

Patterns and Traceability in Teaching Software Architecture

Holger Gast
Wilhelm-Schickard-Institut für Informatik
Sand 13
University of Tübingen
gast@informatik.uni-tuebingen.de

ABSTRACT

Courses on software architecture and software engineering need to explain the role of non-functional properties in software design, and they often use student projects to highlight their interrelations. Despite its critical importance in software development, the property of traceability has so far been mostly neglected.

This paper examines the role of traceability for teaching software architecture and describes the objectives and structure of two consecutive courses given by the author at the University of Tübingen. The courses build on architectural patterns as blueprints for achieving particular non-functional properties. The vehicle for improving traceability is a sequence of small-scale projects, each of which includes the phases of requirements analysis, design, and implementation. This iterative approach allows students to learn from their experiences and to integrate the feedback on their solutions into the next project.

Categories and Subject Descriptors

D.2 [Software Engineering]: Miscellaneous

General Terms

Design

Keywords

software architecture, software engineering education

1. INTRODUCTION

Traceability is a central non-functional property of software (e.g. [4, 23]): in order to produce maintainable, adaptable, and extensible systems, the requirements must be linked, through the system architecture and the design of individual components, to the source code. The resulting consistency of the overall system ensures that the system can be modified in the future without inadvertently breaking existing structures and mechanisms. Furthermore, traceability aids

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ 2008, September 9–11, 2008, Modena, Italy.
Copyright 2008 ACM 978-1-60558-223-8/08/0009...\$5.00

communication among developers, since new team members or maintenance programmers can be confident that understanding single requirements and components will be sufficient for contributing to the overall system.

Besides its established role in professional software engineering, a focus on traceability can also help students to become better software engineers. The first case in point is the relation between traceability and communication skills. Many approaches to teaching software architecture, software engineering, or object-oriented programming use projects worked on by teams of students [18, 24, 17, 2, 14, 22, 20]. They emphasize the need for communication between different stakeholders and the negotiation of technical decisions between architects and developers. The central concern in such situations must be efficiency and precision of communication. Requiring students to trace their architectural and design decisions to the source code forces them to express the concepts that underly their software consistently and succinctly.

A second motivation of enforcing traceability in course work is that even senior students find it difficult to view their software at different levels of abstraction. However, it is a well-known fact (e.g. [30]) that software design requires frequent changes of perspective, alternating between a high-level conceptual view of the overall system structure and a low-level implementation view for establishing feasibility of the proposed solutions. Maintaining mental connections between the different layers of abstraction is a prerequisite, and traceability is a technical, and therefore concrete and teachable concept that helps to fulfill it.

A third benefit of requiring students to trace their design to the source code is that they get immediate feedback about their design decisions. If parts of the implementation are not clear and well-structured, they usually point back to an ill-directed design decision. Unfortunately, many students draw the wrong conclusion at this point by trying to solve the evident implementation problems at the implementation level (see e.g. [22, p. 258, item iv]), such that they fail to identify an inconsistent design as the real source of the experienced problems. Maintaining explicit links between each piece of code and the corresponding design elements may help to move into the right direction.

Despite these arguments, traceability has not received much attention as a teaching objective. A notable exception is Mitra et al. [22] who build a software engineering course

on a software process, called SEEMTM, that emphasizes traceability. Their experience is very positive in that students appreciated the clear structure of the process and could complete their term projects successfully. However, their approach is tightly coupled to the particular process chosen, the term project's structure resembles an example project discussed in the lecture, and the group of students was small. All three points ensure that the instructors can give constant feedback to students. We are going to investigate where traceability occurs implicitly in general software design methodologies, and show how teaching traceability scales to larger, more complex, and more technical courses.

The above considerations form the basis for the project work in two consecutive courses on software architecture given by the author at the University of Tübingen in Fall 2006 and Spring 2007. They were targeted at master-level students with some experience in object-oriented programming in Java, but without extensive exposure to project work or software design. Differing from most courses described in the literature, ours was not small: over 90 students expressed their interest in participating, 79 submitted a solution to the first project and 65 received a grade for first course, 45 for the second course.¹

This paper presents the contents and organization of the courses and highlights the role of traceability therein. Section 2 outlines the content and motivates the chosen technical view on software architecture. Section 3 describes the student projects in some detail in order to highlight the increasing level of expertise that students needed to acquire in order to solve the assignments. It also points out how the organizational framework of the course can contribute to the established goals. Section 4 summarizes the lessons learned in the courses. Section 5 concludes.

