Available online at www.sciencedirect.com
SCIENCE DIRECT
Int. J. Human-Computer Studies 64 (2006) 599–610
International Journal of Human-Computer Studies
www.elsevier.com/locate/ijhcs

Groupware development support with technology patterns

Stephan Lukosch*, Till Schümmer

Department for Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany

Available online 20 March 2006

Abstract

Groupware development support should educate developers on how to design groupware applications and foster the reuse of proven solutions. Additionally, it should foster communication between developers and end-users, since they need a common language and understanding of the problem space. Groupware frameworks provide solutions for the development of groupware applications by means of building blocks. They have become a prominent means to support developers, but from our experience frameworks have properties that complicate their usage and do not sufficiently support groupware developers. We argue for a pattern approach to support the technical aspects of groupware development. Patterns describe solutions to recurring issues in groupware development. They serve as educational and communicative vehicle for reaching the above goals. In this article, we provide a pattern language focusing on technical issues during groupware development. Experiences when using the language in an educational setting and a product development setting have shown that the patterns are a supportive means for the proposed goals.

Keywords: Groupware development; Design patterns; Education of groupware developers; Oregon Software Development Process (OSDP)

1. Introduction

The development of groupware applications is still 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. Among others,

- Network connections between the collaborating users have to be established to enable communication.
- Parallel input from the collaborating users has to be handled.
- Specific group functions have to be included to provide group awareness.
- The data of a groupware application has to be shared and kept consistent to allow users to work on common task at all (Ellis et al., 1991).

These issues are often not part of the professional training of software engineers. Instead, software engineers learn the basic principles that empower them to create any kind of software. These include modeling techniques like the unified modeling language or programming techniques like object-oriented programming and programming languages like Java or C#. These principles and techniques theoretically empower developers to create groupware applications. However, developers first have to understand the groupware specific issues discussed above. Then, they have to learn how to deal with these issues before they finally can implement their first groupware application.

Currently, groupware frameworks are the most prominent means to support developers. They offer solutions for the development of groupware applications as pre-fabricated building blocks. But frameworks in general have properties that complicate their usage and thus do not sufficiently support groupware developers. Groupware frameworks, e.g. prescribe the group process, the programming language, and the distribution architecture. From our point of view, groupware development support should

- Educate developers on how to design and implement groupware applications.
- Ease the communication between developers, clients, and end-users for a better understanding of the requirements.

*Corresponding author. Tel.: +49 2331 987 4117; fax: +49 2331 987 313.
E-mail addresses: stephan.lukosch@fernuni-hagen.de (S. Lukosch), till.schuemmer@fernuni-hagen.de (T. Schümmer).

1071-5819/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2006.02.006

- Structure the gathering of end-user requirements.
- Foster reuse of proven solutions.

We propose to achieve these goals by providing developers with a pattern language that can be used in a development process involving end-users. The idea of patterns originates from Alexander's work (Alexander et al., 1977; Alexander, 1979) in urban architecture. According to Alexander, "patterns describe a problem which occurs over and over again and the core of the solution to that problem". An interconnected set of patterns is called a pattern language. Patterns of a pattern language are intended to be used together in a specific problem domain.

Gamma et al. (1995) applied the idea of design patterns to the application domain of software design. These patterns capture frequently used software practices found in object-oriented design. While the first patterns of the gang of four where focusing on designers only, later pattern publications, e.g. concerning human–computer interaction (Borchers, 2001), hypermedia design (Rossi et al., 1995), or pedagogy (Eckstein, 1999), 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. Johnson (1997) and Brugali et al. (1997) pointed out that the power of pattern languages lies in its potential for serving as an educational and communicative vehicle.

Pattern languages can teach well-known solutions, improve the design and finally increase the software quality. To elaborate on these arguments, we first discuss current groupware development support before we present our approach, provide an extract of a pattern language for groupware development, and finally discuss usage experiences with this pattern language.

2. Current groupware development support

Currently, groupware frameworks are the most prominent means to support groupware developers. Well-known examples for groupware frameworks are *Suite* (Dewan and Choudhary, 1992), *GINA* (Berlage and Genau, 1993), *Rendezvous* (Hill et al., 1994), *DistView* (Prakash and Shim, 1994), *NSTP* (Patterson et al., 1996), *GroupKit* (Roseman and Greenberg, 1996), *COAST* (Schuckmann et al., 1996), *Sync* (Munson and Dewan, 1997), *Habanero* (Chabert et al., 1998), *DOORS* (Preguica et al., 2000), *DyCE* (Tietze, 2001), or *DreamObjects* (Lukosch, 2003).

Groupware frameworks help during the development process by providing components that hide most of the *dirty and difficult* work such as instance network connection management, process scheduling, data sharing, or data consistency. They also impose a specific way of shaping the group process by, e.g. providing means for starting a collaborative session. If the framework perfectly matches the requirements of the project, it will simplify the development. Unfortunately, this is often not the case.

We experienced and identified several properties that complicate the use of frameworks, i.e. the chosen programming language, the supported distribution architecture, the dominance of frameworks in the application, the use of black-box components, the lack of documentation, and communication problems between end-users and developers.

Framework developers often chose the *programming language* that is currently en-vogue or that has been used in other projects before. Reuse is thus limited by the chosen programming language. Two examples of frameworks where the programming languages are a critical issue are COAST and GroupKit. Both used programming languages that provide a very high level of abstraction and make the process of implementation very easy. But the languages, i.e. Smalltalk for COAST and Tcl/Tk in the case of GroupKit, are not part of the standard curricula for software developers, which implies that users of the framework would have to learn the programming language before they can use the framework.

As with the programming languages, framework developers often limit the applicability of the framework by choosing one *distribution architecture* that fits the first intended applications best. This may mean that the framework for instance requires a client-server architecture or that it uses peer-to-peer mechanisms.

Framework developers often assume that the application can be created on top of exactly one framework. They design the framework as center around which the application should be built. The framework in this case *dominates* the application structure. This gets complicated when the application under development could benefit from more than one framework (Fayad and Schmidt, 1997). The components of the framework define the context in which the framework has to be used, e.g. available network connections. Components from different frameworks usually make different assumptions regarding their context, which makes it hard to use them together (Johnson, 1997).

