Studies into interactive didactic approaches for learning software design using UML

Stikkolorum, D.R.

Part VI

Reflection

Chapter 12

Conclusion and Future Work

In this chapter we summarise the conclusions and highlight the contributions of the research of this dissertation. We also relate the findings across chapters. In addition we make recommendations for approaches, interventions, and tooling for teaching software design.

12.1 Reading Guide

This chapter starts with a summary of the research objectives that are explored in this dissertation (Section 12.2). Subsequently it discusses the three main research questions (Sections 12.3, 12.4 and 12.5) with the following structure:

- main objectives,
- research approach, and
- recommendations for teaching software design

In addition, this chapter will continue with explaining the supporting role of our own developed research and educational tooling, as it affects all three research questions (Section 12.6). The chapter closes with future directions research could take (Section 12.7).

12.2 Research Objectives

The research in this thesis supports the following main objective:

How can we improve the ways of teaching software design?

We decided on the following scoping of our research:

- We focus on online learning environments, such as MOOCS. This implies that the tool for doing modelling, the tool for its assessment, and the tools for generating feedback during modelling have to fit into an online learning environment.
- We put an emphasis on the learning of designing. We support the opinion that the student challenge is about the comprehension of the design – an abstract and difficult topic – not on the specific UML notations.

In order to explore the main research objective, we refined it by the following three research questions:

RQ1 Can we assess how good students are at designing software?

RQ2 What guidance do students need to improve their understanding of designing software?

RQ3 How can we increase the motivation and engagement of students for learning software design?

The relationship between the research questions and research topics is shown in Figure 12.1). In the figure is seen that a research question relates to a specific topic area (dashed, rounded rectangles). Each topic area is divided into subtopics that are explored in the different chapters of the dissertation. Some chapters support the research of other chapters (light arrow).

Software design education can be explored in many different ways. In our research we mainly focused on the following topics:

- Assessing students' skills of software design: we wanted to assess students' comprehension of software design (Chapter 2). We explored measuring students' comprehension with a test that is based on general software design principles.
- The role of guidance and feedback in the process of students' learning of software design: we want to understand the way students reason and what strategies they

