germane cognitive load and thus make it possible to improve the building of schemas to store the information in the long term memory; which constitutes learning and understanding. CLT offers different methods to reduce extraneous cognitive load such as offering parsimonious information elements and avoiding split attention that is caused by disintegrated information such as a picture with a separate description [Sweller, Merriënboer et al. 1998]. In addition Pollock et al [2002] suggest that the intrinsic cognitive load can be reduced for complex systems that are difficult to understand even with very low extraneous cognitive load. They suggest offering a basic framework that can be schematized and in which interaction between information elements is removed. These schemas can then be used as a basis to learn the other material by offering the complete information with the interaction as a second step in the learning process.

Cognitive load can thus be reduced, leaving more memory space for germane cognitive load instrumental to the building of schemas. However, when these schemas are already available and automated, as is the case in the mind of an expert, the methods tend to have a reverse effect [Kalyuga, Ayres et al. 2003]. For instance, offering explanation to the meaning of different shapes in a model will help a novice in modeling, but for an expert who already understands the meaning of the shapes, this is redundant information, and trying to ignore it or accidentally reading it will be redundant and thus increase cognitive load. This is called the expertise reversal effect; methods to reduce cognitive load for novices can increase cognitive load for experts and thus design support and modeling support should be different for experts and novices.

Learning to design systems and design skills is a complex learning goal. It requires the understanding of many elements and relations (intrinsic cognitive load), designs are often represented in “coding” such as a modeling language (extraneous

¹ The plural of schema is schemata, but we will stick to the language used by Sweller et al.

cognitive load), and besides building initial schema of these concepts (germane cognitive load), it also requires creativity of the designer (additional germane cognitive load). Offering a supportive method for this design effort can reduce cognitive load of the design task, but should be aimed consciously on either novices or experts to avoid a counter effect as described by Kalyuga et al [Kalyuga, Ayres et al. 2003]. In the next section we will discuss how the use of design patterns affects the cognitive load of the design and modeling effort.

THE EFFECT OF DESIGN PATTERNS ON COGNITIVE LOAD OF EXPERT AND NOVICE DESIGNERS

Design patterns offer ready made solutions for frequently recurring problems. A process often used for design or problem solving exists generally of several steps that include identification of the issue, analysis, finding (and evaluating) alternatives, choice and implementation [Drucker 1967; Ackoff 1978; Checkland 1981; Couger 1995]. The use of design patterns changes this approach. Where a general design approach requires finding and evaluating alternative solutions, a design process with design patterns requires only the choice and instantiation of design patterns, which can be easily combined and customized for implementation. This eliminates a complex and challenging step from the design process (finding and developing solutions) and it makes the design more flexible, as design patterns can be replaced or re-configured to adapt the design to new requirements. It is therefore expected, that design patterns quicken the design process, and especially quicken the effort of altering a model or design.

Besides this efficiency effect, design patterns are expected to have an effect on cognitive load. The effect of design patterns on extraneous cognitive load is not determined by the design pattern concept or the approach. Design patterns can be represented as (software) components or descriptions (recipes), and in some domains they might even be offered as physical components. However, design patterns are coherent components, and thus offer a good basis to integrate documentation and offer a parsimonious component to avoid the split attention effect [Sweller, Merrienboer et al. 1998], meaning that knowledge is offered in an integrated way, and sources of knowledge do not need to be intergraded by the student, which costs cognitive attention.

For intrinsic cognitive load we see a large parallel with the effect described and tested by Pollock et al. [Pollock, Chandler et al. 2002]. Pollock describes an approach to teach students information that is too complex to understand at once. Such information has such a high element interactivity (related information elements that cause complexity and are thus more difficult to learn and understand) that too many concepts should be held in the working memory at once to understand the concept. With no prior schema of the information, trying to understand it is very difficult. The student will have to build schemas without having an overview of the concept he is to learn. The approach prescribes to offer the information in smaller steps. For instance, the student first learns how to perform a certain task, without learning the explanation of why it is performed that way. Once this is understood and captured in a schema, he can step by step learn to understand the logic and reasoning behind the approach.