Frameworks often only provide *black-box components*. These components are designed for a specific context like a specific structure of the underlying data model or specific mechanisms for processing user input. Some frameworks allow developers to exchange components of the framework with application specific components that implement the same interface. Unfortunately, the access points for exchanging these components are limited and depend on the configuration needs foreseen by the framework developers.

Most groupware frameworks have emerged from research projects. Thus, the main focus has been on the functional aspects of the framework rather than the *documentation* of the framework for novices. Johnson (1997) noted that frameworks describe the application domain by means of a programming language. But just by using the framework or even taking a look at source code

of the framework it is difficult to "learn the collaborative patterns of a framework by reading it" (Johnson, 1997). However, a didactically sound description of the framework's dynamic aspects is crucial for training developers in using the framework. Nowadays, many projects are carried out as open source projects. Though this allows to adapt the functionality of the framework, it is still hard as open source projects as well often lack of documentation.

Finally, groupware frameworks often address the problem space from a technical perspective rather than approaching the problem space from a human–computer interaction view. The interaction design thus often becomes technology driven. Developers continue to use their technical language while clients and end-users express their requirements with terms used in the specific interaction setting. The *problem of communication* between clients, end-users, and developers stays an open issue in these cases.

From these observations, we conclude that the framework approach is not sufficient for supporting groupware development. In the next section, we will propose a different approach that introduces developers in the complex interdisciplinary field of groupware design, educates developers on how to design and implement groupware applications, and at the same time enables communication between clients, end-users, and developers.

3. Approach

As other authors pointed out for general software development (e.g. Biggerstaff and Richter, 1987; Johnson, 1997; Brugali and Sycara, 2000), we argue that groupware reuse should focus on design reuse rather than code reuse. The developers should be trained in a way so that they can understand and reproduce the framework developer's design decisions. Capturing and reusing design insights is the most crucial part to reach this goal. For a successful groupware application it is also crucial to involve end-users in the development process. Therefore, developers and end-users should be able to communicate with each other for a better understanding of the requirements. In our opinion, these goals can be reached by using patterns in a development process involving end-users.

The idea of patterns originates from Christopher Alexander's work (Alexander et al., 1977; Alexander, 1979) in urban architecture. According to Alexander, "patterns describe a problem which occurs over and over again and the core of the solution to that problem". Each pattern includes a problem description, which highlights a set of conflicting forces and a proven solution, which helps to resolve the forces. An interconnected set of patterns is called a pattern language. Patterns of a pattern language are intended to be used together in a specific problem domain for which the pattern language guides the design decisions in the specific problem domain.

Beck and Johnson (1994) argued "that existing design notations focus on communicating the *what* of designs, but almost completely ignore the *why*. However, the *why* of a design is crucial for customizing it to a particular problem." Compared to traditional design notations, e.g. UML diagrams (Fowler, 2000), patterns do not only focus on the result but also discuss the design rationale. This makes patterns independent from an implementation in a specific context or programming language. Developers do not have to look at the source code of the framework to try to understand the design decisions taken by the framework developers. From that perspective, patterns are a complementary approach to the documentation of frameworks.

We consider patterns on two levels, each with a different target group (Schümmer and Slagter, 2004):

- High-level patterns describe issues and solutions typically targeted at end-users.
- Low-level patterns describe issues and solutions typically targeted at software developers on a more technical level.

High-level patterns focus on the system behavior, as it is perceived by the end-user. They empower the end-users to shape their groupware application in order to meet the group's demands. Thus, high-level patterns need to be more descriptive and prosaic than low-level patterns that focus on how to implement that behavior. Low-level patterns focus on the implementation of the system and thus include more technical details.

Both low-level and high-level patterns can be positioned on a seamless abstraction scale. The more a pattern discusses technical issues, the lower is its level. Groupware technology patterns that deal with class structures, control flow, or network communication form the lower bound on the abstraction scale of patterns. Groupware usability patterns that focus on human interaction and address computer systems just as tools to support the human interaction would be placed near the upper bound on the abstraction scale. In the extreme, high-level patterns would describe how the end-user can compose off-the-shelf components and embed them in his work process. This would then mean that the software developer would no longer need to assist the end-user in implementing the pattern.

Note that the different levels of scale are focusing on differently sized modules within the system. Compared to construction, low-level patterns discuss, how nuts and bolts should be arranged or how door knobs should combine with a door, while high-level patterns focus on positioning or shaping a whole building or a city to make it a place for social interaction. When two patterns are on the same abstraction level, all forces addressed by the patterns should be roughly comparable in their scope and significance (Alexander, 1964). This is comparable to the analysis method proposed by Cockburn (2000) where requirements are split up or combined to create a set of equally sized requirements. As Cockburn (2000) argued, the different abstraction levels address different

stakeholders' needs. In case of groupware patterns, this means that technicians would more focus on low-level patterns, while end-users focus primarily on high-level patterns.

The main reason for us to require patterns on different levels of abstraction lies in the intended use of the patterns in the Oregon Software Development Process (OSDP) (Schümmer and Slagter, 2004). This process fosters end-user involvement by means of end-user centered patterns. End-users should interact with developers in different iteration types.

In requirements iterations, the end-users outline scenarios of use. This is informed by the use of high-level patterns that describe how group interaction can be supported with groupware. The scenarios are implemented during development iterations. Here, the software developers are the main actors. They translate the scenarios to working groupware solutions using low-level groupware patterns as a development guideline. End-users closely interact with the developers by proposing the application of patterns from the pattern language and identifying conflicting forces in the current prototype. When the groupware is used by the group, end-users reflect on the support offered by the groupware. Whenever the support can be improved, they either tailor the groupware using high-level patterns or escalate the problem to the developers who then start a development iteration. In all iteration types, the patterns primarily communicate design knowledge to the developers and the users. Compared to frameworks, the participants of the OSDP learn how conflicting forces should be resolved instead of just using pre-fabricated building-blocks. This will empower them to create customized solutions that better suit their requirements.

Another important criteria for organizing a pattern language are links between patterns. Alexander (1964, p. 106) noted that forces of a pattern or "requirements interact (and are therefore linked), if what you do about one of them in a design necessarily makes it more difficult or easier to do anything about the other". Thus, "each pattern sits at the center of a network of connections which connect it to certain other patterns that help to complete it. [...] And it is the network of these connections between patterns which creates the language" (Alexander, 1979, p. 313). In case of patterns in the same problem domain, this results in a densely connected graph of patterns with links of different relevance.