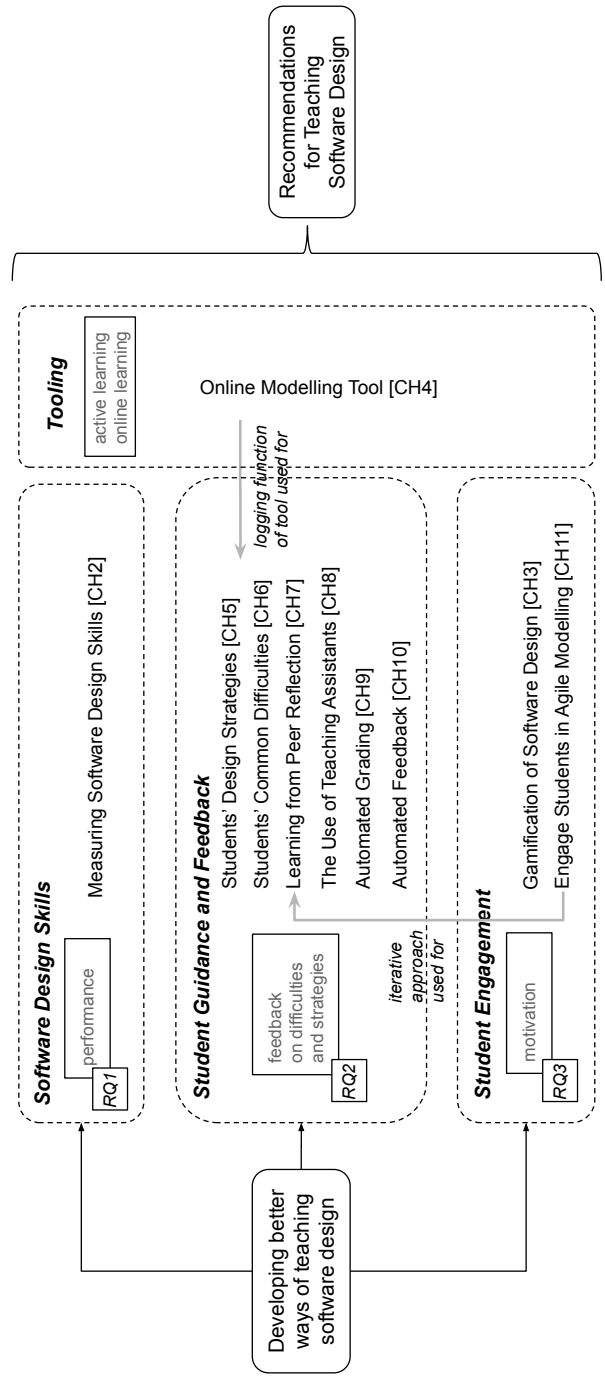

Figure 12.1: *Overview of the research themes and their corresponding chapters.*

use when making decisions while making a software design (Chapter 5). We aimed to reveal students' common problems while making a software design (Chapter 6). Next to revealing the above mentioned strategies and difficulties, we analysed the use of peer-feedback by fellow students (Chapter 7) and guidance and feedback of teaching assistants (Chapter 8) in order to collect best practices.

The knowledge of the common problems and strategies helped to develop tools and educational approaches where feedback and guidance play a role. We explored automated grading with the use of machine learning (Chapter 9) and automated feedback by extending our tools with a feedback agent (Chapter 10)

- Approaches for improving the engagement of students for learning software design: we studied the use of gamification for engaging students into software design. We developed a puzzle game for the learning of software design with UML class diagrams (Chapter 3). From that game we used the concept of an avatar giving feedback and comparing student solutions against example solutions in our online UML editor.

  Another approach for engaging students in software design was the exploration of the use of an interactive workshop on agile software design with UML (Chapter 11).

- Research tooling for supporting the aforementioned topics: In particular, we desired to develop an assessment tool for measuring software design comprehension and a tool for software designing that could monitor design activity (Chapters 2, 4).

In the following sections we discuss *RQ1*, *RQ2* and *RQ3* and relate the findings and contributions from different chapters. For the various angles of our research questions, we provide closing recommendations for teaching software design. For an overview of all the recommendations that emerged from the research in this dissertation, along with the problems they address we refer to Table A.1 in Appendix A.

12.3 *RQ1* – Can We Asses How Good Students Are at Designing Software?

We start by summarising the aspect of assessing students' software design skills. Then, we explain the approach we followed in our research. Subsequently we explain what effects the findings had on the research direction of this dissertation. We conclude this section with recommendations for teaching software design.

12.3.1 Main Objectives

The main objectives for exploring RQ1 (*Can We Asses How Good Students Are at Designing Software?*) were the following:

- To have an objective and repeatable instrument that could be used to provide insight into the software design skills of students. A unique aspect of our approach is that we focused on the understanding of software design principles.
- To be able to compare skills between students and groups of students. This enables us to explore different educational interventions. One practical constraint on the assessment instrument was that it should be applicable to a large group of students. This led us into the direction of an online test.
- To have insight into 'learning yield' of educational interventions (by looking at the difference between the after and before level of the assessment).
- To understand the relation between design skills and generic skills. Therefore, we studied the relationship between the abstract reasoning skills and language skills and software design comprehension.

Next, we move on explaining our research approach.