The use of design patterns in pattern languages does essentially the same. It divides a complex system in recognizable components, and first explains the designer how he can combine and use these components to represent the system. After this is understood, the designer can (if needed) further learn to understand why they work and what the logic behind these design patterns is.

Pollock found convincing evidence that this approach, named isolated-interacting elements approach had a significant effect on novice students, while it did not have an effect on more experienced students (note, no negative effect was found for these students) who, most likely already build some schemas of the information, and therefore do not need to study the separate components. We therefore offer the following propositions:

P (1a): Novices will faster gain understanding in modeling and design skills, when they learn to design with the design patterns approach first, before they learn to understand entire systems.

P (1b): Experienced (domain experts or more experienced designers) will not gain faster understanding of design skills with the use of design patterns.

The last type of cognitive load, germane cognitive load is to be stimulated. More germane cognitive load is better, as it constitutes learning. However, Sweller, [Sweller, Merrienboer et al. 1998] in his explanation about schema construction, raises an interesting conclusion. Schemas have two purposes: the storage and organization of information, and the reduction of working memory load. Through automation, also called chunking [Miller 1956; Simon 1974], larger schemas can be used in the working memory as one component. Experts in complex tasks such as chess and physics do not have better problem solving skills, rather they have a large set of larger, more complex schemas in their long term memory, that represent patterns of problems and related solutions. With these patterns available they can recognize the problem and pick the right solution [Sweller 1988; Sweller, Merrienboer et al. 1998]. Chi, Glaser and Rees [Chi, Glaser et al. 1982] discovered that experts

categorize problems based on the solution type, while novices categorize problems based on surface structures such as shared objects (all problems related to airplanes). This could indicate that the schema that novices build initially when they learn “problem solving from scratch”, are not efficient. The use of design patterns, which are in fact large solution based schemas, might thus offer the novices a framework to build better schemas from the beginning of their learning process. This leads us to the following propositions:

P (2a) Training novices with the use of design patterns will increase the quality of the schemas they build to represent a system.

P (2b) Training novices without the use of design patterns will cause them to build (in a similar time frame) lower quality schemas to represent a system.

In the following section we will describe three experiments in which novices (with different experience levels) and experts design decision support with and without the use of design patterns.

EXPERIMENTS

ThinkLet use in facilitation training

One of the main tasks of a facilitator is to design the collaboration process. In this experiment we asked novices with different amounts of training and experience to design collaboration processes using a thinkLet library. In this experiment² two groups of undergraduate students from Delft University of Technology, (n8 and n5) 12 novice designers in total, (one student participated in both groups) were divided in 3 experience categories based on the number of training hours they received and their experience in facilitation and GSS use. Where the novices only got an introduction about GSS and facilitation, the experienced students had actually experienced what it is to facilitate a collaboration process.

- Level 1: 2-4 hours, no facilitation experience, some with minor technical facilitation experience.
- Level 2: 8-10 hours, some GSS experience, but no facilitation experience.
- Level 3: approximately 20 hours, limited facilitation experience.