The application of a pattern language brings up additional relations between patterns, called *sequences*. A sequence describes the order in which patterns can be or were applied in the context of a specific use case. Since the pattern approach demands that developers focus on one pattern at a time and thus improve the system incrementally in piecemeal growth (Alexander et al., 1980; Schümmer and Slagter, 2004), we can create a linear sequence by ordering the patterns according to the point in time when they were used. A sequence is then a sentence spoken in the pattern language that changes all forces addressed by the patterns used as words in the sentence.

Summarizing, patterns are independent from programming languages. They help educating developers by providing well-known design insights and thereby foster design reuse. Links between the patterns point to other relevant issues and thereby support a piecemeal growth of the application under development. Finally, they improve the communication between clients, end-users, and developers, when used in a iterative development process like OSDP.

Within this paper, we will present a pattern language that covers high-level as well as low-level issues of groupware design. After providing a brief overview over the pattern language, we will illustrate its use by a pattern sequence that was applied in a product development setting, i.e. the development of a collaborative learning environment for our university.

4. A pattern language

The patterns, discussed in this article, concentrate on more technical issues concerning groupware applications. They were mined from experiences with the development of the COAST groupware framework (Schuckmann et al., 1996) and the DreamObjects platform (Lukosch, 2003). We used the pattern language in an educational and in a product development context. The experiences, we have made, are described later in this paper (see Section 5).

4.1. Pattern structure

Our patterns follow the pattern structure outlined in the OSDP (Schümmer et al., 2004). It is shaped to meet both end-user's and developer's needs for detail and illustration.

The *pattern name* is followed by the *intent*, and the *context* of the pattern. All these sections help the reader to decide whether or not the following pattern may fit into his current situation.

Then follows the core of the pattern composed of the *problem* and the *solution* statement separated by a *scenario* and a *symptoms* section. The *scenario* is a concrete description of a situation where the pattern could be used, which makes the tension of the *problem* statement tangible. The *symptoms* section helps to identify the need for the pattern by describing aspects of the situation more abstract again.

After the *solution* section, the solution is explained in more detail and indications for further improvement after applying the pattern are provided. The *collaborations* section explains the main components or actors that interact in the pattern and explains how they relate to each other. The *rationale* section explains why the forces are resolved by the pattern. Unfortunately, the application of a pattern can in some cases raise new unbalanced forces. These counter forces are described in the section labeled *danger spots*.

4.2. Language overview

The pattern map in Fig. 1 shows the patterns of this language and the relations between the patterns. If a pattern A points to another pattern B, pattern A is important as context for the referred pattern B. The pattern map is split up into several domains. The following list discusses these domains and presents a problem/solution pair for one pattern in each of these domains:

- *Data distribution*: The patterns in this domain are concerned about how to distribute and share data. For example, consider the problem/solution pair of the CENTRALIZED OBJECTS pattern:
  To enable collaboration users must be able to share the data. Therefore, manage the data necessary for collaboration on a server that is known to all users. Allow these users to access the data on the server.
- *Data consistency*: The patterns in this domain focus on keeping shared data consistent. An example is the IMMUTABLE VERSIONS pattern:
  Users want to be able to work independently and make their results accessible to other users—regardless the state of the artefact they started their work with. If two users change the same artefact, this results in conflicting changes and one change is often lost. Therefore, store copies of all artefacts in a version tree. Make sure that the versions stored in the version tree cannot be changed afterwards. Instead, allow users to store modifications of the version as new versions.
- *Session management*: The patterns in this domain focus on how to organize a collaborative session. The patterns range from low-level issues concerning latecomer support in WHAT HAS HAPPENED HERE? or high-level issues in the LOGIN pattern:
  You are developing an application that requires non-anonymous interaction. Therefore, provide a LOGIN screen that requests users to identify themselves by entering a login name and a password before they can start to use the application.
- *Awareness*: This domain provides patterns on the user interface level of the collaborative application. An example pattern for this domain is the USER LIST pattern:
  If users only share common data, they have no mutual awareness, who will perceive the activities that each user performs. Users do not like to perform activities, if they do not know, who observes them. Thus, they will not act freely with the shared artefacts. Therefore, show the names of all users who are in the same session in a user list and ensure that the list is always valid.

The next section shows a full pattern from our pattern language. The selected pattern is one of the patterns that