12.3.2 Research Approach

Our initial attempt towards an instrument was to construct an online multiple choice test for measuring the comprehension of software design. The test consisted of questions based on design principles (Chapter 2). Each question presented alternative solutions for designs. These alternatives differed in the way they satisfied a particular software design principle. In this test students had to choose the best solution out of four alternatives. We applied the test to several groups of students in software design courses. We conducted a pre-test at the beginning of a software analysis and design course and a post-test at the end of that course. By running these experiments we found that our instrument was suitable for measuring learning yield. Moreover, the results of the tests were consistent with the course examinations. We consider this as a validation of our instrument. Also, the results demonstrate the scalability (number of students) of the test. Further more, we found that knowledge of the UML notation did not relate to the performance of the students on software design comprehension. This result is in line with our intentions: we aimed to design a test that measures software design skills but at the same time required little knowledge of any particular software design notation.

We explored whether abstract reasoning and language skills influence the performance on the software design skills test. In order to measure this we conducted three tests: one on visual reasoning (Raven test), one on verbal reasoning (verbal analogies) and one that tested general language proficiency (C-test). We found that visual- and verbal reasoning correlate with the performance on the design skills test. An implication of this is to find complementary teaching methods for students that have a lower talent for abstract reasoning. The language proficiency level did not correlate with the performance on the design skills test. We assume this can be explained by the high level of knowledge of the English language in the sample groups. The test results may not generalise to student populations that have a poorer understanding of the English language.

12.3.3 Effect On Research Directions

After having gained experience with performing the design skill test, we found some limitations. In its current shape, the online test does not call on any synthetic capabilities/skills for creating software designs. Instead the test seems to address the ability to recognise good and bad designs - which also involves some analysis skills. The test was usable for diagnosing which design principles students did not understand. Using this test, we are able to guide students into the direction of explanations about the topics on which they achieved low scores.

However, even if students master the design principles, we suspected that *creating* a design involves additional types of reasoning steps (such as making design decisions) that cause difficulties. The design skills test does not cover such skills. These limitations were due to the multiple-choice character of the test. In the subsequent steps in our research, we tried to remedy these limitations. We created an online interactive design environment, WebUML, which enabled us to monitor the synthetic process of constructing software designs. WebUML enabled us to explore higher levels of Bloom's taxonomy (from *apply* and up) than we could with the design skills test. In particular, we started exploring methods for identifying the difficulties students encounter during the design process. More about WebUML in Section 12.6

By using our tools (the online test and WebUML), additional questions arose. These questions relate to the research questions that are discussed in the next sections:

- We want to understand why students have difficulties with specific topics in the area of software design and what specific design decisions they make that lead to their solution. If we understand this, we could improve our guidance and/or feedback towards students (RQ2).

- We want to gain insight into what kind of learning interventions increase the engagement in the learning of software design topics (RQ3).

12.3.4 Recommendations for Teaching Software Design

This section often refers to recommendations for education that can be found in Table A.1

In this section we summarise recommendations for teaching software design distilled from the research of RQ1.

Based on the experience we have with assessing students with our online test and WebUML, we recommend teachers to use design principles as a basis for measuring software design understanding (Table A.1 Recommendations 18, 19 and 22). Although UML syntax should be discussed in courses, we recommend teachers to focus on design principles rather than syntax (Table A.1 Recommendation 20). The understanding of design principles lays the foundation for understanding complex software design challenges.

The test covers the following topics about software design principles: Dependencies, Cohesion, Dependencies and Changes, Maintainability, Dependencies and Reuse, Responsibility, Extensibility, Reuse Information Hiding, and Coupling. Students can use the online design skills test for self-reflection on their software design skills. Such a self-test produces reports that present the number of correct and incorrect answers per topic. Through these reports students can uncover their own deficiencies.

12.4 *RQ2* – What guidance do students need to improve their understanding of designing software?

The second aspect we describe is our approach for understanding what guidance students need when learning software design. First we mention our main objective. Then we describe our research approach in which we used our own tooling. Subsequently we show that the research revealed different strategies that students use for designing software and discuss the common difficulties that they face. We close this section with the recommendations we have for education based on our findings.

12.4.1 Main Objectives

The main objectives for exploring RQ2 (*What guidance do students need to improve their understanding of designing software?*) were the following:

- To have a set of guides that can help students when learning software design.
- To explore if the guidance approach could be implemented in an online environment.
- To be able to recognise students approaches to guide them to a proper solution.