2. COURSE CONTENT

Our courses on software architecture are characterized by a strong emphasis on architectural and design patterns, as well as frequent and detailed references to existing libraries from the Sun JDK [28]. While patterns provide students with well-developed structures for their own designs, the presentation of concrete implementations helps them appreciate this structure on the basis of concrete examples. Section 2.1 motivates our approach to architecture through patterns; Sections 2.2–2.7 present the major topics of the course.

2.1 A Technical Perspective

The field of software architecture encompasses two complementary views on system development. The first view, exemplified by the influential text by Bass et al. [5], emphasizes the architect's task of making early decisions about the overall system structure in such a way that both functional and non-functional requirements are fulfilled and the conflicting interests of different stakeholders are reconciled. Corresponding courses [20, 18, 2, 22] usually simulate work on a real-world, if simplified [29] project in the classroom. The second view focusses on the software design and describes the software system as a set of interacting compo-

nents [16]. A number of courses [9, 24, 6, 7, 17, 14] that treat general software engineering or object-oriented programming, rather than specialized topics of software architecture, have used this view to introduce the students to the notion of software design through concrete project work that leads to a running system.

We have taken the second view one step further by basing our courses on established architectural patterns [11, 25], with supplemental design patterns [15] and Java-specific material [19] where appropriate. For the two-semester course, we have selected 22 patterns and a number of coding idioms, in particular in the realm of concurrent programming. The presentation of the patterns follows the structure advocated in [15, 11], which ensures the link to the first view on software architecture: the benefits and liabilities of architectural patterns clearly state the non-functional properties that are influenced by applying the pattern.

A pattern description also includes an example implementation. Instead of a cut-down toy example, we have found it useful to examine existing, concrete projects that use the pattern. In many cases, good candidates could be found in the API of the Sun JDK itself. Focussing on libraries rather than standalone products has the advantage that the software design of libraries is by nature more general than the solutions to specific problems, and thus exhibits the pattern more clearly. Furthermore, students acquire useful practical skills on the way as they learn to use the library according to the intention of its designers.

The rationale behind our focus on technical aspects of software architecture is two-fold. First, only very few students will be actively involved in architectural decisions in their professional careers, most will work within a given system architecture. Nevertheless, also developers need to understand the overall system structure in order to achieve the task at hand with minimal effort and to design their classes consistently with the system's architectural style. Second, the decisions involved in creating a software design, let alone a software architecture, must be firmly grounded in implementation experience. Even senior professionals find it difficult, for example, to predict the performance-related non-functional properties of a system without building a prototype. At a lower level, the design of a system is necessarily based on the given technical constraints such as the available libraries. Both concerns can be addressed by introducing the students to patterns, since these capture both design and implementation experience of professional architects, have been applied already in numerous existing systems, and clarify their benefits and liabilities.

Finally, we point out the connection between patterns and traceability in teaching. The aim of traceability is greatly simplified through the use of patterns, compared to a design created from scratch by the students. Students only need to link the requirements to the choice of a particular pattern, not its detailed structure. That structure is given, and it only needs to be traced the implementation. Patterns therefore act as building blocks for traceable decisions. Conversely, the aim of traceability also aids in teaching patterns: most benefits of patterns [15] are invalidated if either the wrong pattern is chosen for a particular problem or the pat-

¹Due to the structure of the German diploma programme, drop-outs are very frequent, because there is no penalty for leaving a course.

tern is implemented incorrectly. Forcing students to make these connections explicit helps in avoiding mistakes.

2.2 Responsibility-driven Design

It is a well-known fact that good programmers need a “theory of programming”: it is not sufficient to understand the exact meaning of a programming language; a programmer has to reason about the program structure and the process of programming. A popular approach that has proven very successful in introductory courses to object-oriented programming [6, 14], is to use objects and classes as models of reality. For the purposes of the master-level courses described in this paper, this approach is inappropriate. As Wirfs-Brock and McKean [30] point out, the mechanics of software systems differs so markedly from the behaviour of real-world entities that a resemblance can be at best a guiding principle and a vehicle for communication with users and customers, but not a self-contained basis for software design.

Responsibility-driven design [30] proposes the metaphor of a software system as a community of objects that collaborate to implement the system’s functionality. Even though some objects, called the *domain objects*, may correspond to real-world entities, the focus remains on objects as software artifacts. The design process is iterative: the designers start with *candidate objects* that are assigned individual responsibilities, each of which contributes a particular, logically coherent fragment to the overall functionality. Whenever an object requires a computation step that is part of another object’s responsibilities, it collaborates with that object. Whenever the responsibilities of a candidate object become logically incoherent, or are found to require a large implementation, a re-design is indicated, in which new candidates are introduced, responsibilities are re-distributed, and collaborations are re-evaluated.