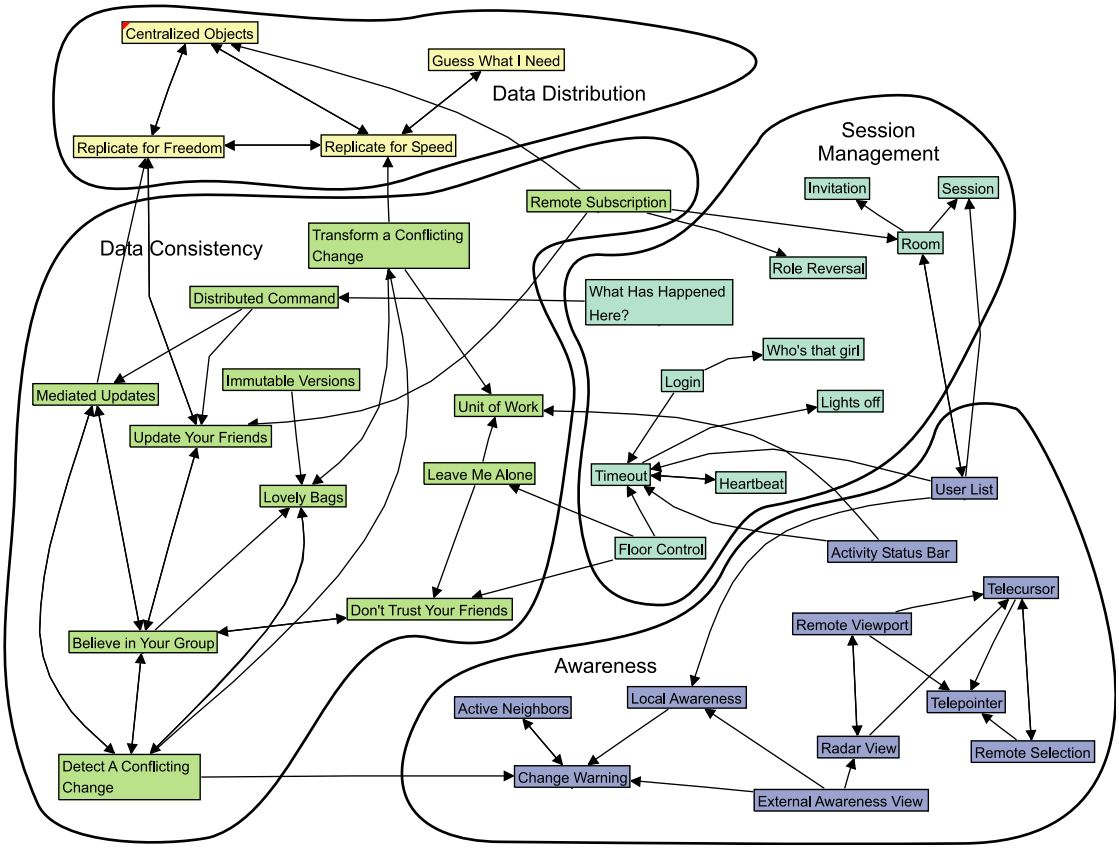

Fig. 1. Pattern map.

were used in our experience report about the product development setting.

4.3. An example pattern: LOVELY BAGS

Intent. Use bags for storing shared objects in a container because they provide the best concurrency behavior.

Context. You are using mechanisms to DETECT A CONFLICTING CHANGE. Now you are thinking about reducing the number of conflicting changes.

Problem. Access operations to shared container objects change the content of the container by adding or removing elements. Most of these operations are very bad regarding concurrency. Thus, synchronous collaboration on container objects often seems impossible.

Scenario. Imagine a chat system that was implemented on the basis of replicated objects. The easiest way of modeling a chat would be to have one long string to which the users append their contributions.

Now imagine that two users send a contribution at the same time. While the first client has appended its contribution (and sent out his update message to the second client), the second client already added its own contribution to its representation of the chat log. Thus, it will add the first client's contribution after its own contribution. And the first client will add the second client's contribution after its own contribution. Thus, both clients will see a different value for the chat log (which implies that one of the changes has to be undone to ensure consistency).

Would it be better if the clients used a list where they stored the individual contributions? No, because again both clients would try to access the same position in the object (the end) at the same time. This would be true for all ordered objects such as Lists, Vectors, Arrays. Whenever two clients try to modify an ordered collection in a way that affects the positions of newly added elements, these accesses cannot be performed concurrently.

Symptoms. You should consider to apply the pattern when ...

- Access operations to container objects are often rejected since two concurrent changes cannot be executed in different orders.

Solution. Therefore, wherever a high level of concurrency is needed model your container objects by means of a bag. If the container's entries need to be ordered equip the data entries with an order criterium that can be uniquely assigned by each client (e.g. the current time stamp together with a unique client ID) but still store the entries in the bag.

Collaborations. The main participant is the bag. The bag is a shared container object that can hold references to other objects. It allows duplicates and does not care about the order of the contained elements (from a mathematical point of view, bags are often called multisets). Clients perform operations on the bag concurrently. These operations are in most cases commutative (because of the ignored order and allowed duplicates). In cases, where the bag may only grow, one does no longer have to check for consistency since clients will never perform operations that are not commutative. In cases where the clients also remove elements from the bag ensure that the clients DETECT A CONFLICTING CHANGE.

When order is needed while iterating over the collection locally, it will be converted to an ordered collection before iterating it. This conversion requires that the contained elements provide one or more attributes that can be used as a sorting criteria.

Rationale. The simple explanation why this pattern works lies in the "lovely" nature of bags: a bag does not care about order and contained elements in the same way as other container classes do.

Compared to an ordered container object where the add operations are related to an insertion position (e.g. arrays or lists), bags produce the same result if two elements are added in different order.

Compared to container objects where the add operations depend on the current set of included objects (e.g. sets or dictionaries), bags produce the same results if an element was already present in the bag and two clients perform an add and a remove of this object concurrently.

This is illustrated in Fig. 2. In the left part, two users collaborate on a list. The list has the initial state `abc`. Then $User_1$ adds `d` while $User_2$ adds `e` at the same time. After both clients have performed their operation, they find time to UPDATE THEIR FRIENDS. But since the $User_1$'s state differs from $User_2$'s state, they will get different states if they perform the updates. Thus, one operation has to be undone.

Now consider the right part of Fig. 2: again, the users performed changes and UPDATED THEIR FRIENDS. But now,

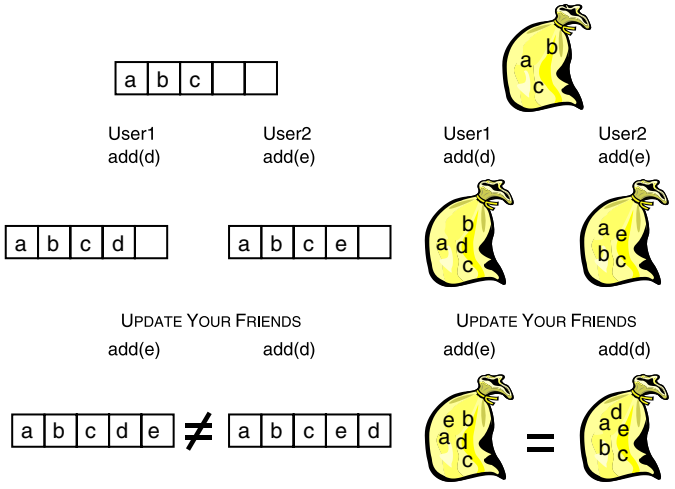

Fig. 2. Concurrent accesses to lists and bags.

the *add*-operations can be performed since adds do not depend on the object's state or the (non-existent) order of its elements. Both users will reach a consistent state after the updates.

In cases where order is needed, one can often restore this order. Consider for instance a sorted collection: this class ensures that the elements stored in instances of sorted collections will be stored according to the sorting order (or iterated according to the sorting order). The main reason for using such a sorted data structure is to speed up the process of iterating over the elements in a sorted way.