We explored the above-mentioned objective in online settings with large number of students. This exploration identified the need for students to perform self-assessments. For such self-assessment we studied the automation of guiding and grading. Thus, we extended our objective:

- To offer automated didactic guidance for students during their software design tasks.
- To explore the possibility of an automated grading system in an online software design environment.

12.4.2 Research Approach

In researching which guidance helps students best, we first needed to understand what difficulties are common for students and what strategies they use when performing a design task. Knowing students' difficulties and strategies gives us leads to possible guidance that can be used in educational interventions.

For exploring students' strategies, we started with equipping our WebUML tool with logging functionality that logged user actions, such as creating of UML elements, deleting elements, moving elements and modifying elements (see Chapter 5). During a students' modelling session, all items were logged with their corresponding time stamp. We were able to analyse the logs with our own developed tool *LogViz* that visualises the student's modelling session.

The analysis revealed two dominant strategies that students use when designing class diagrams:

- Breadth-first: an approach where students first make an overall high level class design, including associations and add details in the form of attributes and operations later.
- Depth-first: an approach where students first create a class with much detail (attributes and operations) and then relate it to another class (associate) with much detail.

We compared the grades of the students that used the breadth- and depth-first strategies. We could not find a significant difference that shows one strategy is better that the other (see Chapter 6). We did however find that the Depth First strategy used fewer re-orderings of the layout of the diagram. Hence, if we would consider 'Movement'/Re-ordering of classes as a measure of efficiency, then this would make Depth First more efficient than Breadth First.

Next to the strategies, analysis shows that students spent a lot of time on adjusting layout and understanding the task. Both adjusting layout and understanding the task also show up in other research, described in Chapters 7 and 8. In our research (see Chapter 6) a relation between the layout and the grade was shown. It seems that students that organise their diagrams better also create a design that is better in terms of design principles. Therefore, next to the focus on design principles, lecturers should guide their students to care about the layout of their diagrams.

For understanding the common difficulties students have, we added a form to WebUML. With this form, we were able to collect difficulties students have when performing modelling tasks. During several practical sessions, we asked students to fill in and submit the form with questions and problems that arose during the performance of their tasks. We were able to categorise the common ones, based on questions students have (see Chapter 6). These categories are:

- task comprehension,
- tool usage,
- tool feedback,
- UML/OO comprehension, and
- UML syntax/notation.

Surprisingly, a low amount of questions was in the 'UML/OO comprehension' category. Knowing these common problems helps to create future interventions for learning software design.

We found that a limitation of our exploration into students' strategies and difficulties was that they did not reveal the thinking process of the students that they performed while creating software designs. Students tend to summarise and abstract their own thoughts or are unable to formulate what they don't understand (unconscious incompetence from the stages of competence [22]). In this way we could not obtain a deeper understanding of students' reasoning.

Because of the aforementioned limitation, we designed a study in which we used peer reflection to reveal more detail about the problems that arise with students while constructing a software design. During a software analysis and design task that was part of a running course, we asked students to be the peer-reflector of their fellow student (Chapter 7). Students worked in pairs and were asked to discuss their intermediate results on the basis of questions that were focused on analysis- and design-comprehension and modelling-skills. This approach triggered the students to actively discuss and critically reflect their task. A lecturer is guiding the process, but can also actively be approached by students for questions. The lecturers that were involved during the peer-reflection sessions were positive about this approach and noticed students were more involved and active than usual. In addition, this approach can be used by students as a self-guiding approach outside of the classroom. By analysing the student peer-reflections, we found 'points of concerns' that can be used when creating courses or for subsequent research.

Still, a limitation of this research was the low amount of detail students provide when documenting their answers. We can say that we found more detail than the research with the online form in WebUML (Chapter 6). Eventually this led to an extended collection of common difficulties, but not yet an in-depth insight into students' thinking strategies.