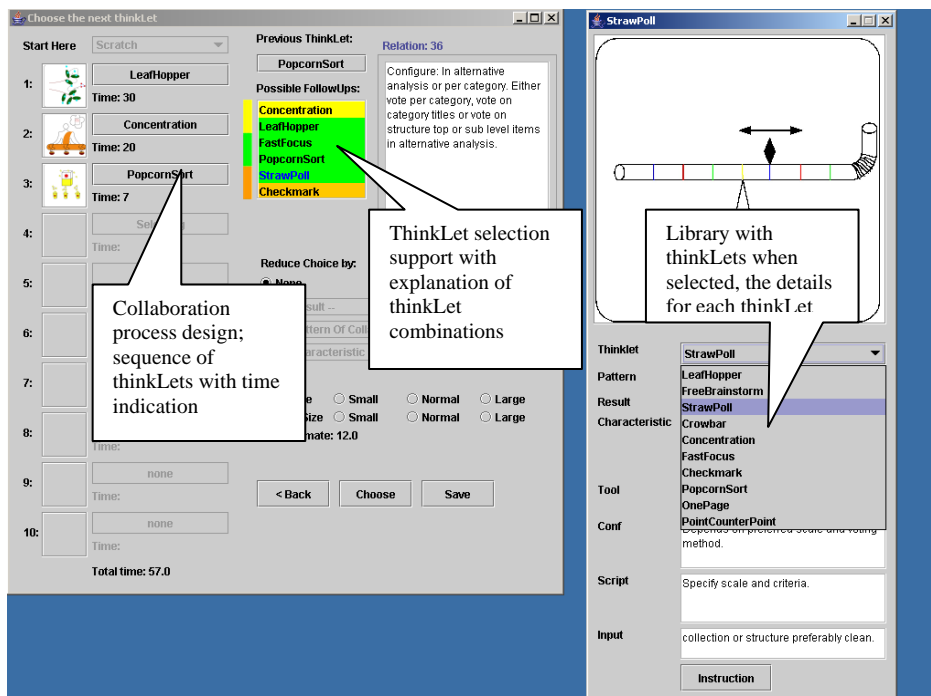


Figure 1. ThinkLet pattern language library with selection support.

² Published also in Kolfshoten, G.L. and Veen, W. (2005). Tool Support for GSS Session Design, *Hawaii International Conference on System Sciences*, Los Alamitos, IEEE Computer Society Press.

The students participated in a full day workshop in which they had to design 3 collaboration processes based on a case description. The case descriptions, based on real sessions, contained a goal statement, a group description, and the contours of the task. The students were supported in the design effort with a tool that offered them support in the choice among ten thinkLets (design patterns). The resulting process model had to specify all information required to facilitate the collaboration process. Despite the small number of students we made some interesting observations.

We measured the quality of design on a 1 to 10 scale. Two teachers of a facilitation class, both experienced facilitators, assessed the quality of the designs, and conflicts in assessment were resolved. Additionally the time spent on each design was measured. Table 1 indicates the quality of the first, second and sometimes third process model (not all students made all 3 designs due to lack of time) made by novices with different training history. Interesting is that the novices get up to speed and outperform more experienced students. The latter already adapted their own design approach, and found it hard to use the new approach. The efficiency is indicated by the time spent for a design. The time spent on design does decrease over time only for more experienced novices. The novices with low training time did not get faster in their design. A general decrease in design time could have been caused by the fact that motivation and energy were lower at the end of the day.

Design	level 3		level 2		level 1	
	Quality	n	Quality	n	Quality	N
1	6.5	6	6.3	2	5.6	3
2	5.9	6	6.7	2	5.7	5
3	6.2	4	6.9	1	7.3	2

Design	level 3		level 2		level 1	
	Time	n	Time	n	Time	n
1	79	6	94	2	108	5
2	83	6	86	2	70	5
3	38	4	60	1	105	2

Table 1 design quality (1-10 scale) and design time (in minutes) per experience level for design with building blocks.

Building block experiment

In 2004 a range of laboratory experiments was performed [Valentin, Verbraeck et al. 2003] by novices in simulation (undergraduate students from the Delft University of Technology) and simulation experts (simulation consultants of the company Rockwell Software, daily working with the simulation environment Arena).

The participants in the laboratory experiment received an individual assignment to develop a simulation model for a public transport system within 8 hours. Half of the participants received a set of simulation building blocks (instantiated design patterns) that were developed for modeling of public transportation systems and the other half had to use the basic modeling constructs of the Arena simulation environment. Fig. 2 shows the building block environment used. Afterwards the result of the novices and the experts was judged by professional simulation experts that have been involved in simulation studies in the public transportation sector for years as well as people who have acted as problem owners in these systems.