If a bag should be iterated in a sorted way, one can first convert it to a (local) sorted collection and then iterate over the local copy. The order (which drastically decreases possible concurrent operations on the data structure) is thus restored locally where needed. This makes the access to the ordered collection slower (and requires that each client sorts the elements) but it makes the data structure more robust for concurrent manipulations.

Another reason for storing objects in an ordered collection is the desire to access it by an index. The reason for this is often to speed up iteration again (by using a counter for the index when, e.g. iterating over an array in Java). As with the sorted access order, the iteration speed is less critical than the reduced concurrency. If the container object needs to be iterated frequently, you can perform the iterations on local copies of the replicated object, which are valid until the replicated object is changed again.

Danger spots. Unfortunately, even the lovely bag is still vulnerable regarding remove operations. If the same element is removed and added at the same time and the bag did not include the element before, this will result in different states of the bag depending on the order of the operations. Thus, prohibiting removes could be an option.

In some cases, arrays can be as attractive (or even more attractive) as bags: when the number of elements in the bag does not change often and an entry of the array is accessed by its index without relating to the other elements, this change is concurrent to all changes at other array positions. But if the array should change its size (or have a shared index as a current index), these attributes will reduce the concurrency of the array.

Known uses. Chat in FUB (Haake and Schümmer, 2003): FUB is a system built on top of COAST for supporting brainstorming in the context of distributed collaborative learning. Thus, the users are provided with two different kinds of chats: a brainstorming chat and a discussion chat, where concepts are discussed. While the brainstorming chat does not require any order (and can thus be directly modeled using a bag), the discussion chat needs to ensure that all chat entries are shown in the same order for all users.

Thus, the chat is modeled as a set of chat entries. Each entry has a time stamp that represents the (synchronized) time when the entry was added at the client. For displaying the chat log, all entries are sorted with the time stamp as a primary and the contained text as a secondary key. This ensures that all entries are shown in the same order at each client.

Arrangement of messages in Usenet (Horton and Adams, 1987): Usenet newsgroups are semantically represented as trees of messages (modeling the reply-relations between messages). While these relations could have been explicitly modeled at the news server, the designers rather decided to hide the relations within the news entries.

Each entry has a unique id, which is determined by the client that generated the entry. The entry can relate to a parent message, while the parent message is not changed at all (it does not know about the child messages). All entries are then stored in an unspecified collection by the server. The important issue here is that the protocol does not demand for any order of the entries. When clients request entries they can ask for a sorted version (by time), which will then be generated (by inspecting the date fields of the messages).

The client is responsible for ordering the entries when displaying the message threads.

Related patterns. Don't Trust Your Friends: When the bag has to be reordered before the client can process it, an additional computation overhead is added. This may slow down the responsiveness of the application. If the reordering process takes more time than obtaining a lock as proposed in Don't Trust Your Friends, you should prefer the locking mechanism.

Detect A Conflicting Change: Even when using Lovely Bags inconsistencies can occur. Detect A Conflicting Change allows to detect these inconsistencies.

5. Experiences

We applied the patterns both in an educational and in a product development context. While a former publication (Lukosch and Schümmer, 2004a) focused on the benefits of patterns in an educational context, we now focus on the experiences in the product development setting after briefly summarizing the experiences from the educational use of the patterns.

5.1. Educational setting

Computer science students at the FernUniversität in Hagen have to take part in a software development lab course to get their degree. The computer science department offers different lab courses each focusing on a different application domain. All lab courses last half a year. Each year our group offers a lab course on groupware development. In the observed course, students had to develop a groupware application for collaborative gaming. As the FernUniversität in Hagen is a distance teaching university, we run these courses in a blended setting. At the beginning of the course, students meet co-located for 3–4

days to form groups, discuss the problem, define work units, assign roles, etc. After the co-located phase, the groups work distributed to solve their task.

In the observed lab course, six groups consisting of up to six students were asked to create a collaborative game. Apart from the lab course the students also attended other lectures. All groups were introduced orally to the patterns of our pattern language at the beginning of the lab course. The students performed well using the patterns as an educative means for learning how to write collaborative applications. They read the problem and scenario section to identify if the pattern fits to their problem space. Relevant patterns affected the design of the application under development. Throughout the lab course, the students used the patterns to communicate necessary design decisions. As the patterns were available to the groups, group members could quickly understand ongoing discussions, even if these discussions were out of their specific work scope.

Compared to groups that focused more on using frameworks, the pattern groups produced better applications and proved a better domain knowledge in the final presentations of their approaches. Due to these results, we are currently using the pattern language in a similar lab course. As this lab course is still running, we do not have final results, but first impressions confirm that the pattern language highly supports the students in solving their tasks.

5.2. Product development setting

The product development setting was a two year project with the goal of creating a collaborative learning environment for our university. The project team consisted out of four senior employees and three students who contributed to the development. The resulting learning environment CURE (Haake et al., 2004) is in use since the end of the first year of the project. Since the project followed an iterative development approach—namely the OSDP (Schümmer and Slagter, 2004)—the software evolved in an organic way and was continuously used by end-users since the first prototype was available after the first weeks. In the second year, the user population grew from 30 initial users to more than 850 users who are currently active in the environment.

In a retrospective view, we could observe that the patterns were applied in the project in a specific sequence (see Fig. 3). The remaining part of this section will illustrate the sequence by elaborating on how the patterns were used in the concrete design problem of creating a web-based collaborative learning environment. Note that the sequence shown on the next pages is one application specific sequence in the pattern language. Other applications naturally have other sequences since the user requirements are different in different projects.

A precondition of the project was that it should utilize a web frontend for the users. The team started discussing whether or not anonymous interaction was appropriate for a collaborative learning environment. Very soon, it became obvious that users should interact with a distinguishable identity. Thus, a first requirement was that users should LOGIN before they were allowed to see any content in the environment.

Reading the LOGIN pattern, it became obvious that user information like a user's name and password need to be stored at a place where users could access it from different sites. The CENTRALIZED OBJECT pattern discusses issues for providing clients access to shared data from different sites. Thus, it was selected for the application. When users enter their login and password, the system can check the password by comparing it to a centralized object and in case of a successful login provide users access to centralized objects.

