Analyzing Students' Software Redesign Strategies

Sylvia Stuurman
Open University
the Netherlands
Sylvia.Stuurman@ou.nl

Harrie Passier
Open University
the Netherlands
Harrie.Passier@ou.nl

Erik Barendsen
Radboud University & Open
University, the Netherlands
e.barendsen@cs.ru.nl

ABSTRACT

The design of software is known to be difficult for novice computer scientists. In this paper, we focus on software redesign and on the refactoring necessary to implement a redesign. Redesigning an application aims to improve non-functional aspects such as extensibility, without changing the functionality. Redesign is a complex task, involving knowledge and skills from software design in general and the use of design patterns in particular. This study is part of an educational design research project aiming at developing scaffolding for students' software redesign activities in the form of procedural guidance. We investigated students' strategies and usage of concepts during a software redesign assignment using students' reports and team colllaboration recordings as data sources, thus focusing on the process instead of on the design results, in contrast with existing studies. We identified several difficulties that can serve as starting points for procedural guidance. For instance, students seem to avoid using a structured analysis method. Our findings indicate that students' activities were mainly directed towards the code rather than the design problem.

CCS Concepts

•Social and professional topics → Computer science education; •Software and its engineering → *Designing software;*

Keywords

Redesign; Refactoring; Design patterns; Procedural guidelines; Education

1. INTRODUCTION

Designing software is a complex task, requiring knowledge about programming and design as well as problem solving skills. Design problems are complex in the sense that these involve many interrelated concepts on various levels of abstraction. Moreover, design problems are usually ill-structured and therefore require reasoning with incomplete knowledge [22]. In order to carry out a design task in an effective and structured way, students need to be able to abstract from a particular case and view it as an instance of a general problem [44].

These observations are consistent with research findings about students' difficulties. Many graduating students turn out to be unable to create a proper software design [14, 25]. Students struggle with the application of concepts such as abstraction and polymorphism [28] and information hiding [16]. These findings suggest that a form of *scaffolding* of students' learning is in order, so the question arises what kind of scaffolding is appropriate.

One particular method to implement scaffolding is to reduce the degrees of freedom in a task: to put constraints on the task, so students know where to focus [30]. A second form of scaffolding is to provide a student not only with conceptual knowledge, but also with *procedural guidance* regarding the steps to take to solve the problem, and ways to recognize an acceptable solution [12, 24].

As students often indicate that they are uncertain about how to apply the conceptual knowledge while solving design tasks, we focus on developing procedural guidance as a scaffolding approach. Such guidance appears to be scarce in traditional textbooks.

Our study is part of an *educational design research* [31] project aiming at developing procedural guidance strategies and suitable teaching materials for software design. The project is carried out in the context of a Master's program at the Open University in the Netherlands. In this project, we focus on the increasingly relevant subtopic of software *redesign*.

This study can be viewed as the first analysis step in our educational design research. In order to identify students' procedural difficulties as input for our development of scaffolding materials, we analyzed students' *strategies* while working on a redesign assignment. In this respect the study complements existing research analyzing the students' software design *products* or conceptual *understanding* [43, 15]. Instead, we focus on the design *process*.

As students' strategies have not been extensively investigated in the context of software design (cf. [2] for engineering) we chose to conduct a qualitative and small-scale exploratory study. Moreover, our aim was to experiment with 'non-obtrusive' data sources, that is, using data that was available in the course without additional instruments or interventions.

Koli Calling 2016, November 24-27, 2016, Koli, Finland

DOI: http://dx.doi.org/10.1145/2999541.2999559

In the remainder of this section, we discuss the conceptual background of our study, as well as related research on students' difficulties concerning software design.

1.1 Software design and redesign

With respect to the problems that students face while designing a software system, it has been observed that they do not try to describe the desired behaviour of the system to design: their analysis of the problem is far from complete. Not analyzing the problem well enough appears to be the main problem in designing the desired system [25].

The differences between novice and expert programmers have been researched extensively. Experts have a top-down, breadth-first approach to decomposition and understanding, while novices approach a problem line-by-line, local and concrete [33].

Expert designers seem to be very good at 'problem setting', naming every part or aspect of a problem to be solved. Experts appear not to analyse the problem extensively, but rather scope the problem adequately [10]. More experienced designers tend to state more assumptions and ask for clarification more frequently [2]. Often, experts choose a solution path from which they do not like to deviate, in contrast to outstanding designers (the best among the experts), who like to maintain parallel lines of thought [10]. In a multi-national study, Tenenberg et al. [43] observed a higher recognition of ambiguity and stronger use of non-textual representations among more experienced students.

When design is performed in a team, 'setting' the problem is an additional hindrance for novices, since this requires the designers to use a common vocabulary: "*The task of design is as much a matter of getting different people to share a common perspective, to agree on the most significant issues, and to shape consensus on what must be done next, as it is a matter for concept formation, evaluation of alternatives, costing and sizing – all the things we teach.*" [5].

Problems of students with respect to software design can be categorised using the taxonomy of Soloway [37], who discerns five phases in problem solving in the case of a programming problem: (1) Understand the problem, (2) Decompose the problem, in order to identify the solution components that will solve the problem components, (3) Select and compose plans to solve problems, composing the components to a functioning system, (4) Implement plans into language constructs, (5) Reflect and evaluate the final system and the overall design process.