Another limitation of peer-reflection we see is that students discuss matters based on their prior knowledge and assumptions. This means that such a discussion can only lead to a certain level of learning yield. Because of the small age gap between novice and senior students, we came up with the idea to explore the role of teaching assistants (TAs). Teaching assistants are selected because of their (high) competence in the subject of the course. Using this competence, we could enable similar discussion as in the peer-reflection approach, but with a deeper level of discussion. Additional motivations for conducting this research were:

- to extend our knowledge on common difficulties that students have.
- to categorise difficulties, and
- to have a categorisation of support approaches used by the TAs.

In a case study, we analysed the involvement of the aforementioned TAs as support of students' learning during a bachelor course on software analysis and design. The TAs were involved in:

- supervision sessions, in which the students worked in groups on a practical assignment.
- weekly meetings with the lecturers running the course, and
- giving feedback to the students on the work they handed in.

The data that was analysed came from transcribed interviews with the TAs, student answers on questionnaires about the TA support and written feedback of the TAs on the hand-ins.

We found that TAs are mostly consulted during abstract reasoning tasks, such as electing conceptual elements in a diagram (creating classes based on some description). In this research tool support was less of an issue. Although we suspect TAs were involved explaining the tool in a more natural way. The explanation of tools by TAs was not initiated by problems that occurred. We observed that students have high expectations of TAs. The TAs are 'forced' into a lecturer role. Just like with lecturers, TAs are expected to give confirmation ("Is this the right way?") and should give clear directions.

We observed that typically for SE assignments, student have difficulties with TAs (just as lecturers) giving inconsistent answers (from the students point of view). We suggest to use our categorisation of common software design difficulties and the categorisation of guiding approaches to create profiles and/or programs to prepare future teaching assistants.

The exploration of student reasoning in online settings led us to the idea of exploring the use of automated grading and guidance in order to enable the online tooling for self-assessment.

We explored the application of machine learning for grading student solutions (see Chapter 9). We approached grading as a prediction problem and trained models for the prediction of the grades of software design assignments, with a regression model as well as classification models (numeric versus symbolic outcomes, at different scales). We tested two different sets of features. One that used a model that was built with a feature set that focused on generic (solution elements that dominate the design construct, i.e. inheritance or aggregation) modelling elements and second experiment that used a feature set that consisted of specific elements (solution elements that carried specific names, i.e. 'car'). We conclude that grading student models with machine

learning with a 10 point scale is not accurate. Although, using a 3 point scale in the second experiment resulted in a better accuracy (69.42%) than the 5 point scale of the first experiment (65.24%), this is still not reliable to use for automated grading. For a rough indication of design quality it could be integrated in online tools. This led us to the idea of having a guide that could do intermediate judgements during the task and provide the student of feedback.

We explored the application of a didactic software component, a pedagogical agent (see Chapter 10). The pedagogical agent guides the students through their software design task. The agent can be asked for guiding feedback on demand during software design tasks. The agent compares a model solution with the (intermediate) submission of a student. In this way the agent guides the student towards the solution of an assignment. Early findings of a tested prototype show that students appreciate the provided guiding feedback as confirmation of being on the right track.

12.4.3 Recommendations for Teaching Software Design

This section often refers to recommendations for education that can be found in Table A.1

In this section we summarise recommendations for teaching software design distilled from the research of RQ2.

Although more research is needed to reveal more detailed information about strategies students use when solving software design problems, we advise lecturers to be aware of the difficulties students have in understanding and constructing software designs. We suggest to use activating teaching forms for gaining insight into students' struggles with learning software design.

We recommend using approaches were students discuss intermediate results in peer-reflection settings (Table A.1 Recommendations 22 and 26). By discussing intermediate results students are more willing to reflect on and discuss their progress. This discussion not only increases the confidence of the student, it also activates the student to formulate questions and approach TAs or lecturers more often.