Without any information about the identity of the client, the corresponding server would have to request the LOGIN for each request. WHO'S THAT GIRL discusses this problem and suggests that the server maintains a session information that associates a client with its technical address revealed in its request. In CURE, this resulted in a dictionary that maps session keys of a web session to user IDs. The server can thereby determine the identity of the requesting user within each request.

Up to this point in time, the application was a standard web application with access control. The patterns helped to identify those aspects of the web application that must be present in the concrete context of the learning environment. Thus, the patterns were used as a means for selecting the required components from a servlet framework.

While the above patterns were focusing on the technical infrastructure, other patterns helped to organize the content managed in the environment. The first content-focused design decision was to organize learning material in a ROOM. ROOM provides the group with a place, where

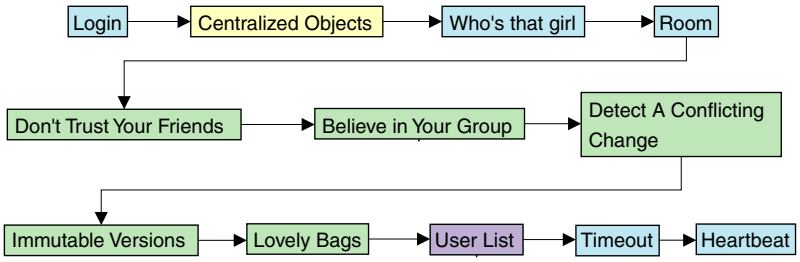

Fig. 3. Sequence of patterns used in the CURE project.

they can meet for collaboration. The pattern seemed appropriate since it eases the process of relating content with discussions on and collaboration with the content. CURE uses the ROOM pattern to cluster related learning material and bring together learners who are interested in the same topic (Haake et al., 2003). If, for instance, learners want to study linear algebra, they join the room for linear algebra, where they can find pages describing the topic, exercise pages that can be solved together with other room members and discussion channels for talking about the problems linear algebra.

After the system was at a state where a user could login and interact with content in a room, the next design step was to structure content modifications in the platform. Unlike in other web-based learning environments, where teachers provide static content for learners, CURE should allow students to actively provide content. Learners should be able to edit content in a room. But what happens if two users decided to change the content at the same time? The DON'T TRUST YOUR FRIENDS pattern suggests to solve this issue by using locks on a object when a user starts his operation and releasing the lock after the user has finished the operation. But as mentioned in the pattern, there are several drawbacks for such a solution:

- The response time would increase since a lock is required before a user could begin his change.
- Since clients could perform long changes (e.g. author a chapter of a seminal work), the rest of the group could not work in parallel.
- If clients intentionally or by accident forget to release a lock, the pages could be locked forever for the whole group.

Thus, the development team decided to follow an optimistic approach using the BELIEVE IN YOUR GROUP pattern. Every user was allowed to change the pages at each point in time. Most conflicts should be avoided using a social protocol. An important part of the pattern is to DETECT A CONFLICTING CHANGE or to resolve conflicting changes. How to resolve conflicts is often specific to the group process and the kind of artefacts used in the group. For CURE, the IMMUTABLE VERSIONS pattern provided the solution on how to store the pages in a room. CURE stores a page as a version tree holding all different versions of the page. When different users edit the same page at the same time, the resulting versions are stored as different leafs of the initial version. CURE then can DETECT A CONFLICTING CHANGE by scanning the version tree for multiple leafs.

The third major design step that we want to report on in this section was the introduction of highly synchronous interaction forms. End-users asked for synchronous communication channels and means for spontaneously meeting other learners.

The team first implemented a chat. One problem in this context was the high concurrency requirement of a chat communication. Chat contributions are often posted at close points in time which means that clients can perform conflicting changes. The first naive implementation of the chat that stored all chat entries as a linked list led to many conflicts. To resolve this, the developers decided to store chat entries in a set and sort them according to their timestamps. As described in the LOVELY BAGS pattern, this improved the concurrency behavior by eliminating all possible conflicts.

The implementation of the USER LIST followed the pattern quite closely. All users, who entered a room were stored on the server together with the room's data model. The user list was modeled as a page that subscribes to changes in the data model of a specific room. Whenever a user moves to another room, the data model is updated and remote subscribers are informed. An open problem was that users could simply close their browsers without notifying the server. The server would thus keep the users as room members in the USER LIST forever. The TIMEOUT pattern provides a solution for this problem. When users did not interact within the room for a specific amount of time, they are removed from the user list. A HEARTBEAT was added to prevent an accidental removal of a user from the user list when the user was just reading a page without requesting additional information from the server. The HEARTBEAT is sent as long as the page is displayed on the user's screen.

Fig. 4 shows the resulting application CURE. Three users are interacting in the room *Linear Algebra*. They are collaborating on a page with the most recent assignments. The active users are shown in a user list on the top left corner of Fig. 4. The communication channel is visible on the bottom of the page whenever a page from the room is shown. The examples mentioned above provide an excerpt of how patterns were used by the development team.

5.3. Results

Combined with the experiences of the student groups in the lab courses, we can observe that the patterns

- Served as an educative means both for novice students and more experienced developers.
- Helped to select required technology from larger frameworks for groupware applications by pointing to known uses of the pattern.
- Provided the teams with a common language and metaphors that ease the communication within the team and between the team and the end-users.
- Pointed the developers to related problems and solution, which they could have ignored otherwise.
- Helped to focus development activities on small problems and thus encourage iterative development and piecemeal growth of groupware applications.