The novices in simulation that worked with the generic simulation environment worked hard, but were miles away from a complete solution. Most of these novices used a structured way of working and first focused on the movement of trains, before introducing passengers that use the trains. However, these novices spent a lot of time searching through the help files and evaluating different alternatives for modeling the system. Therefore, the novices spent most of their time in learning how to use the simulation environment, instead of developing a simulation models. The experts that used the generic simulation environment succeeded in developing models, but they got lost in details. The part of the simulation model that they had working was fine, but their model was not yet fully working and they would need a couple of hours more to finish the model.

The developers (both experts and novices) that worked with the simulation building blocks did not have any difficulty in developing a simulation model. Within only two hours they had simulation models that seemed to represent the system, but the models did not work as intended. The novices expected that they made mistakes. They started to make changes to the configuration of the building blocks and searched in the documentation of the building blocks to solve the problems and in the end, provided a model that offered sufficient insight for decision making. The experts worked in a different way. When they noticed problems, they expected that the building blocks were incorrect. They knew the simulation environment, and the things they noticed in the simulation building blocks collided with their existing schemas of simulation models and their working. The experts tried to understand the technology behind the simulation building blocks and once they noticed the

structure and logic underneath, they started to trust and understand the simulation building blocks and noticed that the mistake was due to their parameterization of the simulation building blocks and not due to faulty building blocks. The experts succeeded in correcting their simulation model and performed a couple of simulation experiments.

