Studies into interactive didactic approaches for learning software design using UML

Stikkolorum, D.R.

Part I

Introduction, Context and Problem Statement

Chapter 1

Introduction – Challenges in Software Design Education

In this chapter we introduce the research context, our main objective and research questions. Further we present our research methods. In addition, we provide an overview of the structure of this dissertation. Furthermore this chapter concludes with an overview of the published work that this dissertation is based on as well as work that was published during the research period of this dissertation.

1.1 Problem Context – Education in Software Design

Software Engineering (SE) is a rapidly changing discipline. Software developers have to respond and adapt to constantly changing hardware and implementation platforms in a society with increasing challenges [94]. Therefore, software developers are challenged to create increasingly complex systems. The effective development and maintainability of such complex systems requires well thought out designs.

Software designs are the 'blueprints' for the system that is going to be delivered. The designs lay the foundation for later implementation steps. Software designs are expressed with software models. They are the means for communicating designs. In software engineering it is common to use visual representations for models. A collection of diagrams that focus on different views of a system (e.g. structure and behaviour) represent the system's model.

After its introduction in the 1990's the Unified Modelling Language[1] (UML) became the de facto standard for the modelling of software designs, and most universities started introducing UML in the classroom at the end of the 1990's. UML integrated the notations of the Booch method [20], object-oriented modelling technique (Rumbaugh et al.) [115] and object-oriented software engineering (OOSE, Jacobson) [58] into one language. UML consists of a collection of diagrams that visually express (parts of) a system's model. From this set, class diagrams are the most used diagrams for modelling software designs.

Even when students have an understanding of the syntax of a modelling language, the application of the language for creating a software design remains challenging. In particular software design is considered to be a difficult problem solving task, because it heavily depends on abstraction skills [69]. Students are constantly challenged to identify concepts from the problem domain and translate them to software artefacts. This, combined with the different views and abstraction levels that models can have, makes the understanding and application of modelling techniques challenging for students.

Educators face challenges at different levels:

- motivate and engage students in their courses,
- facilitate and use appropriate tools, and
- give feedback to students at the right level at the right time.

For helping educators to overcome the above mentioned challenges, we focus in our research on the involvement of education in the development of students' reasoning skills, students' software design strategies and the use of appropriate modelling and learning tools during software design tasks.

1.2 Background - UML, Design Principles, Tools, Agile Development and Theories of Education and Learning

This section discusses UML, the use of modelling tools, the application of common software design principles during software design, the increasing popularity of agile development and theories of education and learning.