Besides the learning of the syntax of modelling languages and the comprehension of software design principles, lectures should focus on the layout and the organisational aspect of the diagrams used for modelling (Table A.1 Recommendation 29). A clear layout supports a better understanding of a diagram, even when constructing a diagram.

Use the logging of students' design tasks in order to enable teachers to analyse students' design approaches and reflect on the patterns they observe. This can be applied

during lectures or other feedback moments. By using a logging system or peer review approaches, students are enabled to reflect on each other and also on themselves. Lectures can use the reflections to focus on those matters in follow-up lectures. We see feedback as a key factor for successfully understand the topic of software design (Table A.1 Recommendations 2, 7, 9, 13 and 19).

By knowing a student's strategies, lecturers might have to choose another approach when supporting a student. It is advisable to collect and discuss the problems that arise in student-groups.

When employing TAs during a course, we suggest to plan weekly meetings with the TAs. On the one hand, these meetings should discuss the possible solutions for the assignments (so that the TAs understand the range of possibilities). On the other hand they serve as a bridge between the lecturer and the students that follow the course (Table A.1 Recommendations 7 and 13). In addition, we recommend to prepare TAs before a course starts. Our categorisations of common challenges students face where TA support can help, can be used to create profiles for the selection of TAs. Also our findings can be used for creating programs for training future teaching assistants.

12.5 *RQ3* – How can we increase the motivation and engagement of students for learning software design?

The third aspect we studied are approaches for engaging students in activities for learning software design. First we summarise our main objective. Second we discuss two different research approaches in which we explored the application of gamification and we explored an interactive workshop as educational tool. We close with recommendations for education based on this research.

12.5.1 Main Objectives

For studying engagement, we explore the use of gamification. A main objective then becomes the design of a gamification approach that satisfies the following goals:

- It has a positive influence on the engagement of students during software design activities.
- It can be used to breakdown the difficult topic of software design into smaller parts that could be exercised separately and repeatedly.

As an alternative instrument for student engagement, we studied the use of an interactive workshop. Key elements of this workshop are i) the simulation of a 'real life' agile development process, and ii) stimulation of peer-modelling, peer-discussion and peer-review between students while making a UML software design.

For this workshop, we studied:

- How the workshop affects the student's motivation for the learning of software modelling.
- Whether the workshop benefits the understanding of the relationships between the UML models used in different phases of software development.

12.5.2 Research Approach

From our experience with several UML editors, we believe that an integration of a learning platform with a UML editor could be a good approach. This led us to the construction of a game that was based on the idea of a UML editor: The Art of Software Design.

We developed an educational game called *The Art of Software Design* (AoSD). AoSD uses a gamified UML editor as the basis of the game (Chapter 3). The game supports the "learning by doing" approach. The aim of the game is to solve a series of puzzles, where puzzles correspond to the completion or creation of a design or a design-fragment.

By using actual software design concepts, such as class or association, as game elements we aim to motivate the students for learning design principles such as cohesion, coupling and modularity. AoSD was designed in such a way that it offers a breakdown of topics in increasing level of difficulty. Students are able to choose the order in which they want to work on topics - while staying within logical paths of learning demonstrating prerequisite knowledge before moving on to topics that build on this more basic knowledge. The ability to choose your path (implementing a need for autonomy [106]) in a video game supports the engagement of the player.

The AoSD game includes a scoring mechanism (which implements feedback on competence and achievement [106]). The scoring mechanism works interactively: it shows the updates to the score while a students is making changes to his/her solution. The feedback on the score was provided through a combination of visual and audio effects.

A preliminary study with AoSD showed that students valued the freedom of choice in learning paths. When a student got stuck, it was possible to choose another puzzle and come back to the difficult one later. With this approach, the student remained engaged

in the game. In the same study, we also found that textual instruction to a puzzle should be kept short. Otherwise, students are more likely to skip such instructions. We found that subjects that were not used to UML terminology were using UML terminology after playing the game.