6. Conclusions

Groupware developers have to consider various issues on a technical level. These issues are often not part of the

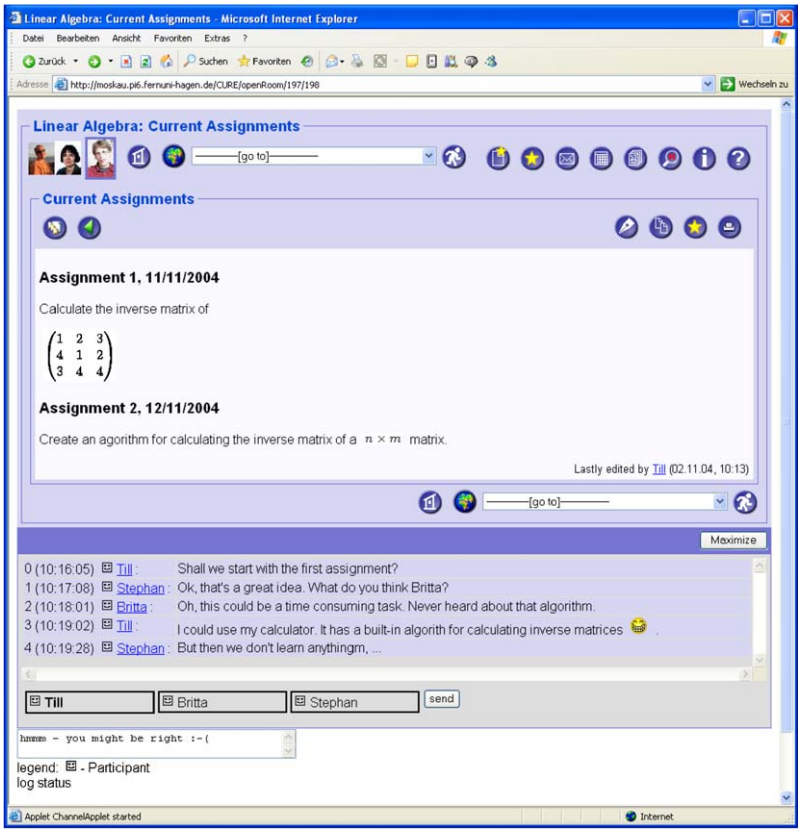

Fig. 4. A room in CURE.

professional training of software engineers. To relieve developers from recurring issues groupware platforms offer programming abstractions. In this article, we have shown that frameworks have properties that complicate their usage, i.e. the chosen programming language, the supported distribution architecture, the dominance of frameworks in the application, the use of black-box components, the lack of documentation, and communication problems between end-users and developers.

In our opinion, frameworks alone do not sufficiently support groupware developers. We argue that developers should be supported in developing groupware by teaching them on how to design and develop groupware applications and reuse proven solutions. When designing interactive applications, end-user involvement is a crucial issue. To foster communication between developers and end-users, they need a common language and understanding of the problem space. Pattern languages are an educational and communicative vehicle for reaching this goal.

This paper presented parts of a novel pattern language for groupware development. The discussed patterns focused on technical issues during groupware development. These technical issues are among the main obstacles during groupware development. The pattern language allows developers to use these solutions in their intended context. Experiences during software labs at our faculty and the development of a collaborative learning platform have shown that the patterns are a supportive means to instruct developers on how to implement groupware applications in a educational as well as a project setting.

The patterns in this paper are only a part of a bigger pattern collection that focuses on various issues in groupware development, e.g. virtual communities (Schümmer, 2004b), privacy (Schümmer, 2004c), group formation (Schümmer, 2004a), or shared object management (Lukosch and Schümmer, 2004b). Besides our pattern collection other collections have evolved in the area of groupware design. These collections are complementary to our's and can be combined to provide a greater understanding of all issues related to groupware development. On our scale some of them can be regarded as low-level patterns for distributed systems, e.g. POSA2 (Schmidt et al., 2000) or the collection of Guerrero and Fuller (1999) that focuses on technical infrastructures for groupware applications. Other collections have focused on specific groupware domains like knowledge management (Herrmann et al., 2003) or group support systems (Kolfschoten et al., 2004). As Erickson (2000) pointed out, pattern languages can serve as a *Lingua Franca* for human–computer interaction design. This motivated the collection of high-level human–computer interaction patterns, e.g. Borchers (2000).

Still, all these collections cover only a part of all issues concerning groupware development. Future work is needed to develop a more complete pattern collection and combine them in a coherent pattern language for supporting groupware development. We thus invite the community

to share their expertise by means of patterns and to evaluate the groupware patterns collection in diverse projects.

References

Alexander, C., 1964. Notes on the Synthesis of Form, 7th ed. Harvard University Press, Cambridge, MA.

Alexander, C., 1979. The Timeless Way of Building. Oxford University Press, New York.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., 1977. A Pattern Language. Oxford University Press, New York.

Alexander, C., Silverstein, M., Angel, S., Ishikawa, S., Abrams, D., 1980. The Oregon Experiment. Oxford University Press, New York.

Beck, K., Johnson, R., 1994. Patterns generate architectures. In: Proceedings of the ECOOP'94, vol. 821. Springer, Berlin, pp. 139–149.

Berlage, T., Genau, A., 1993. A framework for shared applications with a replicated architecture. In: Proceedings of the Sixth Annual ACM Symposium on User Interface Software and Technology. ACM Press, New York, pp. 249–257.

Biggerstaff, T., Richter, C., 1987. Reusability framework, assessment, and directions. IEEE Software (March), 41–49.

Borchers, J., 2001. A Pattern Approach to Interaction Design. Wiley, New York.

Borchers, J.O., 2000. A pattern approach to interaction design. In: Conference Proceedings on Designing Interactive Systems: Processes, Practices, Methods, and Techniques. ACM Press, New York City, NY, pp. 369–378.

Brugali, D., Sycara, K., 2000. Frameworks and pattern languages: an intriguing relationship. ACM Computing Surveys 32 (1es), 2.

Brugali, D., Menga, G., Aarsten, A., 1997. The framework life span. Communications of the ACM 40 (10), 65–68.

Chabert, A., Grossman, E., Jackson, L., Pietrovicz, S., Seguin, C., 1998. Java object-sharing in Habanero. Communications of the ACM 41 (6), 69–76.

Cockburn, A., 2000. Writing Effective Use Cases. Addison-Wesley, Boston.

Dewan, P., Choudhary, R., 1992. A high-level and flexible framework for implementing multiuser interfaces. ACM Transactions on Information Systems 10 (4), 345–380.

Eckstein, J., 1999. Workshop report on the pedagogical patterns project: successes in teaching object technology. In: Proceedings of OOPSLA'99 Educator's Symposium, Denver.

Ellis, C., Gibbs, S., Rein, G., 1991. Groupware—some issues and experiences. Communications of the ACM 34 (1), 38–58.