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

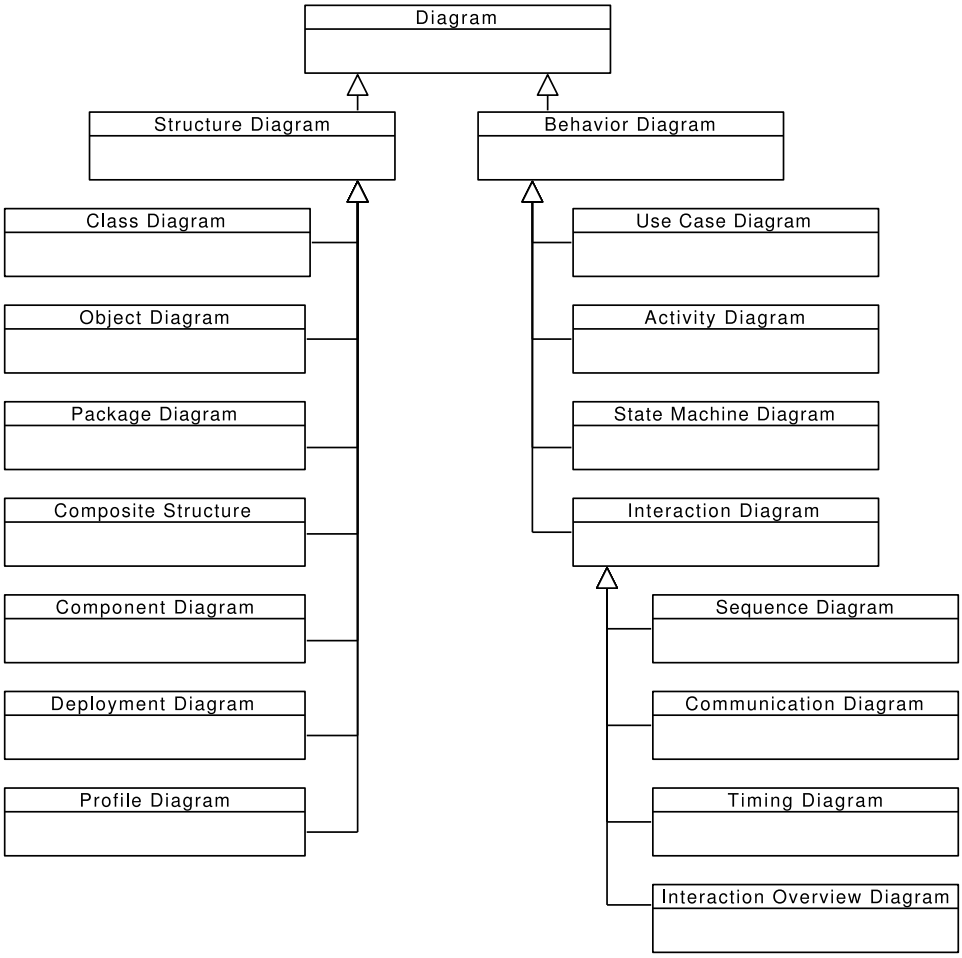

Figure 1.1: *Simplified diagram overview of UML 2.5.*

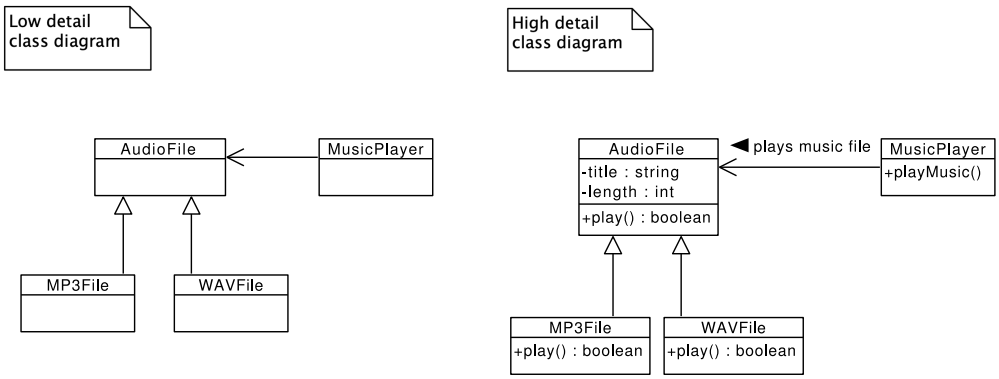

Figure 1.2: *A UML class diagram in low detail and in high detail*

1.2.1 Unified Modelling Language

UML is widely taught at universities and can be considered as the de facto standard for object-oriented modelling in SE courses. The universities that were involved in the studies that are discussed in this dissertation also use UML as their standard language for modelling during software development (analysis and design).

UML consists of several diagrams that can be used during analysis and design in software development methodologies. The diagrams describe mainly two types of views: structural views and behavioural views (see Figure 1.1).

The most commonly used diagrams from the *Structure Diagram* set are: the Class Diagram and the Component Diagram. From the *Behaviour Diagram* set the most commonly used are: the Use Case Diagram, the Activity Diagram, the State Machine Diagram and the Sequence Diagram [35]. In this research we mainly focus on the structural perspective of the design which is typically presented through class diagrams.

Class diagrams describe the (internal) structure of problem domains during analysis or software systems during design. In this dissertation we focus on the design phase(s). The diagram structure consists of related classes that can have attributes and operations. Eventually classes can be programmed with object-oriented software languages such as C++, Java or for example C#. Class diagrams are considered to be the blueprint of (parts of) the to-be system. Depending on the development stage or purpose of the diagram, a class diagram can have a low amount of detail or a high amount of detail. Figure 1.2 shows an example of a low detailed and high detailed class diagram of the same system.