A limitation of AoSD is that AoSD is not equipped with the possibility to monitor the activities of students which can be used to determine their strategies. As mentioned earlier, monitoring strategies or modelling steps could give us insight into students reasoning.

The use of WebUML in our own classroom and experiences in sister courses on agile software development projects led us to the idea of integrating software design and agile practices in a workshop. The aim of this combination was to explore if an integrated approach could contribute to the understanding of software design and the role it plays in the software development process and provide a motivating approach for learning software design. In Chapter 11, we presented a workshop approach to address the difficulties students have with understanding the relationship between the different phases of software development and software modelling. By performing short sprints, inspired by the LEGO scrum workshop [76], students delivered iterations of their software designs. Because the sprints were time-boxed, the students were confronted with their design decisions almost instantly. By using a follow up questionnaire we used the students' self reflection as a measure of their understanding of modelling in a software development process. The students indicated they are better at relating the different UML diagrams with the different phases of software development. By choosing the form of a short workshop we aimed to let students discover the overall overview of a software project and the (UML) models that can support the different stages of software development. The students judged that they learned from the workshop and suggested to use this form of exercising more often.

12.5.3 Recommendations for Teaching Software Design

This section often refers to recommendations for education that can be found in Table A.1

In this section we summarise recommendations for teaching software design distilled from the research of RQ3.

We believe that the learning of software design requires practising of software design (Table A.1 Recommendation 5). This should be facilitated by a tool that supports the learning by doing approach. As explored with the AoSD game, we recommend to breakdown the difficult topic of software design into smaller bits that are linked to common software design principles, such as coupling, cohesion or modularity (Table

A.1 Recommendations 18 and 19).

Although practising with isolated assignments can be helpful for addressing the learning of the syntax of a modelling language, we advice to also integrate the modelling of software designs into the overall software development process (Table A.1 Recommendation 34). Especially with the use of UML and the multiple ways a diagram can be used, it is important that students understand the purpose of the diagram in the software development process. In this way they are able to see the 'big picture'.

We experienced students are more engaged in software design activities when gamification is applied (Table A.1 Recommendation 27). Game mechanics that give students freedom of choice, challenges, competition and feedback are engaging students in the software development process. As seen with AoSD, UML modelling can be integrated well into a game-based learning approach.

Use modelling tools in a simulation (workshop) of the software development cycle. Students can judge about their own understanding of modelling during the development process they experience (Table A.1 Recommendations 7, 13 and 19). They will be able to discover their understanding of the difference between analysis and design, and the purpose of the different diagrams that can be used during the different phases of software development.

12.6 Supportive Tooling

In this research we developed various types of online tools support for conducting our research. In this section we explain how the two most important tools support the different research questions of this dissertation. In addition we close this section with recommendations for teaching software design that are distilled from the experience of working with these tools.

12.6.1 Design Skills Test

In our research we were in the need for a tool that was easy to scale up. If we wanted to be able to perform statistical analysis on our data, we needed to be equipped with a tool that could easily handle larger groups of students. We used Limesurvey [1] as a basis. Limesurvey is a web based survey tool. We needed a tool that met the following requirements:

[1] https://www.limesurvey.org

- time restriction (we added this to Limesurvey)
- statistical data (e.g. right / wrong answers, overall time spend, time spend per question)
- combine question text with images
- anonymous answers - this was provided by a unique survey token
- export of data for further statistical analysis (e.g. in R or SPSS)

The design skills test supports RQ1 and was used as a diagnostic tool in several of the classroom settings were research was performed for this dissertation.

12.6.2 WebUML

WebUML is an online UML editor that enables us to monitor students while constructing their software design (Chapter 4). It was especially developed for educational purposes. *WebUML* was developed for better fulfilling students' needs. In different researches students mention the frustration of using modelling tools. Installation and configuration of tools takes a lot of the students' time and does not always yield a working set-up. By using an easy to install application in combination with a simplified subset of UML we aim to contribute to the students' motivation for learning.