Erickson, T., 2000. Lingua francas for design: sacred places and pattern languages. In: Proceedings of the Conference on Designing Interactive Systems. ACM Press, New York, pp. 357–368.

Fayad, M.E., Schmidt, D.C., 1997. Object-oriented application frameworks. Communications of the ACM 40 (10), 32–38.

Fowler, M., 2000. UML Distilled, 2nd edn.—A Brief Guide to the Standard Object Modeling Language. No. ISBN: 020165783X. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Guerrero, L., Fuller, D., 1999. Design patterns for collaborative systems. In: Proceedings of the Fifth International Workshop on Groupware (CRIWG).

Haake, J., Schümmer, T., Haake, A., Bourimi, M., Landgraf, B., 2004. Supporting flexible collaborative distance learning in the cure platform. In: Proceedings of HICSS-37. IEEE Press, New York.

Haake, J.M., Schümmer, T., 2003. Some experiences with collaborative exercises. In: Proceedings of CSCL'03. Kluwer Academic Publishers, Bergen, Norway.

Haake, J.M., Schümmer, T., Haake, A., Bourimi, M., Landgraf, B., 2003. Two-level tailoring support for cscl. In: Favela, J., Decouchant, D. (Eds.), Groupware: Design, Implementation, and Use. Proceedings of the Ninth International Workshop (CRIWG 2003), vol. 2806. Lecture Notes in Computer Science. Springer, Heidelberg, pp. 74–82.

Herrmann, T., Hoffmann, M., Jahnke, I., Kienle, A., Kunau, G., Loser, K.-U., Menold, N., 2003. Concepts for usable patterns of groupware applications. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work. ACM Press, New York, pp. 349–358.

Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F., Wilne, W., 1994. The Rendezvous architecture and language for constructing multiuser applications. ACM Transactions on Computer–Human Interaction 1 (2), 81–125.

Horton, M.R., Adams, R., 1987. Standard for interchange of USENET messages. Request for Comments 1036, IETF.

Johnson, R.E., 1997. Frameworks = (components + patterns). Communications of the ACM 40 (10), 39–42.

Kolfschoten, G.L., Briggs, R.O., Appelman, J.H., de Vreede, G., 2004. thinkLets as building blocks for collaboration processes: a further conceptualization. In: de Vreede, G.-J., Guerrero, L.A., Raventós, G.M. (Eds.), Groupware: Design, Implementation, and Use, 10th International Workshop, CRIWG 2004. Lecture Notes in Computer Science 3198. Springer, Berlin, Heidelberg, San Carlos, Costa Rica, pp. 137–152.

Lukosch, S., 2003. Transparent and Flexible Data Sharing for Synchronous Groupware. Schriften zu Kooperations- und Mediensystemen—Band 2. JOSEF EUL VERLAG GmbH, Lohmar—Köln.

Lukosch, S., Schümmer, T., 2004a. Communicating design knowledge with groupware technology patterns—the case of shared object management. In: de Vreede, G.-J., Guerrero, L.A., Raventós, G.M. (Eds.), Groupware: Design, Implementation, and Use, 10th International Workshop, CRIWG 2004. Lecture Notes in Computer Science 3198. Springer, Berlin, Heidelberg, San Carlos, Costa Rica, pp. 223–237.

Lukosch, S., Schümmer, T., 2004b. Patterns for managing shared objects in groupware systems. In: Proceedings of the Ninth European Conference on Pattern Languages and Programs. Irsee, Germany, pp. 333–378.

Munson, J.P., Dewan, P., 1997. Sync: A java framework for mobile collaborative applications. IEEE Computer 30 (6), 59–66.

Patterson, J.F., Day, M., Kucan, J., 1996. Notification servers for synchronous groupware. In: Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work. Boston, MA, USA, pp. 122–129.

Prakash, A., Shim, H.S., 1994. Distview: Support for building efficient collaborative applications using replicated objects. In: Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work. Chapel Hill, NC, USA, pp. 153–164.

Preguica, N., Martins, J.L., Domingos, H., Duarte, S., 2000. Data management support for asynchronous groupware. In: Proceedings of the 2000 ACM conference on Computer Supported Cooperative Work. ACM Press, New York, pp. 69–78.

Roseman, M., Greenberg, S., 1996. Building real-time groupware with groupkit, a groupware toolkit. ACM Transactions on Computer–Human Interaction 3 (1), 66–106.

Rossi, G., Garrido, A., Carvalho, S., 1995. Design patterns for object-oriented hypermedia applications. In: Vlissides, J.M., Coplien, J.O., Kerth, N.L. (Eds.), Pattern Languages of Program Design 2. Addison-Wesley, Reading, MA, USA, pp. 177–191.

Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., 2000. Patterns for Concurrent and Networked Objects, vol. 2 of Pattern-Oriented Software Architecture. Wiley, New York.

Schuckmann, C., Kirchner, L., Schümmer, J., Haake, J.M., 1996. Designing object-oriented synchronous groupware with coast. In: Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work. Boston, MA, USA, pp. 30–38.

Schümmer, T., 2004a. GAMA — a pattern language for computer supported dynamic collaboration. In: Henney, K., Schütz, D. (Eds.), Proceedings of the Eighth European Conference on Pattern Languages of Programs (EuroPLoP'03). UVK, Konstanz, Germany.

Schümmer, T., 2004b. Patterns for building communities in collaborative systems. In: Proceedings of the Ninth European Conference on Pattern Languages of Programs (EuroPLoP'04). UVK, Konstanz, Germany, Irsee, Germany, pp. 379–440.

Schümmer, T., 2004c. The public privacy — patterns for filtering personal information in collaborative systems. Pattern Language workshopped at the CHI 2004 workshop on Human–Computer–Human-Interaction Patterns, FernUniversität in Hagen.

Schümmer, T., Slagter, R., 2004. The oregon software development process. In: Proceedings of XP2004.

Schümmer, T., Borchers, J., Thomas, J.C., Zdun, U., 2004. Human–computer–human interaction patterns: workshop on the human role in hci patterns. In: Extended Abstracts of the 2004 Conference on Human Factors and Computing Systems. ACM Press, New York, pp. 1721–1722.

Tietze, D.A., 2001. A framework for developing component-based cooperative applications. Ph.D. Thesis, Technische Universität, Darmstadt.