1.2.2 Software Design Principles

In software development certain design principles are applied. A software design principle is a guideline for decision making during software development. Eventually it supports several benefits, such as:

- organised software (e.g. readable code)
- maintainable software
- changeable and extensible software
- less software bugs

In general such principles steer away from bad design constructions and thus aim to contribute to a better quality of the software.

An overview of key works in the field of software engineering that explicitly discuss design principles [20, 88, 73] are presented in Figure 1.3. The overview also shows the design principles that we consider as a generic design principle. These design principles do not only apply to object-oriented design but also to other design paradigms and software design in general.

In our research we use design principles as a basis for measuring students' design skills (see Chapter 2) and to evaluate the quality of their software design models. In this dissertation the following design principles play an important role:

- abstraction – define the conceptual boundaries (outside view) of an object by focusing on the characteristics of that object related to the context of a problem. This separates the object's expected behaviour from its implementation.
- modularity – split your software into separate units that are responsible for a specific task.
- low coupling – try to create less dependencies as possible to other software units. This lowers the amount of modifications in other (related) units when a unit has to be modified.
- high cohesion – software units should fulfil tasks and contain data that belong together.

We believe that this collection of principles constitutes a generic basis of the large collection of design principles. Moreover these principles are not specific to any programming language or paradigm.

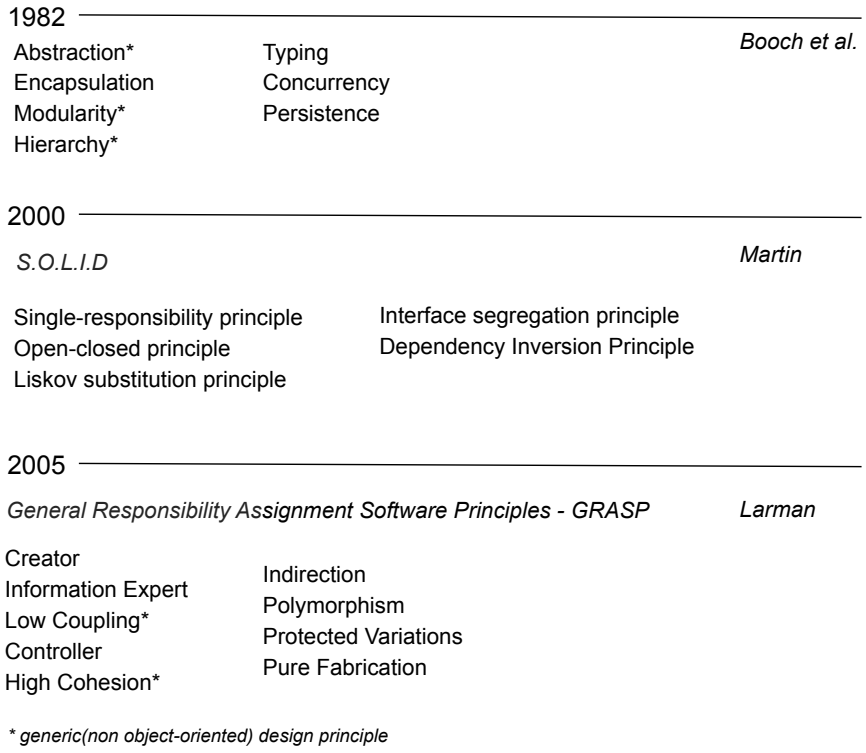

Figure 1.3: *An overview of the widely known software design principles.*

Unfortunately an optimal design is not achieved by optimising all principles individually. For example: lowering the coupling could result in lower cohesion. The challenge for software designers is to make design decisions that create an optimal balance of the applied design principles. Mastering this skill requires training and experience.