In programming tasks, novices' procedural difficulties appear mainly in Phase 1 (concerning the ablity to set the problem), in Phase 2 (lack of strategies to decompose the problem into parts), and in Phase 3 (regarding plan composition) [33]. In view of the related research discussed above, we expect similar hindrances in design tasks.

Some experts regard software *redesign* as more complex than software design. Indeed, the starting point is an existing application that has generally been designed by others, often with non-documented requirements and design decisions.

It is helpful to view software redesign as a case of problem solving. A problem solving process starts from a certain *initial state*, while rules help to arrive at a *goal state* [1]. A problem solving strategy is a method to apply rules in a specific sequence.

In the case of a redesign problem, the initial state is special, namely the source code belonging to a working application. In the ideal case, the requirements for the software are explicit. One might also have access to the original description of the problem that the software tries to solve. The *goal state* in the case of redesign is what the developers want to achieve. In general, the goal state is not well-defined, and developers may differ in what they see as the purpose of redesign. In our course, the purpose of redesign is to improve maintainability with respect to future changes, which is not measurable in a direct way.

With respect to rules to reach the goal state from the initial state, one can discern *strong* and *weak* methods. Strong methods are domain-specific, problem-specific, and guarantee a solution. Weak methods are procedures that are imprecise and do not guarantee a solution. Instead, they serve as general-purpose problem-solving strategies or rules of thumb that solvers can fall back on when they do not know what to do directly to solve the problem [32]. An example is the following four-stage problem solving framework: orientation (assess and understand a problem), organization (planning and choice of actions), execution (conform to plans), and verification (evaluation of orientation and organization and evaluation of execution) [20]. Soloway's five phases can also be viewed as a weak method [37].

Giving procedural guidance for a complex task can be seen as providing a strong method for the task at hand. There are not many procedural guidelines for the task of redesigning a system. In fact, only weak methods seem to be available in textbooks, for example [34].

1.2 Refactoring

Refactoring is the process of changing source code to improve its internal structure without changing the external behavior of the code [18, 26]. After a system has been redesigned, refactorings are necessary to make the code compliant with the new design.

Refactorings are described in the form of catalogues [18]. Advices about how to refactor to patterns take the same shape [23]. As for redesign in general, there seems to be a lack of procedural guidance for applying them to code. Moreover, refactoring catalogues tend to include an overwhelming number of refactorings. As a consequence, few courses have refactoring as an explicit learning goal. Teaching a clear sequence of refactorings might help overcoming the hindrances [36, 38].

Mens and Tourwé [26] present refactoring guidelines, but these are highly abstract and therefore can be seen as a weak method: (1) identify where the software should be refactored, (2) determine which refactoring should be applied, (3) guarantee that the applied refactoring preserves behaviour, (4) apply the refactoring, (5) assess the effect of the refactoring on the quality characteristics of the software (in terms of for example complexity and maintainability), and (6) maintain the consistency between the refactored program code and the other software artifacts.

Design smells are often used as indicators for the choice of refactorings in Step (1) [42]. However, the connection between design smells and refactoring does not take the form of guidelines, but is rather presented (again) as a large catalogue of design smells and their remedies.

In our case, the leading force for the refactoring steps is the redesign. Given a specific redesign, however, the particular sequence of refactoring steps leading to compliant code

is not obvious. Apparently, no guidelines exist to find such a sequence.

1.3 Design patterns

Design patterns offer general solutions to general (design and code related) problems [19]. Design patterns are discussed in several textbooks [4, 19, 34].

In postgraduate courses, students appear to find it difficult to relate design patterns to specific design problems, and to explain why certain metrics (for instance regarding low coupling) change as a result of design pattern application. Chatzigeorgiou et al. [6] conjecture that this is due to the fact that most patterns are presented in textbooks and courses as solutions to known implementation problems rather than as a treatment for design quality issues.

Gestwicki and Sun [21] observe that students find it difficult to comprehend patterns from isolated examples (e.g., [21]).

Like refactorings, design patterns tend to be characterized in the form of a catalogue. For each pattern, the applicability is explained in the form of a design problem, or a problem in the code. This makes it difficult for students to determine when and where to apply a pattern to solve a problem. There have been attempts to provide tools to help developers to apply design patterns, for instance, using a recommendation system [29] that checks whether one of the general problems applies by asking the developer questions about the code. In some courses, the task of determining *which* patterns could be applied *where* is scaffolded by providing a framework of interfaces, indirectly prescribing the choice of patterns [13]. An additional problem of design patterns is that it appears hard for students to identify the problems that they can solve [6]. A possible approach is to let students first experience the consequences of an ad-hoc design and afterwards teach them the benefits of a design pattern that forms a general solution to the specific problem [35].

To support understanding the problem, decomposing the solution into parts and to decide where variability can be enclosed using patterns, the *commonality-variability analysis* (CVA) [8, 9] is a method intended to help identifying variation in software systems and thus determine where patterns can be applied in a meaningful way to encapsulate this variation. The essence of the CVA is to determine which structures are unlikely to change over time (the commonalities) and which structures are likely to change (the variabilities). Design patterns often factor out the variability. For example, the strategy pattern factors out changing algorithm implementations from a common interface. Therefore, finding the commonalities and the variabilities in a domain helps in finding suitable design patterns. The CVA-method is suited for designing as well as for redesigning a software system.

1.4 This paper

This paper is structured as follows. We describe the educational context of our study and formulate specific research questions in Section 2. In Section 3, we describe our research method. The results are shown in Section 4. We formulate our conclusions in Section 5. Finally, we discuss our findings, our method, and the implications for both educational development and future research in Section 6.