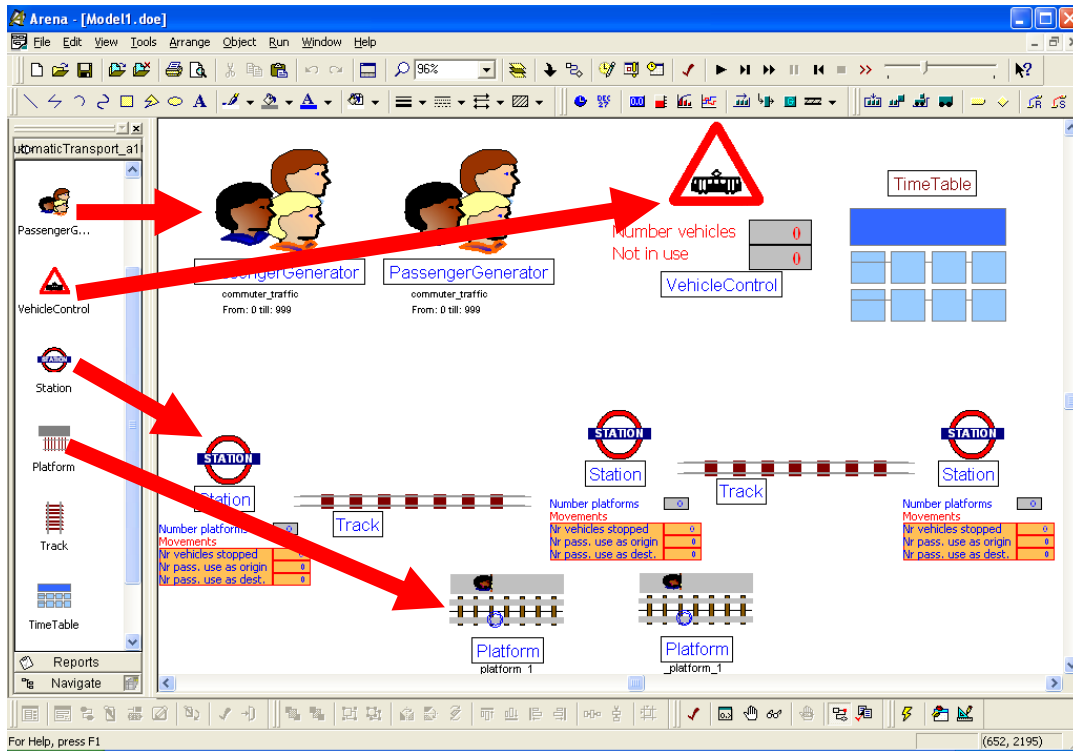


Fig. 2: Simulation building blocks

Designing groupware with patterns for computer-mediated interaction

The development of tools for computer-mediated interaction is a challenging task. Apart from the actual task of the application, e.g. editing texts or spreadsheets, developers have to consider various aspects ranging from low-level technical issues up to high-level application usage. As other authors pointed out for general software development (e.g. [Johnson 1997], [Brugali and Sycara 2000], or [Biggerstaff and Richter 1987]), the development of tools for computer-mediated interaction should focus on design reuse rather than code reuse. Capturing and reusing design insights is the most crucial part to reach this goal. For a successful tool for computer-mediated interaction it is also crucial to involve end-users in the development process [Schümmer, Lukosch et al. 2006]. Therefore, developers and end-users should be able to communicate with each other for a better understanding of the requirements. These goals can be reached by using patterns for computer-mediated interaction [Schümmer and Lukosch 2007] in a development process involving end-users [Schümmer, Lukosch et al. 2006].

We regularly use the patterns for computer-mediated interaction in educational as well as well product development settings to design and develop tools for computer-mediated interaction. Educational settings, e.g., include lab courses or thesis supervision. Product development settings, e.g., include the development of a collaborative learning environment or a group decision support system. Here, we want to report on our experiences when using the pattern language for computer-mediated interaction during a software lab course on groupware development. Six groups were asked to create a collaborative game. The groups consisted of up to six students, which either study in part-time or full-time. Apart from the lab course the students also attended other lectures. The lab course lasted half a year. All groups were introduced orally to the patterns for computer-mediated interaction at the beginning of the lab course.

Three of the groups used a groupware development framework to simplify the development of the collaborative game. These groups had to learn how to use the groupware development framework. One of these groups, e.g., implemented a collaborative labyrinth. The center of the labyrinth contains a treasure room. When the game starts, all players are positioned

outside of the labyrinth. They have the task to reach and enter the treasure room as a group in the fastest possible way. As the walls of the labyrinth sometimes change their position, the path to the treasure room can be blocked for some players. These position changes can only be undone from one side of the wall. Thus, sometimes players need help to unblock their path.

The other groups were implementing the collaborative game from scratch. One of these groups decided to implement a game called Pacmen. Pacmen is a multi-player adoption of the well-known arcade game Pacman, in which one user had to move around a Pacman through labyrinth walks. On its way through the labyrinth the user had to eat pills and avoid collisions with ghosts. In the multi-player version there are several Pacman, each controlled by one user. The users have to coordinate their movement to eat all pills in the fastest possible way.

The groups using the groupware development framework were complaining about the introductory overhead concerning the framework. All students reported that they experienced this learning as a difficult and very time consuming task. At the time when they felt familiar in the environment, the course was already in the final third. In the final third of the course, they still could not focus on issues concerning the collaborative game. As already noted by [Fayad and Schmidt 1997] this confirms that due to the large learning efforts the use of a framework only makes sense if it is used in long-term projects.

The members of the Labyrinth group took a long time until they finally agreed to use the framework. This was basically because one group member resisted learning a framework. His argument was that collaborative applications were not difficult to implement and the group would reach better results, if they chose to develop the collaborative game from scratch. The other group members were less confident and overruled the skeptic one. The skeptic decided to leave the group. After this discussion, the group started learning the framework, but had problems in using the framework. Finally, we directed their attention to some patterns, e.g. the LOVELY BAGS pattern. These patterns improved their understanding of the framework and they finally finished their project.

The pattern groups chose a programming language, which fit best for their group, and the patterns which fit best for their problems. After choosing the patterns, they started implementing their collaborative game without restrictions given by a framework. With the help of the patterns they soon had first design decisions, while the framework groups were overloaded by the available functionality. Thus, the pattern groups could concentrate on the issues concerning their collaborative game, the applicability of selected patterns, and their implementation. Especially, they focused on one pattern at a time which helped them to make piecemeal changes to the software and not lose any team members because of a too steep learning curve.