Responsibility-driven design has several immediate attractions for the teacher. Budd [10] demonstrates with several examples that responsibility-driven design appeals very strongly to the intuition of the designer, while maintaining the link to the implementation level at all times. Furthermore, we have found that the emphasis on graphs, rather than hierarchies, of collaborating objects helps students to leave behind the mechanics of method calls and procedural solutions in favor of a message-passing style of programming. Next, CRC cards (e.g. [30]) are inexpensive and very concrete representations of candidate objects. Logical connections between objects can be indicated by placement, and collaboration scenarios can be simulated by moving cards around. Finally, the concepts of responsibility-driven design match directly with the elements-and-interactions view on software architecture [16], and its terminology can be found throughout the software engineering literature, in particular in pattern descriptions [15, 25, 11].

Responsibility-driven design is also eminently suitable to teaching traceability, since its concepts have one-to-one correspondences with the source-code level. Beyond the obvious matches for objects and classes, we have found the concept of *roles* to be a useful foundation for explaining the correct use of Java interfaces: many, especially advanced students tend to over-design their first object-oriented programs in order to create “abstractions” and to introduce “flexibility”.

The interfaces defined for this purpose are, in fact, derived from a specific implementation, and have therefore only one concrete class implementing them. Forcing students to specify explicitly the responsibilities of *any* object implementing the interface helps discern real and useful abstractions from the technical, shallow generalizations that merely introduce complexity without providing benefits. Using roles for explaining Java interfaces also complements the pedagogical considerations by Schmolitzky [26] for introducing interfaces before inheritance.

Beyond enabling the technical link between design and code, responsibility-driven design also emphasizes the traceability from requirements to design. The design process proposed by Wirfs-Brock and McKean [30] explicitly requires allotting the responsibilities for given requirements to objects, and to validate the design by checking that the objects’ collaborations will finally implement all requirements. With this last step, the process enables complete traceability from requirements analysis to source code.

2.3 Interactive Systems

Since most software systems today include a graphical user interface, we have chosen their basic software structure as the first topic in the course. Graphical user interfaces also have the advantage that the concrete representation on the screen can be traced directly to the design and implementation, which supports understanding and is very motivating for the students.

The central pattern for this application area is, of course, the MODEL-VIEW-CONTROLLER (MVC) pattern [11]. An OBSERVER pattern [15], embedded inside the MVC, connects the model with the views to keep the screen representation up-to-date. Most courses that treat graphical user interfaces stop at this point (e.g. [22, 24]). However, the field offers many relevant patterns beyond the basic ones, and we have treated them: the VIEW HANDLER [11] introduces a component that maintains the relations between views and models; its main responsibility is the management of both model and view life-cycles. COMMAND PROCESSOR [11] explains how undo/redo mechanisms can be implemented on top of the MVC, using COMMAND [15] to represent user actions and possibly MEMENTO [15] for storing extracts from previous model states without breaking encapsulation. In order to present a different view on the area, the lecture, but not the projects, also included the PRESENTATION ABSTRACTION CONTROL (PAC) pattern [11].

2.4 Extensibility

The second broad topic covered in the first semester was extensibility. In the context of maintainability and adaptability, software engineers have always emphasized this non-functional property. With the success of the Eclipse IDE [13], the importance of plugin mechanisms has been commonly accepted. It is therefore a natural question to ask how extensibility can be achieved by appropriate software structures.

Extensibility is a particularly difficult subject to teach. It requires designing a family of software systems, rather than a single system, and to capture the commonalities of all desirable extensions without having a definite and complete

list. In short, extensibility requires inventing abstractions, which in turn can only be drawn from extensive experience with similar systems.

We have started with the classical PIPES&FILTERS and LAYERS [11]. Both are not only patterns, and therefore solutions to specific design problems, but also general architectural styles [16] underlying many software systems. Because of their relative simplicity, they can serve to demonstrate the requirement of defining general, abstract interfaces with which extensions can collaborate. We have chosen the INTERCEPTOR pattern [25] as the central structure of extensible systems, since it explains how to extend a given, and often substantial framework with new services that can be implemented in comparatively small modules. It is also substantiates the claim that extensibility requires very abstract design: the behaviour of the framework is represented by a finite state machine, whose transitions trigger notifications of the registered interceptors. An analysis of the request processing of the Apache web server [12] has made the general setting more concrete. There, the FSM is reduced to a linear sequence of processing steps, and the interceptors can register to perform one or more of these steps.

2.5 Concurrency

The second semester of the course focuses on performance-related non-functional properties such as scalability, latency, and throughput. Its overall topic is concurrent and parallel execution as a fundamental prerequisite to attaining these goals. The treatment is based on [19, 25].