1.2.3 Tooling

There is an ongoing discussion going on about the use of (modelling) tools in education. Examples of topics of these discussions are: the difficulty of using tools, tools support and standard industry tools versus tailored tools for education [2, 83]. In our research we have looked at industrial modelling tools and educational tools (including serious games) or a combination of both (see Chapters 3 and 4)

1.2.4 Agile Software Development

In the past decades, there is an increasing popularity of evolutionary development methodologies such as agile. Instead of assuming that the set of requirements is clear from the start of a project, agile processes plan for very frequent feedback from stakeholders on the direction of the project. Openness to change is central to the way that agile development processes are organised. The consequence of such a flexible approach is that also the design process should be organised in such a flexible and incremental way. This requires for example the ability to evolve designs instead of 'freezing' a design from the start of a project. We explored whether designing software in an agile way contributes to the understanding of design concepts and to the understanding of design concepts in relation to the different software development phases (see Chapter 11).

1.2.5 Theories of Education and Learning

In this section we discuss related theoretical background on education and learning in general and related to software engineering. Based on the educators' challenges that were mentioned in Section 1.1 we discuss the following topics: course creation and the levels of learning, student motivation and didactic forms.

Creation of Courses and Learning levels Educators have to consider different matters when creating programs and courses within these programs. Matters such as:

- The main learning objectives of a program.
- The main learning objectives of a course.
- Smaller learning goals for lectures, workshops or other didactic moments.
- The order of courses and topics within courses.
- The increasing competence (e.g. knowledge and skills) levels (e.g. complexity).
- The prior knowledge of the students.

In addition, the field of software engineering develops constantly. Therefore programs are constantly subject to change.

Different taxonomies are supporting educators all over the world for designing learning programs. The following three are well know and most applied:

- Bloom's Taxonomy of Educational Objectives (revised by Anderson in 2001) [70] – a taxonomy that consists of three hierarchical models related to a domain: cognitive (knowledge-based), affective (emotion-based), psycho motor (action-based). The levels per domain build upon each other. The cognitive domain model is most used by educators for creating their courses. Later Anderson et al. revised the taxonomy. Amongst other things they added a dimension to knowledge, the kind of knowledge: factual, conceptual, procedural and meta-cognitive. This is the version that most people refer to as *Bloom' taxonomy*

- Fink's Taxonomy of Significant Learning [38] – a taxonomy that is not hierarchically organised. Each element in the taxonomy interacts with one and other. Like Bloom, Fink also has an affective part, but integrates this in the model. In addition Fink also recognises the importance of the meta-cognitive knowledge level.

- Wiggings and McTighe's Six Facets of Understanding [161] – like Fink a non hierarchical framework that is student centred. By examining six facets of understanding this should give a holistic picture of a student's understanding about a certain topic. The facets are examined with the help of questions to the student (e.g. 'What are examples of ... ?' or 'How might ... feel about ... ?'). You could say that this framework also integrates the affective area that Bloom handles as a separate domain.

Because of its popularity and the fact that the universities that collaborated with us in our studies also use Bloom's Taxonomy we chose to stay with Bloom's cognitive domain taxonomy.

Student Motivation As mentioned in Section 1.1 there is a challenge in motivating students for learning software design. Irvine maps several theories that are related to motivation in education onto two major theories: expectancy-value and intrinsic-extrinsic motivation [57]. Expectancy-value theory states that peoples' motivation depends on the belief that they have for being successful in a certain task and the value the task has for them. Intrinsic-extrinsic motivation theory states that people are motivated because: they are inherently interested in something or enjoy it; or, it leads to a distinct outcome (e.g. is/feels rewarded on success).

Literature shows [57] that when students are engaged in learning because of intrinsic motivation they develop a positive attitude toward the topic that was learned. While software design is an ever evolving discipline it is important for professionals to keep their knowledge and skills up to date. Intrinsically motivated students would become the life long learners that the industry needs.

It seems that intrinsic motivation can be influenced by educators. Positive feedback could enhance a student's motivation, while negative feedback would reduce it.

When people experience flow (an intense state, when a student is fully immersed in a task and enjoys it highly) it has an enormous positive impact on learning. Flow is the result of the balance between skills and challenge. An optimal situation is when skills and challenge are similar. When aware educators can use this for developing material. Flow is also a well-know term in games. Educational games could therefore be a good means of educating students.

In this research we explored the use of games and gamification and tailored tooling in relationship with students' intrinsic motivation and live long learning.

Didactic Forms In 2007 researchers state that constructivism (Piaget, later Papert: constructionism) is the most popular approach in computer science education [86]. Currently this is still the case. Although many didactic approaches are used, they mostly have their origins in constructivism.