The Pacmen group, e.g., chose to implement a highly interactive game. After we presented the patterns to the group, the members immediately decided to use the REPLICATED OBJECTS and DECENTRALIZED UPDATES pattern. Both pattern descriptions note that the application of the pattern reduces the response time of the resulting application. This was the main concern in the development of Pacmen since the group wanted to focus on a fast and fluid game experience. The group members were able to transfer the symptoms of the presented patterns to their problem domain.

Later, there were discussions about using other patterns like MEDIATED UPDATES or even CENTRALIZED OBJECTS. These discussions mainly stem from the fact that these patterns promise less implementation effort. However, the group did not change their mind. At the end of the software lab all groups had to present their results. Each group tested the collaborative games of the other groups. Finally, the students were asked to vote for the best collaborative game. The first three places were occupied by the games of the groups that were developing their game from scratch, using our pattern language.

These are just first experiences. However, they indicate that developing an application from scratch can lead to competitive results, if developers are instructed on how to design and implement groupware applications.

DISCUSSION

In each of these illustrative experiments we saw the effect of design patterns on the cognitive load of the design and modeling effort, and of acquiring these skills. We saw a clear difference in the effect of the design patterns on experts and on novices. In this section we will further elaborate on this difference.

Experts and novices make simulation models in a different way. Experts recognize a system and are capable of linking the knowledge they have from the system to the modules available in a simulation environment. Novices are less aware of the capabilities of the modules of a simulation environment and thus have more difficulty to create a simulation model. We noticed the same effect with the thinkLets. Experienced students complained that they had to find the design pattern that offered them the tools and methods they were used to apply, while novices in the end provided better models with the use of the design patterns than the experienced users. This seems to corroborate with our first proposition; design patterns help

novices to faster gain understanding in modeling and design skills, while experts felt disturbed and disrupted in their effort by the design patterns.

Our second proposition stated that novices would develop higher quality schemas. Our first experiments suggest that the models made by novices were of higher quality than those of the experts, while the time they spend on working with and learning about the modeling approach and the decision support systems was significantly shorter. This leads us to the tentative conclusion that the use of design patterns does not only affect the efficiency of the design effort, it also constitutes *learning efficiency of novices* to gain design skills and it enhances the quality of their design. Especially in business setting optimizing both learning efficiency and outcome quality offers double value.

Future research should further analyze the added value of the use of design patterns in the transition of complex skills, such as the development of DSS, but also in other complex design disciplines. Increased understanding and improvement of the learning efficiency effect of design patterns could allow for the building of design and modeling support suites that allow novices to these disciplines to model and design their own decision support systems and this would make customized decision support available for a larger audience of managers and professionals in organizations without the need for large investments in training and the acquisition of external expertise. Additionally it might be interesting to look into the transition of such design approach to experts, which would require a technique to replace or alter an existing schemas construction to build one that is more efficient.