The starting point are the concurrency control mechanisms built into the Java language, i.e. interference control by `synchronized` blocks, thread synchronization via the methods `Object.wait()`, `Object.notify()`, and thread control via `Thread.interrupt()`. Building on these, we have treated the utility classes from the `java.util.concurrent` package of the JDK. This also includes presentations of the classical concepts of mutexes, semaphores and solutions to producer/consumer problems with synchronized queues.

To make the complexity involved in concurrent and in particular parallel programming very explicit, we have also given one lecture on formal reasoning about concurrent programs [3] and one on the Java memory model [21].

2.6 Encapsulating Concurrency

Systems that use concurrent execution are hard to design and hard to implement correctly. It is therefore desirable to introduce abstractions that shield the larger part of the system from considerations on concurrent execution, while still maintaining its benefits.

The ACTIVE OBJECT pattern [25] achieves a hidden concurrent execution by means of a PROXY [15] object: calling the proxy's method internally triggers an asynchronous, possibly concurrent execution of the corresponding method in a service provider object. Using ACTIVE OBJECT also requires to handle the results of asynchronous operations, which is explained by FUTURES [25], COMPLETION CALLBACKS [19], and ASYNCHRONOUS COMPLETION TOKEN [25]. A second, related way is given by lightweight executable frameworks [19], which allow the application to submit *task* objects for

later execution. This concept can again be made concrete by the `Executor` framework from the JDK.

2.7 Asynchronous Processing

After mastering the technicalities of concurrent execution, we have treated the problems of latency and throughput by considering the overhead that classical concurrency control mechanisms incur.

The REACTOR [25] captures a possible structure of scalable high-performance servers. The reactor itself is single-threaded to avoid the performance penalties of locking while receiving client requests, but it is desirable to execute the requests concurrently. Even though an ACTIVE OBJECT achieves this goal, it requires synchronized access to a request queue. The LEADER/FOLLOWERS pattern [25] addresses this concern. The lecture also presents a concrete realization of a chat server that broadcasts incoming messages to all connected users. Using Java's NIO *channels* [28] introduces the students to an existing infrastructure that allows them to build reactors. While the REACTOR describes the efficient handling of requests arriving from multiple clients, the PROACTOR [25] pattern addresses the complementary problem of dispatching requests to multiple servers asynchronously and receiving the results efficiently. Finally, we have presented optimistic concurrency control in the form of a transaction framework [19].

This final and most complex topic of the course also makes the combined benefits of patterns and traceability for teaching most visible, since the underlying considerations and structure of a reactor-based system cannot be understood by considering the technical foundations alone. For instance, even Stevens' very detailed and most readable presentation [27] of the `select` function, which is a possible technical basis for implementing a REACTOR, cannot convey how the shown toy examples scale up in a real server, and Stevens gives scarcely any motivation for preferring the complex `select` solution over the simple blocking `accept/fork` solution. Precisely the missing arguments are obtained by tracing the non-functional requirements of scalability and low latency to the choice of the REACTOR pattern, and from there to the concrete implementation.

3. THE PROJECTS

Project work is a central ingredient in many courses on software engineering and object-oriented programming [14, 24, 17, 22, 2, 18, 20]. This section motivates our approach of using several small-scale, rather than one large-scale project, and describes the assignments.

3.1 Considerations on Assignments

The standard approach in project work is to set a single assignment that is solved in stages through the semester, mostly in parallel with the software process taught in the lecture. Since our course is not process-centered, we have decided to use several small-scale projects of increasing complexity and with different types of systems. In this setup, students can gain experience with a greater number of patterns. Ideally, the patterns that they use in the projects have just been covered in the lecture. Furthermore, elementary design-problems will occur repeatedly and will be learned

more easily by students (see also EARLY BIRD and SPIRAL in [8]). A second motivation for a repetitive approach is that the students have the chance to make mistakes, to discuss them with the teaching assistants, and to avoid them in the subsequent projects (see also MISTAKE in [8]). As the first project can be solved by elementary means, students also get a system to run early, which is always a very rewarding and motivating experience.

The next question is how to set the assignments. Since the course content focuses on architectural patterns, we have chosen to let students re-implement well-known applications. This decision reduces requirements analysis to choosing features, and focuses the project work on the main teaching objectives of software design and implementation. Students have also told us that they were motivated precisely because they had already seen a similar application, and felt challenged to emulate or even exceed its functionality.