In general constructivism states that learning occurs when students are actively involved in the process of gaining knowledge. The practical approach that computer science programs offer fits the idea of active learning.

Researchers observed a shift from traditional teaching towards a more active and practical approach. In general the following approaches are used in the field of software engineering [87]:

- traditional – classroom teaching and lab sessions (constrained assignment under supervision of a TA and/or lecturer).
- learning by doing – experience of clash of theory and practice by taking on a project.
- case study – investigation of a real-life problem that is recorded in one or several sources (e.d. documents, video's, etc.)
- problem- / outcome based learning(PBL) – a practical problem as driver for the learning process.

Based on a literature review in the field of teaching computational thinking (k-12 and university level) the researchers show that in the reviewed studies the following approaches stand out: game-based learning, collaborative learning, problem-based learning and project based learning as an answer on the need for another approach of

teaching in this area [56]. The authors describe computational thinking as 'a process of problem-solving'. We see problem solving as an elementary skill for software design.

The didactic forms that were applied in the cases that this dissertation discusses were mainly practical and were aimed for training problem solving skills and for revealing students' reasoning to themselves and to the researchers.

1.3 Problem Statement

Several researchers point out that a large number of students have difficulties with (object-oriented) modelling during analysis and design of software systems [69, 50, 122, 62]. Students have to elicit concepts from the problem domain, transfer the concept of this domain to the solution space and propose software structures that should lead to maintainable software. These tasks require abstract reasoning skills and these skills seem to be difficult to teach. It is not well understood what the hurdles are in students reasoning while creating software designs. In this dissertation we explore the characteristics of students' learning in order to improve our educational approaches.

Some initial thoughts about the sources of the difficulties students face are:

- Modelling can be done in different phases of software development such as during analysis and design. During analysis students explore and model the problem. They are required not to 'think' about implementation details and see the 'big picture'. In contrast, when they are designing a solution they have to introduce implementation details into their designs. For students (and novice software developers) it is difficult to stay away from implementation. Especially when the solution domain is familiar for them. In addition it is also difficult for them to be aware of the development phase they are in. The fact that in software development the same UML notation (although with detail difference) is used for modelling in different phases confuses students.
- Modelling languages introduce the possibility of having multiple views of software systems (e.g. structural view and behavioural views, but also object view versus class/concept view). This seems a great feature, but is challenging for students. Identifying important concepts in the problem domain and creating extensible designs already requires a lot of abstraction power of the brain. Switching between views requires a high memory load for related diagrams (e.g. state machine diagrams that relate to instances (objects) from class diagrams).
- Modelling languages introduce different levels of abstraction. UML for example offers compressed versions of notations (e.g. a class without any detail, only the

name) and applies diagrams on different 'zoom levels' of a system (e.g. sequence diagrams on system levels and sequence diagrams on object level). We observe that students have difficulties to distinguish between abstraction levels and their purpose.

- A difference between modelling and programming is that with programming students have the possibility of continuous feedback by means of testing and compilation. This may positively affect the motivation for programming. With modelling this is not the case. Although it is possible to model in high detail, because of the general nature of (in our case) UML there is a gap between the model and the implementation. An executable model is therefore mostly not possible. We are aware of model driven environments such as low/no code platforms where the model is the implementation. But they are often developed for a specific problem (context).

Now and even more in the future, we depend on and trust software systems. At the same time these systems are not only growing in size, but also in complexity. Students should be well prepared and capable of handling these complexities. To keep future systems maintainable, modelling should play an important role [94].

1.4 Research Objective and Research Questions

This section presents the main research objective followed by the research questions of this dissertation.

The main objective of this dissertation focuses on how to improve our approaches in software design education:

Main Objective *How can we improve the ways of teaching software design?*

Not a lot research has been done on the particular topic of software design education, therefore we are motivated to contribute to this.

Education can be explored in many different ways. In our research we focus on the understanding of students' reasoning and decision making. This dissertation discusses three aspects of the educational process of students:

1. students' comprehension of software design,
2. students' motivation for learning software design, and

3. the learning process of students during software design tasks.