2. AIM OF THE STUDY

To prepare for our development of scaffolding materials, we intend to investigate what strategies students use when solving a substantial redesign and refactoring task. Our study will involve activities and hindrances, as well as their usage of conceptual knowledge.

In this section, we describe the course in terms of central concepts and course assignments. Finally we formulate our research questions.

2.1 Context

We performed our explorative research during a half semester 5 EC course on advanced object oriented software design for graduate students taught by the first and second author[41]. The course is part of the master's program Software Engineering at the Open University (OU) in the Netherlands. The OU offers part-time studies and distance learning for bachelor's and master's programs.

The central conceptual content of the course consists of design patterns and the underlying object-oriented design principles. The emphasis in the course is on improving systems with respect to improving maintainability.

We use a textbook that explains several design patterns in detail [34], as well as the classic Gang-of-Four book [19] as background material. We wrote a workbook describing more patterns and emphasizes the underlying design principles (for example, design to an interface, prefer delegation over inheritance, or the Open-Closed principle).

Procedural guidance in software design is scarce. The textbook by Shalloway [34] proposes a procedure for software design consisting of six steps: (1) analyse the problem using a commonality-variability analysis (CVA) [9], (2) apply design patterns to encapsulate the variations that the CVA shows, by designing at the conceptual level, postponing the creation of objects, (3) design at the specification level, still postponing the creation of objects, (4) apply creational patterns to create objects, (5) implement, and (6) evaluate. This procedure can be considered strong in that it is domain-specific. At the same time, it is weak: it is a high-level procedure without any details or criteria to determine correctness of a solution, see Section 1.1.

The procedure can be used both for design and for redesign. During the course, we teach students how to use the procedure by showing them examples, but we do not check whether they actually apply this procedure in their redesign tasks.

The assignments in the course constitute a sequence of tasks with an increasing level of complexity. During the whole course, students make small, 'closed' exercises, with solutions they can check themselves. The course comprises three bigger assignments. This paper focuses on the second assignment.

At about one-third of the course, students complete Assignment 1, in which they design a citation management system consisting of about ten classes using a fixed set of design patterns. They receive extensive feedback from the teachers.

At the end of the course there are two assignments, that are performed by teams of two students, to stimulate discussion and reflection.

In Assignment 2, the students redesign and refactor an existing system (JabberPoint, a slide show application, con-

sisting of fifteen Java classes [11, 40]). Students are expected to apply at least the following patterns in their solution: *Command*, *Composite*, *Builder*, *Model-view-control* and *Observer*. The teams submit their results of this assignment in the form of UML class diagrams representing their redesign, the refactored source code, and a working program. Moreover, the students write a *process report* of about one page. In this report, the students reflect on the way they approached the redesign and refactoring of Jabberpoint, the problems they encountered and what they have learned. We do not prescribe a specific format for the process report.

Assignment 3 involves designing and implementing an additional feature to JabberPoint. The specification of the feature is handed to the students only after they have finished the refactoring – the idea is that the teams will maximize the maintainability (and thus flexibility) of their redesign.

2.2 Research questions

As a first step in developing procedural guidance, we investigate how students tackle the complex task to redesign and refactor an existing system, with a higher degree of maintainability as a redesign goal. In particular, we investigate to what extent they use the high-level Shalloway guidelines and what difficulties they encounter. Special attention is given to the use of CVA as the key element of the Shalloway guidelines. We also analyze the students' use of the core concepts of the course, namely design patterns and underlying design principles. Our research question are:

- Q1 What specific activities do students perform and in which order? To what extent do students deviate from the Shalloway guidelines?
- Q2 In what way do students use the CVA in their redesign process?
- Q3 How do students use the design patterns and underlying design principles?

3. METHOD

3.1 Participants

All our students have a bachelor degree in Computer science. Most students study alongside a full-time job, usually IT-related. Students either live in the Netherlands or in Belgium.

The students formed teams by themselves. There were nine teams in total, with six teams of two students and two teams of one individual.

3.2 Data collection and analysis

We used two data sources in the context of Assignment 2: the students' process reports and recordings of teamwise collaboration sessions. Since we were interested in the process rather than the outcome of the redesign, we did not analyze the resulting applications.

Below, we describe the details of our research method, organized by data sources.

3.2.1 Process reports

The reports were subjected to qualitative analysis aiming at unraveling the students' activities and in particular the construction and use of the CVA.

Activities. During a first open coding phase [7] we identified text segments describing students' activities in the design process. In a second, more analytic [39] coding phase we merged similar activity labels into more general codes until a stable set of distinct activities was obtained. For example, *"We generated class diagrams out of the original code"* and *"We made* UML *of the application as we received it"* were mapped onto the activity code 'generate UML'. The coding was done by the first two authors using Atlas-ti. Each report was coded by one of the researchers and reviewed by the other, and differences were resolved when necessary. The final codes resulted from a joint discussion.

Using the final coding, we identified the order of activities for each team. Moreover, we looked for similarities and differences between the students' strategies and the Shalloway guidelines.

CVA use. We checked whether students reported about the *construction* of a CVA, which was coded as an activity in the above analysis. Subsequently, we looked for indicators for the *usage* of the CVA while deciding which patterns are useful in which places. These indicators were coded.