3.2 Organization and Grading

The projects are worked on by teams of 3 or 4 students in in about 3–4 weeks, depending on the expected size of the solution. Since all students are expected to cover the entire material, there is no distinction between different roles within a team. The small size of teams was chosen to enable effective assessment of the individual contributions. The work proceeds in two phases: in a 1-week analysis and design phase, the students decide what functionality should be implemented and create an initial design using CRC cards. They present this design to a teaching assistant who gives feedback on possible flaws and the correct application of patterns. It is important to point out that no grade is given for the design at this point, because we want to encourage students to learn about software design from their implementation experiences (see MISTAKE in [8]). The remaining 2–3 weeks are spent on the implementation and documentation. The students hand in the source code and a 2-page project description. The project description presents the final design and indicates how it is implemented in the source code. Furthermore, any changes to the original design have to be highlighted and justified.

We have found this latter traceability requirement very important for practical purposes: it forces students to think about how the teaching assistant can understand the code they have written by relating code and design. In this manner, we have been able to teach 75 students, i.e. 20 teams, with three teaching assistants, the instructor getting an equal share of teams.

Grading criteria must match the teaching objectives to be effective. The major teaching objective is delivering understandable source code that corresponds directly to the design, and a design that matches the given requirements. There are four possible grades: a “+” is given for understandable, clearly designed solutions that employ patterns correctly, and in which the project description clarifies the links between design and implementation. A “√” indicates that the design and implementation use the concepts from the lecture adequately. A “–” indicates that the software is not acceptable, because it violates responsibility-driven design principles, or because patterns are applied incorrectly. Finally, a “fail” is given if fundamental concepts of object-

oriented design, such as encapsulation of fields, are violated in several places. Over the semester, the students had to compensate “–” grades with “+” grades, but were allowed one remaining “–”.

3.3 MailViewer

The *MailViewer* application is a simple, light-weight tool for inspecting the content of standard UNIX mail boxes. Its task is to read in a mailbox file and to display a summary table of all mails with subject, sender, and date. When the user selects a mail, the header fields and mail body are to appear in a separate window.

The *MailViewer* project serves as an introduction to the Swing library [28]. Extracting the headers and bodies from a mail box file is straightforward. The students therefore can concentrate on reading the API documentation and applying elementary Swing mechanisms for `ListSelectionListeners` and `ActionListeners`. The `JTable` component with a `DefaultTableModel` is sufficient for this project. The lecture has at this point provided examples on Swing, such that the students obtain an immediate success in creating their own user interface. Finally, students make the experience that their designs are very much constrained by the available libraries, and that existing mechanisms and abstractions should be exploited where possible (see also [1, Section 6.1]).

3.4 Calculator

The next project is a simple desktop calculator. The user enters digits by buttons, and the digits form a number in a display component. The four basic arithmetic operations are also triggered by buttons. When the user enters 4, 2, and +, for example, the number “42” is stored as a temporary result and the operation + is stored to be executed as soon as the next number is complete. Advanced students are challenged to implement operator precedences and parentheses.

This project clearly requires responsibility-driven design, as none of the obvious components can provide the functionality on its own: each button enters only a single digit or operation symbol, but they do not have sufficient information to carry out any operations. Note also that building an object-oriented model of the calculator is not enough, since objects that store temporary results and the already typed operation symbols are software artifacts that do not have a real-world counterpart. Many teams also came up with a central object that coordinated the services of the others, thus finding the MEDIATOR pattern [15] by themselves.

3.5 MiniXcel

MiniXcel is a spreadsheet application. It displays a table in which the single cells contain formulae that can contain references to other cells of the table. Each cell therefore has a current value that is computed from the stored formula by reading the values of the referenced cells. The computation stops when a cell contains a constant formula, it fails when cyclic references are found. When the user updates the formula of one cell, its value is re-computed and all dependent cells need to be re-computed as well.

The project starts when the MVC pattern has been introduced in the lecture. The main design goal is to ob-

tain a strict separation between the model of the spreadsheet, which manages formulae, cells, and references between cells, from the user interface, which merely displays the current value of each cell and allows the user to edit formulae in a separate `JTextField`. A parser for formulae was provided. The requirement of using a `JTable` also forced students to implement their own `TableModel` and to notify `TableListeners`, in particular the `JTable`, about any changes. Using the already fixed structure of Swing's variant of the MVC helps students to obey a strict model/view separation, since this separation is already manifest in Java `interface` definitions.

In a second 2-week phase, the students extend their spreadsheet model with Undo/Redo functionality using the `COMMANDPROCESSOR` pattern, and possibly a `MEMENTO` to store state information. By extending their already understood software, students can see the relations between the MVC and the `COMMANDPROCESSOR` clearly.

3.6 Formula Editor

Many word processors today feature a graphical editor for mathematical formulae. A reduced version for the project handles formula with fractions and parentheses that grow to the size of the contained formula. The user can manipulate the formula interactively: a cursor indicates the current position and typing or deleting elements changes the layout immediately. Fractions and large parentheses are inserted using toolbar buttons, other characters are just typed.