Using this instrument we could analyse several modelling - and design activities, such as creation, deletion and editing of UML elements. These modelling activities are recorded by a logging system that is part of the WebUML editor. In addition to our gained insights about the comprehension level of software design, our aim with *WebUML* was to collect data about the modelling/design process of students. With this data we were able to analyse strategies students use to create their designs.

We equipped *WebUML* with the capability of logging students activities and recording students' questions. With these features we aim to monitor students strategies and try to identify difficulties. Lecturers can give their feedback to the students, based on the recorded data. By supporting the student's learning process we aim to keep students motivated. By choosing an online platform *WebUML* fits well in modern tools and online platforms, such as MOOCs.

WebUML had to fulfil the following requirements:

- handling of large scale groups

- monitoring of student software design activities, such as creation of UML elements and edits of previous actions, and storing log files
- tool interaction
- extensible for future modules that interpret student solutions, such as the feedback agent (Chapter 10)

WebUML supports the research of Chapters 5, 6, 7, 9 and 10.

While using WebUML during our experiments we noticed that the tool was easy to use after a small set of instructions for both students and lecturers. In experiments, we were able to scale up to larger groups of students.

While the online test was focused on testing skills based on knowledge or comprehension, WebUML facilitated the judgement of the students' design process.

Recommendations for Teaching Software Design *This section often refers to recommendations for education that can be found in Table A.1*

In this section we summarise recommendations for teaching software design learned from the use of different tools.

Teachers should offer students an easy to use tool that uses a simple subset of UML. WebUML enables students to practice software design assignments in a simple environment (Table A.1 Recommendation 5). In addition, we recommend a feedback mechanism, such as our pedagogical agent for providing feedback during practising designing (Table A.1 Recommendations 7 and 9).

Students give up on using modelling tools because of complex installation and configuration problems. Industrial tools more often frustrate students because of their steep learning curve. Courses on Software Design should use tools that are easy to install and easy to use (Table A.1 Recommendation 33). We advise to use a small subset of the UML notation when teaching novices. In this way we stay away from specific object-oriented dependent principles and focus on general software design principles.

12.7 Future Work

For future research we propose to refine our online test so that the regression model can be used by lecturers to test their students and to choose appropriate interventions.

One of the directions that could be interesting to pursue is to use the test as a diagnostic instrument for students.

In addition, for further research in assessing students' software design capabilities, an extended version of our online test could be explored. This can be done by integrating WebUML and the online test. With this extension also the synthetic skills for creating a software design can be assessed. We propose to set up a similar experiment as described in Chapter 2 with a new set of questions that target synthetic skills.

To integrate WebUML into other systems, it should be extended to better fit users needs. We want to test our tools as part of other educational suites, i.e. e-learning or MOOC-like environments. For example as suggested by Vesin et al. ([155]) In general more research should be conducted on the use of modelling tools in education. WebUML could be enriched with the outcomes of this research.

Further research is needed to demonstrate the learning effect of our game based approach. We suggest to conduct a study that uses 'The Art of Software Design' to gain more insights into the validity of the scoring metrics that were used and the assumed learning effect.

To let students learn from the experience to go through a complete software development process, the agile UML modelling workshop could be extended to also include code generation or coding steps. In this way, students also reflect on the final product. Moreover, even maintenance steps could be integrated.

Future research should continue to explore students' strategies and difficulties in more detail, with the aim to develop educational programs in the field of software design that fit students' profiles better. Approaches have to be found that reveal more of the detailed reasoning and decision making steps that students take during their design-effort. For example, a combination of peer-reflection sessions, performing think-out-loud and follow-up interviews about the students' decision making could lead to new insights.

We see a huge opportunity for automated guiding and grading feedback systems in online learning environments. The explored machine learning and pedagogical agent approaches could be developed to a mature didactic tool which improves the quality of self assessment as part of self-study material and/or environments.