In the case of a redesign, a pitfall is to base the CVA on the source code, instead of on the problem domain. In each case where this activity was mentioned, we identified whether students mentioned that they based the CVA on the code or on the problem domain.

3.2.2 Team collaboration recordings

We asked the teams, on a voluntary basis, to use Blackboard Collaborate as a collaboration tool, with the restriction that their sessions would be recorded and used in our research. Blackboard Collaborate serves as an online conferencing tool, with the possibility to share documents, applications or the desktop. Four teams made use of this opportunity.

Collaborate records audio as well as screen events, which means that we are able to see everything the team members share (UML diagrams, code, etc.) and hear everything they say. Such a recording can be compared to a think-aloud session [17]. The advantage over classic individual think-aloud sessions is that the students naturally tend to share their thoughts with the other team members, so there is no need for continuous monitoring and prompting by a researcher. In particular, the sessions were not influenced by a teacher's presence or a teacher's intervention.

All recordings were transcribed verbatim. We analyzed the sessions with respect to design activities and the use of CVA, and moreover the use of the core concepts of the course, including possible misconceptions. Each of these aspects was analyzed using a coding procedure followed by a more in-depth qualitative analysis, using the codes as pointers to relevant text segments. We were interested in the students' reasoning and paid special attention to the the discussions, decisions and difficulties within the teams. We compared the students' strategies with the Shalloway guidelines when applicable.

Below, we describe the coding of each of the analysis aspects.

Activities. We categorized the activities the team members carried out during the sessions, as well as the activities discussed by the team (that were carried out at another time). We took the final codes of the process report analysis as initial categories in this analytic coding process. We

added codes when necessary. The researchers reviewed and discussed the codings like in the process report analysis.

CVA use. We coded usage of the term CVA as part of the coding of activities. Like in the case of the process reports, we classified whether the CVA was based on the problem domain or the program code, and whether the CVA was used to decide which design patterns could be used.

Use of design principles. We coded the students' usage of the design principles that were discussed in the course. We distinguished between *explicit* usage (i.e., the design principle was named) and *implicit* usage (i.e., the principle was used without explictly naming it).

Use of design patterns. We identified the use of design patterns by coding which design patterns were mentioned explicitly.

Misconceptions. We coded the occurrence of misconceptions concerning design principles and design patterns.

4. RESULTS

4.1 Process reports

Activities. The following activities were identified:

- *Create a* CVA. Four teams mentioned that they constructed a CVA.
- *Generate* UML. Sometimes, students literally stated that they generated class diagrams from the original code; sometimes, they wrote that they constructed class diagrams based on the original code. We categorized both as Generate UML.
- *Discuss code.* In two cases, students mentioned that they discussed the original code, for instance, *"First took a look at the code. Obviously, a lot of spaghetti code"*.
- *Conceptual design.* Two teams explicitly mentioned that they first made a design at the conceptual level.
- *Design.* Three teams wrote that they 'designed' as an activity. We distinguished 'design' from 'conceptual design' because the course emphasized designing at the conceptual level first, followed by designing at the specification level.
- *Divide code.* Four teams mentioned that they divided the code in packages, according to the MVC pattern.
- *Desired patterns.* Five teams explicitly stated that they looked for places to use the patterns that we mention in the assignment. Sometimes, they use the term 'desired patterns' or 'compulsory patterns', in other cases they mention these five patterns by name.
- *Extra patterns.* Two teams literally stated that they looked for places to use 'extra patterns'.
- *Add factories.* Three teams mentioned that they added factories to the code, to separate use and creation of objects.
- *Implement.* Obviously, all teams implemented their design, but seven teams explicitly mentioned implementation as one of their activities.
- *Experiment.* One team wrote that they 'experimented with the code', during their refactoring process.
- *Adjust* UML. Three teams wrote that they adjusted the UML to reflect the changes they made in the code. This means that they coded first, and reviewed the design afterwards.
- *Detect bad smells.* When students discussed bad smells in the code, they invariably talked about their own code, after refactoring (*"During the last phase, we changed the remaining bad smells in our code"*). Therefore, we separated this activity from discussing the original code.
- *Test.* One team explicitly stated that they tested the code after each change.

In Table 1, we show which activities each team reported, and in which sequence. The teams are labeled *A* to *I*. The numbers show the order in which the activity took place. In those cases where a column shows identical numbers (for instance, the column under '*A*' shows the number 3 three times), these activities took place simultaneously.

Table 1: Occurrences and order of team activities

Activity	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
Create CVA	1	-	1	2	-	-	-	2	-
Generate UML	-	-	-	1	1	-	-	-	2
Discuss code	-	-	-	-	-	-	-	1	1
Conceptual design	2	-	-	4	-	-	-	-	-
Design	-	-	2	-	-	-	1	-	3
Divide code	-	1	-	3	2	1	-	-	-
Desired patterns	3	2	-	-	3	2	-	3	-
Extra patterns	-	-	-	-	4	-	-	4	-
Add factories	-	-	3	5	-	-	-	5	-
Implement	-	-	4	5	3	1	2	2	4
Experiment	3	-	-	-	-	-	-	-	-
Adjust UML	4	-	5	6	-	-	-	-	-
Detect bad smells	5	3	6	-	-	-	3	-	-
Test	-	-	-	-	-	-	4	-	-

Notably, every team followed a different path. The sequences of activities are very diverse. The only pattern that emerges is that students work rather code-oriented: they generate UML, discuss the current code, and the activities 'desired patterns' and 'extra patterns' are expressed in the form of *"We looked in the code where we could place the patterns"*.