In support of the *Main Objective* the following research questions are defined:

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?

Section 1.6 presents an overview that maps the chapters of this dissertation onto the research questions.

1.5 Research Approach

The research questions of this dissertation are answered with the use of empirical research. Empirical research gains knowledge by means of observation and experimentation. Empirical comes from the ancient Greek word for experience: έμπειρία (empeiría). Empirical research methods typically study real-world phenomena, such as human activities. Empirical research is being applied in several kinds of scientific fields, such as physics, psychology and medicine.

In the field of software engineering (SE) this research type is relatively young. Perry et al. discussed the state of empirical research in the SE domain in the year 2000 [102] and stated it to be relatively immature. Two years later Kitchenham et al. suggested preliminary guidelines for empirical research in the field of software engineering [65]. They suggested guidelines for six basic topic areas: experimental context, experimental design, conduct of the experiment and data collection, analysis of the results, presentation of the results, and interpretation of the results. Two examples of guidelines from Kitchenham et al. are:

> *If the research is exploratory, state clearly and, prior to data analysis, what questions the investigation is intended to address and how it will address them* – **experimental context**
>
> *Define the process by which the subjects and objects were selected* – **experimental design**

Currently empirical research is embraced more widely. The *International Conference on Model Driven Engineering Languages and Systems (MODELS)* publishes empirical work and has hosted the *International Workshop on Human Factors in Modelling* that was initiated to be a venue for empirical research involving human factors in modelling. Empirical work is also published by *The International Conference on Software Engineering (ICSE)*. This conference also organises the *International Workshop on Conducting Empirical Studies in Industry*. Other parties that embrace empirical studies in software design are: The *Computer Science Education Research Conference (CSERC)* and *Special Interest Group Computer Science Education (SIGCSE)*

In an empirical research, the collected data is analysed qualitatively or quantitatively, depending on the chosen method. The methods that were used for our research were experiments and case studies.

In this dissertation our main subjects are students. We investigate students' learning activities. An empirical approach is therefore suitable.

1.5.1 Experiment

An experiment is set up to measure a specific effect that was caused by the change of an (independent) variable. In most experiments the researcher wants to test his/her prediction (hypothesis). Data collection can be done with the use of a survey, or with measurements. Quantitative analysis is dominant in the experiment research method. In our research the experiments did not take place in a laboratory, they were *field experiments*. The experiment method is used in Chapters 2, 5, and 9.

1.5.2 Case Study

In a case study a group or person is studied in a particular situation. The researcher has an observing role. The data collection is often done by interviewing the subjects or let them 'think aloud' [81]. Qualitative analysis is dominant in case study researches. Often the analysis is used to create a theory or provoke a discussion. The case study method is used in Chapters 3, 7, and 8.

1.5.3 Mixed Methods

Some studies require a combination of qualitative and quantitative analysis: mixed methods. The mixed methods approach is used in Chapters 6, and 11.