REFERENCES

- Ackoff, R.L. (1978). *The Art of Problem Solving*, John Wiley & Sons.
- Alexander, C. (1979). *The Timeless Way of Building*, New York, Oxford University Press.
- Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I and Angel, S. (1977). *A Pattern Language, Towns, Buildings, Construction*, New York, Oxford University Press.
- Biggerstaff, Ted and Richter, Charles (1987). Reusability Framework, Assessment, and Directions, *IEEE Software*(March): 41--49.
- Borchers, J. (2001). *A Pattern Approach to Interaction Design* New York, John Wiley and Sons Ltd.
- Brugali, D.; Menga, G. and Aarsten, A. (1997). The framework life span, *Communications of the ACM* **40**: 65-68.
- Brugali, Davide and Sycara, Katia (2000). Frameworks and pattern languages: an intriguing relationship, *ACM Computing Surveys* **32**,(1es): 2.
- Checkland, P.B. (1981). *Systems Thinking, Systems Practice*, Chichester, John Wiley & Sons.
- Chi, M.; Glaser, R. and Rees, E. (1982). Expertise in Problem Solving. *Advances in the Psychology of Human Intelligence*. R. Sternberg. New Jersey, Erlbaum.
- Coplien, J.O. (1997). Idioms and Patterns as Architectural Literature, *IEEE Software* **14**,(1): 36-42.
- Couger, J. D. (1995). *Creative Problem Solving and Opportunity Finding*, Danvers, Mass: Boyd And Fraser.
- Drucker, P.F. (1967). *The Effective Executive*, London.
- Eckstein, J.; Manns, M.L. and Voelter, M. (2001). Pedagogical patterns: capturing best practices in teaching object technology, *Software Focus* **2**,(1): 9-12.
- Erickson, T. (2000). Lingua Francas for design: sacred places and pattern languages, *Conference on Designing Interactive Systems*, ACM Press.
- Fayad, M.E. and Schmidt, D.C. (1997). Object-oriented Application Frameworks, *Communications of the ACM* **40**: 32-38.
- Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. (1995). *Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company.
- Harrison, B.N. and Coplien, J.O. (1996). Patterns of Productive Software Organizations, *Bell Labs Technical Journal*.
- Johnson, R.E. (1997). Frameworks = (Components + Patterns) *Communications of the ACM* **40**: 39-42.
- Johnson, Ralph E. (1997). Frameworks = (Components + Patterns), *Communications of the ACM* **40**,(10): 39-42.
- Kalyuga, S.; Ayres, P.; Chandler, P. and Sweller, J. (2003). The Expertise Reversal Effect, *Educational Psychologist* **38**,(1): 23-31.
- Kolfschoten, G.L. and Veen, W. (2005). Tool Support for GSS Session Design, *Hawaii International Conference on System Sciences*, Los Alamitos, IEEE Computer Society Press.
- May, D. and Taylor, P. (2003). Knowledge Management with Patterns: Developing techniques to improve the process of converting information to knowledge, *Communications of the ACM* **44**,(7): 94-99.

- Miller, G.A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information, *Psychological Review* **63**: 81-97.
- Monroe, R. T.; Kompanek, A.; Melton, R. and Garlan, D. (1997). Architectural Styles, Design Patterns, and Objects, *IEEE Software* **14**,(1): 43-52.
- Niegemann, H.M. and Domagk, S. (2005). ELEN Project Evaluation Report, from <http://www2tisip.no/E-LEN>.
- Pollock, E.; Chandler, P. and Sweller, J. (2002). Assimilating Complex Information, *Learning and Instruction* **12**: 61-86.
- Rising, L. (2001). *Design Patterns in Communication Software*, Cambridge, Cambridge University Press.
- Rossi, G.; Garrido, A. and Carvalho, S. (1995). Design Patterns for Object-Oriented Hypermedia Applications. *Pattern Languages of Program Design 2*. J.M. Vlissides; J.O. Coplien and N.L. Kerth, Addison-Wesley: 177-191.
- Schümmer, T. and Lukosch, S. (2007). *Patterns for Computer-Mediated Interaction*, West Sussex, England, John Wiley & Sons Ltd.
- Schümmer, Till; Lukosch, Stephan and Slagter, Robert (2006). Using Patterns to empower End-users - The Oregon Software Development Process for Groupware, *International Journal of Cooperative Information Systems, Special Issue on '11th International Workshop on Groupware (CRIWG'05)'* **15**,(2): 259-288.
- Simon, H.A. (1974). How Big is a Chunk?, *Science* **183**,(4124): 482-488.
- Sweller, J. (1988). Cognitive load during problem solving: Effects on learning, *Cognitive Science* **12**: 257-285.
- Sweller, J.; Merrienboer, J.G. van and Paas, F.G.W.C. (1998). Cognitive Architecture and Instructional Design, *Educational Psychology Review* **10**,(3): 251-296.
- Valentin, E.C.; Verbraeck, A. and Sol, H.G. (2003). Effect of Simulation Building Blocks On Simulation Model Development, *International Conference of Technology, Policy and Innovation*.