When comparing these activities with the Shalloway guidelines, we observe that six out of nine teams mention that they paid attention to problem analysis (i.e., the topmost three rows in Table 1). However, only one team performed a problem analysis in the strict sense, namely focusing on the problem domain instead of on the current code (see CVA use below). Two of the nine teams mentioned to have designed at the conceptual level, while none of the teams appears to have worked at the specification level: we did not detect any activity of this type. Three of the nine teams explicitly mentioned the creation of objects.

CVA use. In their process reports, four out of nine teams mentioned creating a CVA, and three of those teams explicitly stated that they based their CVA on the code instead of on the problem. Only Team *D* stated to have constructed a CVA based on the problem.

None of the teams discussed how they used their CVA to find patterns that could implement the variations in the CVA in a flexible manner.

4.2 Team collaboration recordings

Table 2 displays the number and the durations of the recorded collaboration sessions. The team names (A–D) correspond to those of the process reports.

For each of the analysis aspects we describe our general findings, followed by examples of our more in-depth analysis of the considerations, discussions and decisions within the teams, illustrated by quotes taken from the transcripts.

Activities. Table 3 summarizes the activities we observed during the team collaboration sessions.

Table 2: Collaboration session recordings per team

Session	*A*	*B*	*C*	*D*
1	1 h	1 h	1.5 h	0.5 h
2		1.5 h		0.5 h

Table 3: Activities per team

Team	*Activities*
A	evaluate redesign
B	fit patterns in evaluate redesign
C	discuss code fit patterns in
D	discuss code fit patterns in

Table 4 shows which explicit decisions about activities were observed.

Table 4: Decisions per team

Team	*Decisions*
B	compare designs conceptually divide into modules apply desired patterns
C	analyse the code instead of problem
D	analyse the code instead of the problem create implementation design fit patterns in

Team *A* had a session rather at the end of their design process. They both had constructed a redesign for the application, and discussed the merits of these designs in terms of design principles. In fact, the whole session could be called an instance of 'Evaluate redesign', which was not mentioned in any of the process reports.

The discussion within Team *C* was mainly centered around the code:

> *"Ok, the next class, MenuController. I see instances that are all protected. That is totally unneccessary. MenuController does not have subclasses, as far as I know. So why would you use protected? And static?"*

The first session of Team *B* was after both team members had created an initial design. The students explicitly decided to compare both designs at the conceptual level. At the end of this session, they decided to divide the application into modules together, and to divide those modules between them, to work out individually. In the next session, they decided where to apply the 'desired' patterns.

Team *C* discussed, in the first (and only) session, how to start. One of the team members wished to analyze the problem and redesign the system; the other one preferred to analyze the code and start from there. The decision was made to analyze the code.

Team *D* also decided to analyze the code. After that, the students divided the code and made conceptual designs which were subsequently implemented.

CVA use. In none of the sessions we found evidence of CVA usage to decide which patterns could be applied to implement variations.

Team *A* did not mention the CVA at all.

We could observe that Team *B* created a CVA based on the problem:

> *"What are the concepts? In the first place, the presentation. You can load and save a presentation, clone it, navigate throught it, and the variation is that you can save it into* XML*."*

Team *C* did create a CVA based on the code:

> *"Shall we first look at Jabberpoint, and then to the* CVA*? Because then we have the base, and the* CVA *is based on what we have seen in the code."*

The same applies to Team *D*:

> *"I do not have Slide in my* CVA*, and yesterday evening I discovered that I also forgot Style."*

Team *D* discussed whether to use the CVA to analyze the problem:

> *"You could check, for each pattern, how you could use it. That is one method. Another method is the* CVA*-method. Then you analyze the problem. In fact, you are going to redesign the whole program."*

The CVA-method was abandoned:

> *"Let us consider that we have this system and that it is built with these objects, that someone has been thinking about it. Then we just have to think about how we can change it to get it more dynamic."*

One of the team members even says:

> *"The approach with the* CVA *is only of interest to the teachers."*

At the end of the session, the team decided to create, each individually, a design at the implementation level, and to compare those designs in the next meeting.

In the next session they discussed possible strategies to approach the problem of redesigning the system again:

> *"I have looked again at the original code, and I notice that you get overwhelmed by details very easily. And then I thought that it would be possible to tackle the problem like the book says. You could take each pattern, and look for each pattern whether you can apply it. I think you then should analyze how it works at this moment, and try to discover weaknesses, and try to solve them. But another method is to take the* CVA*-method. Then you can check how the program functions, and how you can redesign it from scratch."*

They decide to try to fit in patterns.

The teams *C* and *D* started their sessions by comparing the CVAs they both had created, based on the code. They use these CVAs to discuss the original code. The screen captures provided additional evidence of code-based CVA construction. These captures tend to show an Eclipse code view rather than an UML design diagram during the students' work.

Use of design principles. Table 5 shows, for each principle, how often a principle was mentioned, We distinguish explicit (by name) use and implicit use. An example of an implicit usage of a design principle, in this case cohesion, is the following:

> *"Yes, this is not, ehh, you do too many things. You are creating Styles, and you create a Presentation, that is not really good."*

Table 5: Use of design principles

principle	*total*	*expl*	*impl*
low coupling	19	4	15
high cohesion	19	-	19
separate creation	12	5	7
encapsulate variation	11	4	7
design to an interface	11	2	9
single responsibility	5	-	5
information hiding	5	-	5
locality	3	-	3
abstraction	1	1	-
open-closed	1	1	-
prefer delegation	1	-	1

Use of design patterns. In many cases, students mentioned a pattern by name.