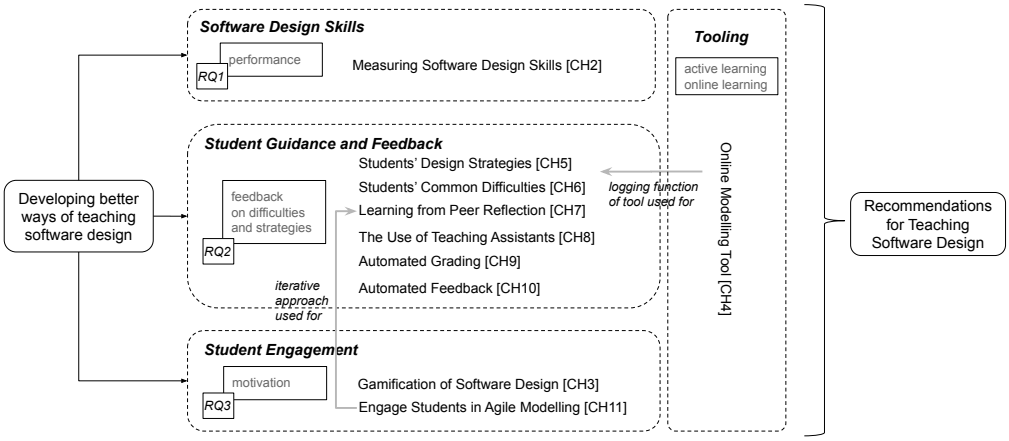

Figure 1.4: *Overview of the research themes and their corresponding chapters.*

1.6 Contribution and Dissertation Structure

This research contributes to the improvement of software design education. The contribution is five-fold. First this dissertation provides a method for assessing students' software design skills. Evidence is found for: i) measuring the learning yield of students during a software design course and ii) the relation between abstract reasoning performance and the performance of students on software design tasks.

Second we identified engaging educational approaches and tools for learning software design. We showed this by using gamification and the integration of modelling in agile software development processes. We developed a modelling tool that enables on-line learning and equipped the research with the possibility to investigate larger groups of study subjects.

Thirdly our research contributes to the collection of common design strategies and difficulties in students' reasoning processes. With the use of the investigated educational approaches we gained insights in the type of guidance or feedback students need and we are able to make recommendations to lecturers in the field of software engineering.

Fourthly we explored the automation of grading and feedback for providing direct and/or on demand feedback for students. Based on the results of our approach that uses machine learning for grading we conclude that the models of the experiment were not accurate for 10 point grading, but may be used as a rough quality indicator of the quality of a software design. We developed a feedback agent module for WebUML to explore automated guiding when students perform software design tasks. Based on

the early findings, we see a future for integrating feedback agents into future learning environments.

Lastly, with our findings we aim to contribute to research approaches and tools for studying students doing software design.

Figure 1.4 shows the set of ideas and approaches we have developed. We started off with a tool for assessing student's knowledge of software design (using classical multiple choice testing). As a complement, we moved into the area of monitoring and assessing students while *doing* design. In that area, we found that feedback during the doing of design is a good aid in learning; i.e. instead of assessing for the purpose of grading, we looked into assessing for the purpose of providing feedback and thereby enhancing the learning. In line with this theme, we also looked into enhancing the engagement of students in the learning of software design. In addition Figure 1.4 illustrates the relationships between the contribution themes and the chapters of this dissertation. Some chapters relate to previous chapters. This is also illustrated in the figure.

The structure of the remainder of this dissertation is as follows:

Part II - Software Design Comprehension

Chapter 2 – An Instrument for Measuring Design Skills The main objective of this chapter is i) to create an instrument that assess students' software design skills and ii) investigate the relation between abstract thinking abilities of students and performance of software design tasks. (supports RQ1)

Part III - Tooling

Chapter 3 – Exploring the Application of Game-Based Learning in Software Design Education The main objective of this chapter is to explore the influence gamification has on the engagement of students in software design activities. (supports RQ3)

Chapter 4 – WebUML - a UML Class Diagram Editor for Research and Online Education The main objective of this chapter is to present our on-line UML editor. The editor is a research instrument that was needed to conduct the several experiments that are discussed in the different chapters of this dissertation.

Part IV - Student Guidance and Feedback

Chapter 5 – Strategies of Students Performing Design Tasks The main objective of this chapter is to reveal common strategies students use to solve their software design tasks. (supports RQ2)

Chapter 6 – Uncovering Common Difficulties of Students Learning Software Design The main objective of this chapter is to uncover difficulties students have during a software design task. (supports RQ2)

Chapter 7 – Teaching of Agile UML Modelling: Recommendations from Students' Reflections The main objective of this chapter is to distil students difficulties in the process of analysis and design modelling by analysing their peer-reflections. From the distilled difficulties recommendations for the guidance of software engineering students are presented (supports RQ2)

Chapter 8 – Evaluating Didactic Approaches used by Teaching Assistants for Software Analysis and Design using UML The main objective of this chapter is to create a profile for teaching assistants that are involved in the education of software design courses. (supports RQ2)

Chapter 9 – Towards Automated Grading of UML Class Diagrams with Machine Learning The main objective of this chapter is to explore the application of machine learning for automated grading of class diagrams. (supports RQ2)

Chapter 10 – An Online Educational Agent: Automated Feedback for Designing UML Class Diagrams The main objective of this chapter is to explore the application of automated feedback during software design tasks with a didactic software agent.(supports RQ2)

Part V - Student Engagement

Chapter 11 – A Workshop for Agile Modelling The main objective of this chapter is to explore the integration of UML modelling in an agile development process in order to engage students in software modelling. (supports RQ3)

Part VI - Reflection

Chapter 12 – Conclusion and Future Work The closing chapter of this dissertation discusses recommendations based on pitfalls and practices of software design education, conclude this research and propose future work.