This project introduces two typical complexities connected with the MVC pattern. First, the view component has to store the layout in a data structure that mirrors the structure of the model. This stored layout data enables the view to position the cursor and the controller to interpret keypresses and mouse-clicks. Second, it is necessary to find for each kind of formula element in the model a corresponding view element that can perform the layout. Since the model is to be independent of the view, the necessary mapping leads to a natural application of `ABSTRACT FACTORY` [15]. To simplify the assignment, we allowed students to use `FACTORY METHOD` [15] instead.

3.7 Desktop Publishing

The final project is a desktop publishing tool that is extensible by plugins. The overall idea is that a *page* contains general *elements*, and plugins can provide new types of elements. For the project, elements always occupy rectangular areas on the page and do not interact with one another. The user can manipulate the position and size of elements interactively and can also edit the content of elements.

The project demonstrates the complexity of achieving extensibility (Section 2.4). The students have to implement a core framework, and to define roles and corresponding interfaces for models and views of page elements. A class that loads classes from JARs was provided. The goal is a minimal solution with two plugins, one providing text elements based on a `JTextPane`, the other implementing picture elements based on a `JLabel` painting an `Icon`. The main challenge is to design the collaboration between the framework and the plugins for different events.

3.8 WWW Link Graph

The second semester of the course, especially the Java thread semantics and the foundations of concurrency (Section 2.5), contains much technical material and concepts that students are not familiar with. In particular, the fact that the scheduler decides on the order of the execution of individual JVM instructions is in stark contrast with the deterministic execution model taught in introductory courses. We have therefore decided to have only two projects, and to use weekly assignments to prepare the students for the projects. The grading of these weekly assignments was based on a percentage of points, rather than the four categories of the first semester.

The first project is a program that constructs a graph in which the nodes are WWW sites and an edge (A, B) exists if one of the pages from site A contains a link to one page from site B . The program is to trace links between HTML pages to discover edges. In order to optimize the overall runtime, several downloads should occur simultaneously, and links from an HTML page should be extracted concurrently with the downloads.

The assignments before the project treat the Java language mechanisms and the utility classes from the library for thread synchronization. The project itself is then solved by elementary responsibility-driven design in order to understand the structure. In a second step, the students refactor their solutions to use the `ACTIVE OBJECT` pattern.

3.9 File Server

The second project addresses the `REACTOR` pattern for scalable systems. The goal is to implement a simple file server resembling an FTP server. In the weekly assignments before the project, the students implemented an abstraction *message*, such that message objects could be received and sent by the reactor. Any message contains a fixed-size header, whose structure varies with the message type, and an arbitrary-length body of byte data. The main objective in the project was to make all socket-related operations non-blocking in order to avoid stalling the server in case one client failed to send data or accept the transmitted data.

4. EVALUATION

This section summarizes the insights gained from teaching the course. We derive them from two sources: the feedback given by students and the experiences reported by the teaching assistants.

4.1 Student Feedback

There have been two evaluations of the courses: the official survey conducted by the department and a set of specific feedback questions prepared by the author. The nature of the survey questions allows only deductions on the quality of the course. Each question can be answered on a scale from 1 to 5, where 1 is positive and 5 is negative; the students may also choose "not applicable". Figure 1 below shows the detailed counts for those questions connected to the course organization. From the average values, we conclude that students approved of the general quality of the courses (A). In particular, they found that the assignments were strongly related to the content of the lecture (B) and have helped in

repeating and understanding the concepts (C). They also felt that they had learned something useful and important (D). Even though the second course appears to have been judged slightly better, no strong conclusions about the organization should be drawn: only those students who approved of the first course would continue in the second one.

The final question (E) concerns the students’ feeling about the adequacy and fairness of the grades assigned to their solutions. Here, a substantial change in the quality of answers occurs between the courses. Even if we assume that those answering 4 and 5 did not attend the second course, the relative weight between 1 and 2 is reversed. We conclude that the strict division between “acceptable” and “unacceptable” solutions introduced in the first course (Section 3.2) was perceived less fair than the more fine-grained grades in the second course (Section 3.8).

Question	First Course					Avg	Second Course			
	1	2	3	4	5		1	2	3	Avg
A	26	14				1.39	15	3	3	1.17
B	23	15	2			1.48	11	7		1.39
C	28	10	2	1		1.41	13	6		1.32
D	34	9	1			1.25	16	3		1.16
E	9	17	4	1	2	2.09	10	6	1	1.47

Figure 1: Survey Results

The specific feedback questions had a positive and a negative form and were meant to elicit good and bad practices for teaching object-oriented programming in general and patterns in particular.

- Which three points did you like/dislike about the assignments?
- What has/would have helped in learning and using patterns?
- What has/would have helped in learning object-oriented design?