> *"And next, I will examine the application of the Observer pattern on the commands."*

> *"What we had, but is probably wrong, is that slide items are the leaves in the Composite pattern."*

In Table 6, we show how many times each team mentioned each pattern. In the assignment, we mention five patterns that we at least expect to see in the solution. We discern these patterns from other patterns by marking them grey.

Table 6: Use of patterns

pattern	*A*	*B*	*C*	*D*	*total*
mvc	7	24	7	3	41
command	6	9	8	4	27
composite	10	7	5	5	27
observer	6	7	4	2	19
builder	4	14	-	-	18
factory	-	-	7	7	14
visitor	5	13	-	-	13
abstract factory	5	2	-	1	8
bridge	-	3	2	-	5
iterator	-	2	3	-	5
object pool	-	-	3	-	3
singleton	2	-	-	-	2
strategy	-	-	-	2	2
null object	-	-	1	-	1
template	-	-	1	-	1

The first session of Team *B* was after both team members had created an initial design. When discussing patterns in both designs, there was no distinction between factory patterns and other patterns: they did not adhere to the advice to postpone decisions about object creation. One of the members said he had first 'extended' the CVA (which probably means that he added more detail), and then had checked at which places he could 'fit in' design patterns:

> *"And after that, I have thought about which patterns we can apply, and how."*

The other team member also started the design by trying to fit patterns in; it was not clear whether he did this by looking at the code or based on something else. Implicitly, he says he has designed at the conceptual level:

> *"I have been tinkering too, and I have written down which patterns we could apply. But I did not look at the level of methods, so to speak."*

In summary, we observed this team evaluating redesign, while they spoke about having 'fitted design patterns in', at the conceptual level.

Team *C* discussed each class of the original code, in the sequence that the file manager showed, and tried to point out what was wrong or *"ugly"*. Obviously, there was no postponing of object creation.

> *"Well, you are right, two things are tangled here. That is how I see it. You have parts of the model and parts of the view. And yes, there should be some image here somehow, but the presentation should be left out of it."*

The students also discussed which patterns they could use in which places:

> *"Yes, I have been playing with patterns too, and yes, for that part I also had* MVC*, and then, also the same as you have, use Command pattern model to view, yes. You probably mean, for instance, next slide and previous slide, and so on, you probably want to put that in a Command pattern, I suppose?"'*

Misconceptions. We encountered the following misconceptions while analyzing the group collaboration sessions.
- The commonality-variability analysis is considered as a product of code inspection:

> *"Shall we first look at Jabberpoint, and then to the* CVA*? Because then we have the base, and the* CVA *is based on what we have seen in the code."*

- How to create objects is included in a textsccva:

> *"I extended the* CVA *by adding create in the system."*

- A Bridge is considered to be an alternative for Observer. When talking about Observer using events or using Observable:

> *"I had thought of events for Observer, but I understood that you would solve that using a Bridge, and that is fine."*

- Composite leads to a loss of performance because it 'creates' lots of 'webbing objects':

> *"We have to pay attention, that with that Composite, that we do not overdo it, and make an absurd number of webbing objects, because that hurts, and is very bad for performance."*

- One should start with factories, in contrast to what is advised in the course:

"There is a very interesting part in the Design patterns book, about refactoring as simple as possible, that the best way to do that is to start with that factory. So that is what we should do first. Then the rest is easier."

5. CONCLUSIONS

In this section we recapitulate our research questions and formulate our conclusions as answers to these questions.

Q1: *What specific activities do students perform and in which order? To what extent do students deviate from the Shalloway guidelines?*

Students appear to carry out a range of activities, in which 'implement' and 'apply desired patterns' prevail. Activities focused on conceptual design scarcely appear. In contrast, many activities, such as those connected to problem analysis, are directed at the code rather than at the problem domain.

Students appear to carry out their activities in different orders. We did not find any universal pattern. However, we observed a tendency to concentrate on the code almost directly. The Shalloway guidelines were never followed. In particular, the creation of objects is not always postponed. Students appear to either skip the CVA or perform the CVA on the basis of the code.

We could observe that students find it difficult to reason and decide about the sequence of events. When students decide to deviate from the Shalloway guidelines, their reasoning appears not to be technical, but instead based on their perception about the practical ('real-world') value of the guidelines, the intention of the teachers, or the necessity of a thorough analysis for the particular case at hand.

Q2: *In what way do students use the* CVA *in their redesign process?*

The students tend to base their CVA's predominantly on the application code. We have not found any usage of the CVA results for choosing design patterns.

In some cases, student contemplated to use a CVA, but decided at the last moment to abandon it anyway. In line with our observations about the Shalloway guidelines in general, some students appear not to take the CVA as a serious (re)design method (*"that is something for the teachers"*).

Q3: *How do students use the design patterns and underlying design principles?*

With respect to design principles, we observed that students often mention and use these principles in their discussions. Implicit usage appears slightly more frequent than explicit usage.

The principles playing a major role in the course, i.e., low coupling, high cohesion, separate the creation of an object from its use, encapsulate variation and design to an interface, are all frequently mentioned, both implicit and explicit.

Students use design principles extensively in discussions about the value of candidate solutions.

Patterns are mentioned and use extensively by the students. The patterns that were mentioned in the assignment appear more often than the other patterns. As mentioned above, factory patterns are used in an early stage.

Deciding which patterns to apply was never done on the basis of a CVA. Instead, the reasoning appears ad-hoc, taking place while browsing through the code looking for spots where patterns might be applied.

Misconceptions regarding design principles and patterns turn out to be scarce.

6. DISCUSSION

In this section we reflect on the findings and the methodology of our study. We also discuss the implications for educational development and our plans for follow-up research.

6.1 Reflection on the findings

Our conclusion that the activities of students are rather code-oriented (as opposed to problem-oriented) is in line with the observation that students tend to produce a problem analysis that is far from complete [25].

In terms of the SOLO taxonomy [3], software design requires students' response at the highest aggegration level, that is, *extended abstract*. We often observed students discussing at a unistructural level (when discussing details in the code), or at the multistructural level (when discussing classes); sometimes, they discussed at the relational level (discussion the division of responsibilities between classes for instance). The extended abstract level (in this case, that would be discussing how an aspect of the problem could be solved using a design pattern) appeared to be absent.

Our students lack the top-down, breadth-first approach of experts, and sometimes show the line-by-line, local and concrete approach of novices [33]. This can be seen as an indication that more procedural guidance is needed, as well as more practice in applying the procedures, cf. [27].

In one case we observed that a team made a choice between refactoring the code without a redesign and redesign using a CVA. This suggests that these students find high level redesigning too difficult, and therefore try to refactor without redesign. This observation can be seen as another indication that more practice is needed.

The observation that students appear skillful in reasoning about design principles is consistent with a finding by Tenenberg et al. [43] about the relative stability (throughout their studies) of students' knowledge about design principles.

It is difficult to interpret the fact that students often use design principles implicitly instead of referring to them explicitly. One could argue that the direct use of design principles, by name, indicates a more mature understanding of those principles, but the fact that they do not need to name a principle might also mean that they are very familiar with the meaning of a principle.

Our students talked far more often about the 'compulsory' patterns then about other patterns. Also, five out of nine teams mentioned that they first looked for places where they could use the patterns we prescribe. A possible explanation is that students interpreted our mentioning of those patterns as a form of procedural guidance ("first try to see where those patterns can be of help, and then check whether the design still lacks maintainability, and try to use other patterns to enhance this maintainability"). Of course, the fact that students want to create a design and code that will satisfy the teachers plays a role as well. On the other hand, students might be aware of the fact that the teachers would like to see a CVA, but this did not lead to all process reports mentioning a CVA, which indicates that 'pleasing the teachers' does not play a big role.

6.2 Reflection on the methods used

Our analysis method appeared to be useful to investigate various aspects of the students' strategies, including the students' reasoning and decision making processes. The data turned out to be surprisingly rich. The non-obtrusive character of the analysis makes it worthwhile to apply in other contexts.

The number of students involved in this exploratory study is small, which can be seen as a weakness of the method. However, we appeared to reach theoretical saturation after analyzing the data of 2–3 teams. We expect that, in our homogeneous student group, a larger sample would not have brought up much more detail. However, we are planning to repeat the research with a bigger sample, as well as in other universities. Moreover, the analysis method will be used in follow-up studies in the bigger design research project.

Discussions between the coders mainly concerned characterizations for merged sets of activity codes. After deciding upon the final categories, inter-coder agreement in our analysis turned out high. In follow-up research we will investigate the reliability of the method in more detail.

We have seen that students do not mention every activity they perform in their process report (for instance, evaluate a redesign was never mentioned in the process reports, but we observed it twice during the collaboration sessions). This means that process reports do not give a complete image of the process that students followed. It might be possible that students report activities they did not perform, to 'please the teachers', but the absence of many 'Shalloway-activities' in many reports does not make that likely. One the other hand, the collaboration sessions also do not give a complete image, because they form a small portion of the whole process. Therefore our triangulation is sensible: the evidence is strongest where both forms of data show the same tendencies. Moreover, the recorded collaboration sessions show only part of the whole process, because team members, in general, divide the work, work individually, and collaborate by mail, Skype, phone, face-to-face, and in some cases using Collaborate. We plan to extend the analysis method to other data sources.

6.3 Implications for educational development

Our findings suggest that students find it difficult to grasp the value of a CVA. The reason can be threefold. First, it might be the case that students have to explicitly practice redesign: the problem with redesign is that there is an escape in the form of refactoring without any redesign. Second, we might have to give students more opportunity to practice in general using the CVA. Third, it might be the case that the CVA in itself does not give enough guidance to be of use for students.

6.4 Future work

We would like to expand this study by observing more students, and compare process reports and observations to the designs and implementations they hand in. We could compare the design patterns students really used to the design patterns that we could hear them speak about, and we could relate the quality of their work with the activities they mentioned.

We will use the results of our study to develop procedural guidance for scaffolding students' redesign activities. For the development of procedural guidance it is relevant that Shalloway's guidelines appeared not to be helpful for the students. Besides the need to provide more details, it might be worthwhile to construct more specific guidelines that are closer to how students proceed naturally. Moreover, systematic practice seems to be in order, as suggested by the developers of the 4C/ID model [27].

We are planning to make the research methods of the present exploratory study more robust and extendible to other environments. Thus, future procedural guidance can be tested by colleagues in partner universities.

7. REFERENCES

[1] J. R. Anderson. Problem solving and learning. *American Psychologist*, 48(1):35, 1993.