Since answers were to be given in textual, non-standardized form, statistics are not available. For the first pair of questions, most students appreciated teamwork on concrete and obviously “useful” projects as an essential ingredient. Several pointed out that developing their ideas on their own into a concrete system was very motivating and gave them confidence in their abilities. It is interesting that many preferred the more complex and also more time-consuming project assignments of the first course to the more confined worksheets introduced in part two.

We can summarize the answers for the second and third pair of questions, because they turned out to be similar. A large majority of students agreed that detailed feedback from teaching assistants and the concrete implementation experience were essential for learning both patterns and general object-oriented design. Planning projects with patterns, developing CRC-cards for pattern, and writing a design description were further answers. One student also commented on the small-scale projects: “Many new examples in projects,

in which one could always try anew to make the design more perfect.”

The overall result is that students needed to follow the concepts presented in the lecture to concrete source code in order to understand them. The need for discussions at every stage shows that neither the step from patterns to a concrete design, nor the step from a complete design to the source code are straightforward for the learning software engineer. Even though no student mentions traceability explicitly, their preferences show clearly that they appreciated the opportunity to see how their initial solutions would turn out in the concrete implementation, and to connect the concepts with the structure of the source code.

4.2 Observations by the Teaching Assistants

The teaching assistants and the instructor met on a weekly basis to discuss the progress of students and in particular the performance in projects. The most important problem that occurred was that students found it difficult to implement the patterns presented in the lecture directly, even though example implementations on the basis of the JDK library had been given. Students also could not explain their design properly without recurring to the concrete implementation details. A most motivating feedback, on the other hand, was the immense zeal that the students showed in solving the project assignments, in particular the application programs in the first part of the course.

We found, however, that through the semester students got more “professional” in dealing with design presentations and subsequent implementation work. Where the first solutions needed much re-structuring and students were insecure about their designs, the later projects were approached with much more confidence and some overview and even discussions about alternative designs.

Again, these observations argue for a greater emphasis on traceability in teaching patterns: if most students find the task of transforming patterns to source code hard, then it is better to confront them with this complexity during their university education rather than on their first job, where the quality of the produced code may have a direct impact on the success of the project they work on.

A final remark concerns the grading scheme from Section 3.2. In general, it has fulfilled its goal in forcing students to deliver solutions acceptable by software engineering standards. However, the teaching assistants often found it hard to differentiate between “√” and “-” grades, because a “-” had a severe impact on the team’s chances of passing the course. If students are to be allowed to make mistakes, as postulated in Section 3.1, then mistakes remaining in the solution must not have too severe consequences. This observation coincides with the students’ feedback about grading discussed in Section 4.1. In conclusion, a better grading scheme would define firm lower standards for acceptable solutions, but beyond this would be flexible to capture differences in quality in a fine-grained manner.

5. CONCLUSION

We have presented a two-semester master-level course on software architecture with a strong focus on architectural

and design patterns. It covers topics ranging from basic responsibility-driven design, over interactive and extensible systems, to architectures for scalable network servers. The software structure of these systems has been presented in the form of 11 architectural and 11 design patterns. Most example implementations of patterns have been taken from the Sun JDK. The course has been taught in Fall 2006 (79 participants) and Summer 2007 (45 participants) and will be repeated starting in Fall 2008.

The assignments are organized as 7 projects to be solved by teams of 3–4 students. Each project starts with a 1-week analysis and design phase, followed by a 2–3 week implementation phase. After one week, the students present their design to a teaching assistant and receive feedback. At the end of each project, they hand in a 2 page project description and the source code of the running system.

The main insight gained from the course is the strong connection between patterns and traceability in teaching. Students report that they have learned patterns best by applying them in a concrete project, by discussing the resulting design with their team members and the teaching assistants, and by implementing the design in a running system. In other words: they found it necessary to trace the pattern structure to the design, and from there to the implementation in order to grasp the details of the pattern.

The second observation is that multiple small-scale projects are a viable way of teaching a complex topic. Through the repetition of the development process, students get more and more confident about the expected outcome of each phase. They also can try their hand at a design, learn from the experience of implementing it, and improve their next design with this new knowledge.

Acknowledgement. I am greatly indebted to my teaching assistants, Monika Kochanowski, Julia Trieflinger, and Daniel Faber, for their eagerness to discuss and elaborate software designs, and to support students in attaining the course goals. With their own expertise and overview in software design, they have greatly helped to clarify the concepts of the lecture and have expertly supervised the concrete realization in student projects.