[2] C. J. Atman, J. R. Chimka, K. M. Bursic, and H. L. Nachtmann. A comparison of freshman and senior engineering design processes. *Design Studies*, 20(2):131–152, 1999.

[3] J. B. Biggs and K. F. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, New York, USA, 1982.

[4] E. J. Braude. *software design, from programming to architecture*. John Wiley & Sons, 2006.

[5] L. L. Bucciarelli. Reflective practice in engineering design. *Design studies*, 5(3):185–190, 1984.

[6] A. Chatzigeorgiou, N. Tsantalis, and I. Deligiannis. An empirical study on students' ability to comprehend design patterns. *Computers and Education*, 51(3):1007 – 1016, 2008.

[7] L. Cohen, L. Manion, and K. Morrison. *Research methods in education*. London, New York: Routledge, 2013.

[8] J. O. Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, Departement Informatica, 7 2000.

[9] J. O. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, 1998.

[10] N. Cross. Expertise in design: an overview. *Design studies*, 25(5):427–441, 2004.

[11] I. Darwin. Gui development with Java. *Linux Journal*, 1999(61es):4, 1999.

[12] S. P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237–267, 1993.

[13] C. Denzler and D. Gruntz. Design patterns: Between programming and software design. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 801–804, New York, NY, USA, 2008. ACM.

[14] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, and C. Zander. Can graduating students design software systems? *ACM SIGCSE Bulletin*, 38(1):403–407, 2006.

[15] A. Eckerdal, R. McCartney, J. E. Moström,

M. Ratcliffe, and C. Zander. Categorizing student software designs: Methods, results, and implications. *Computer Science Education*, 16(3):197–209, 2006.

[16] P. Flores, N. Medinilla, and S. Pamplona. What do software design students understand about information hiding?: A qualitative case study. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 61–70, New York, NY, USA, 2014. ACM.

[17] M. E. Fonteyn, B. Kuipers, and S. J. Grobe. A description of think aloud method and protocol analysis. *Qualitative Health Research*, 3(4):430–441, 1993.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the design of existing programs*. Addison-Wesley Reading, 1999.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[20] J. Garofalo and F. Lester. Metacognition, cognitive monitoring, and mathematical performance. *Journal for Research in Mathematics Education*, 16(3):163–176, May 1985.

[21] P. Gestwicki and F.-S. Sun. Teaching design patterns through computer game development. *J. Educ. Resour. Comput.*, 8(1):2:1–2:22, Mar. 2008.

[22] D. H. Jonassen. Toward a design theory of problem solving. *Educational technology research and development*, 48(4):63–85, 2000.

[23] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[24] P. A. Kirschner, J. Sweller, and R. E. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[25] C. Loftus, L. Thomas, and C. Zander. Can graduating students design: revisited. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 105–110. ACM, 2011.

[26] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[27] J. J. Merriënboer and P. A. Kirschner. *Ten Steps to Complex Learning*. Taylor & Francis, New York, USA, second edition, 2013.

[28] R. Or-Bach and I. Lavy. Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin*, 36(2):82–86, 2004.

[29] F. Palma, H. Farzin, Y.-G. Guéhéneuc, and N. Moha. Recommendation system for design patterns in software development: An dpr overview. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 1–5, Piscataway, NJ, USA, 2012. IEEE Press.

[30] R. D. Pea. The social and technological dimensions of scaffolding and related theoretical concepts for learning, education, and human activity. *The journal of the learning sciences*, 13(3):423–451, 2004.

[31] T. Plomp. Educational design research: An introduction. In T. Plomp and N. Nieveen, editors, *Educational design research*, pages 10–51. Enschede: SLO, 2013.

[32] S. I. Robertson. *Problem Solving*. University of Luton, UK, 2001.

[33] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[34] A. Shalloway and J. R. Trott. *Design patterns explained: a new perspective on object-oriented design*. Pearson Education, 2004.

[35] D. Skrien. Learning appreciation for design patterns by doing it the hard way first. *Computer Science Education*, 13(4):305–313, 2003.

[36] S. Smith, S. Stoecklin, and C. Serino. An innovative approach to teaching refactoring. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 349–353, New York, NY, USA, 2006. ACM.

[37] E. Soloway, J. Spohrer, and D. Littman. E unum pluribus: Generating alternative designs. In *Teaching and Learning Computer Programming*, pages 137–152. Lawrence Erlbaum Associates, 1988.

[38] S. Stoecklin, S. Smith, and C. Serino. Teaching students to build well formed object-oriented methods through refactoring. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 145–149, New York, NY, USA, 2007. ACM.

[39] A. Strauss, J. Corbin, et al. *Basics of qualitative research*, volume 15. Newbury Park, CA: Sage, 1990.

[40] S. Stuurman and G. Florijn. Experiences with teaching design patterns. *ACM SIGCSE Bulletin*, 36(3):151–155, 2004.

[41] S. Stuurman, F. J. Wester, and M. Witsier-Vogel. *Design patterns*. Open Universiteit Nederland, 2002.

[42] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells*. Morgan Kaufmann, Elsevier, Waltham, MA, USA, 2015.

[43] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, T.-Y. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. E. Moström, M. Petre, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Schwartzman, B. Simon, C. Stoker, A. E. Tew, and T. VanDeGrift. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.

[44] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. Kumar, and C. Prasad. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 243–252. Australian Computer Society, Inc., 2006.