6. REFERENCES

- [1] ABBOTT, R., AND SUN, C. Abstraction abstracted. In *ROA '08: Proceedings of the 2nd international workshop on The role of abstraction in software engineering* (New York, NY, USA, 2008), ACM, pp. 23–30.
- [2] ALRIFAI, R. A project approach for teaching software architecture and web services in a software engineering course. *J. Comput. Small Coll.* 23, 4 (2008), 237–240.
- [3] ANDREWS, G. R. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.
- [4] ANTONIOL, G., CAPRILE, B., POTRICH, A., AND TONELLA, P. Design-code traceability for object-oriented systems. *Ann. Softw. Eng.* 9, 1-4 (2000), 35–58.
- [5] BASS, CLEMENTS, AND KAZMAN. *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.
- [6] BENNEDSEN, J., AND CASPERSEN, M. E. Programming in context: a model-first approach to CS1. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education* (New York, NY, USA, 2004), ACM, pp. 477–481.
- [7] BENNEDSEN, J., AND CASPERSEN, M. E. Teaching object-oriented programming – towards teaching a systematic programming process. In *Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (Oslo, Norway, June 14, 2004)*. Affiliated with 18th European Conference on Object-Oriented Programming, *ECOOP 2004* (June 2004).
- [8] BERGIN, J. Fourteen pedagogical patterns. <http://pplc.pace.edu/~bergin/PedPat1.3.html>.
- [9] BUCCI, P., LONG, T. J., AND WEIDE, B. W. Teaching software architecture principles in CS1/CS2. In *ISAW '98: Proceedings of the third international workshop on Software architecture* (New York, NY, USA, 1998), ACM, pp. 9–12.
- [10] BUDD, T. A. *An Introduction to Object-Oriented Programming*, 3rd ed. Addison-Wesley, 2002.
- [11] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-oriented Software Architecture: A System of Patterns*, vol. 1. Wiley & Sons, 1996.
- [12] DOC, A. A. Request processing in apache 2.0. <http://httpd.apache.org/docs/2.0/developer/request.html>.
- [13] The Eclipse workbench. <http://www.eclipse.org>.
- [14] FERREIRA, G. M., NASCIMENTO, M. Z., ASSIS, K. D. R., AND RAMOS, R. P. Teaching object oriented programming computer languages: learning based on projects. In *ICSEA '07: Proceedings of the International Conference on Software Engineering Advances* (Washington, DC, USA, 2007), IEEE Computer Society, p. 81.
- [15] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
- [16] GARLAN, D., AND SHAW, M. An introduction to software architecture. Tech. Rep. CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, 1994.
- [17] JARZABEK, S., AND ENG, P.-K. Teaching an advanced design, team-oriented software project course. In *CSEET '05: Proceedings of the 18th Conference on Software Engineering Education & Training* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 223–230.
- [18] LAGO, P., AND VAN VLIET, H. Teaching a course on software architecture. In *CSEET '05: Proceedings of the 18th Conference on Software Engineering Education & Training* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 35–42.
- [19] LEA, D. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Addison-Wesley, 1999.

- [20] MÄNNISTÖ, T., SAVOLAINEN, J., AND MYLLÄRNIEMI, V. Teaching software architecture design. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 117–124.
- [21] MANSON, J., PUGH, W., AND ADVE, S. V. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), ACM Press, pp. 378–391.
- [22] MITRA, S., RAO, T. M., AND BULLINGER, T. A. Teaching software engineering using a traceability-based development methodology. *J. Comput. Small Coll.* 20, 5 (2005), 249–259.
- [23] MURTA, L. G., HOEK, A., AND WERNER, C. M. Continuous and automated evolution of architecture-to-implementation traceability links. *Automated Software Engg.* 15, 1 (2008), 75–107.
- [24] NOONAN, R. E., AND HOTT, J. R. A course in software development. *SIGCSE Bull.* 39, 1 (2007), 135–139.
- [25] SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. *Pattern-oriented Software Architecture: Patterns for concurrent and networked objects*, vol. 2. Wiley & Sons, 2000.
- [26] SCHMOLITZKY, A. Teaching inheritance concepts with Java. In *PPPJ '06: Proceedings of the 4th international symposium on principles and practice of programming in Java* (New York, NY, USA, 2006), ACM, pp. 203–207.
- [27] STEVENS, W. R. *UNIX Network Programming*. Prentice-Hall, 1990.
- [28] SUN. Java2 Standard Edition API. <http://java.sun.com/j2se/1.5.0/docs/api/>, 2004.
- [29] VAN VLIET, H. Reflections on software engineering education. *IEEE Softw.* 23, 3 (2006), 55–61.
- [30] WIRFS-BROCK, R., AND MCKEAN, A. *Object Design: Roles, Responsibilities, Collaborations*. Pearson Education, 2003.