1.7 Publications

Below we list a chronological list of publications that were (co-)authored during this doctoral research:

1. Dave R. Stikkolorum, Michel R.V. Chaudron, and Oswald de Bruin. The art of software design, a video game for learning software design principles. In *Gamification Contest MODELS'12 Innsbruck*, 2012

2. Hafeez Osman, Arjan van Zadelhoff, Dave R. Stikkolorum, and Michel R.V. Chaudron. UML class diagram simplification: what is in the developer's mind? In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, page 5. ACM, 2012

3. Dave R. Stikkolorum, Claire Stevenson, and Michel R.V. Chaudron. Assessing software design skills and their relation with reasoning skills. In *EduSymp 2013. CEUR, vol. 1134, paper 5*, 2013

4. Seiko Akayama, Marion Brandsteidl, Birgit Demuth, Kenji Hisazumi, Timothy C Lethbridge, Perdita Stevens, and Dave R. Stikkolorum. Tool use in software modelling education: state of the art and research directions. In *the Educators' Symposium co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, 2013

5. Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty HC Cheng, Philippe Collet, Benoit Combemale, Robert B France, Rogardt Heldal, James Hill, et al. The relevance of model-driven engineering thirty years from now. In *Model-Driven Engineering Languages and Systems*, pages 183–200. Springer, 2014

6. Dave R. Stikkolorum, Birgit Demuth, Vadim Zaytsev, Frédéric Boulanger, and Jeff Gray. The MOOC hype: Can we ignore it? In *Reflections on the current use of massive open online courses in software modeling education. MODELS Educators Symposium, EduSymp*, 2014

7. Dave R. Stikkolorum, Truong Ho-Quang, and Michel R.V. Chaudron. Revealing students' UML class diagram modelling strategies with WebUML and LogViz. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 275–279. IEEE, 2015

8. Bilal Karasneh, Dave R. Stikkolorum, Enrique Larios, and Michel R.V. Chaudron. Quality assessment of UML class diagrams. In *Proc. Educators' Symp at MoDELS*, 2015

9. Dave R. Stikkolorum, Truong Ho-Quang, Bilal Karashneh, and Michel R.V. Chaudron. Uncovering students' common difficulties and strategies during a class diagram design process: an online experiment. In *Educators Symposium 2015, co-located with the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, 2015

10. Dave R. Stikkolorum and Michel R.V. Chaudron. A workshop for integrating UML modelling and agile development in the classroom. In *Proceedings of the Computer Science Education Research Conference 2016*, pages 4–11. ACM, 2016

11. Dave R. Stikkolorum and Michel R.V. Chaudron. Teaching of agile uml modelling: Recommendations from students' reflections. In *Proceedings of the 20th Ibero-American Conference on Software Engineering, Buenos Aires, Argentina*, 2017

12. Dave R. Stikkolorum, F. Gomes de Oliveira Neto, and Michel R.V. Chaudron. Evaluating didactic approaches used by teaching assistants for software analysis and design using uml. In *Proceedings of the 3rd European Conference of Software Engineering Education*, ECSEE'18, pages 122–131, New York, NY, USA, 2018. ACM

13. Boban Vesin, Aleksandra Klašnja-Milićević, Katerina Mangaroska, Mirjana Ivanović, Rodi Jolak, Dave Stikkolorum, and Michel Chaudron. Web-based educational ecosystem for automatization of teaching process and assessment of students. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, pages 1–9, 2018

14. Dave R. Stikkolorum, Peter van der Putten, Caroline Sperandio, and Michel R.V. Chaudron. Towards automated grading of uml class diagrams with machine learning. In *BNAIC/BENELEARN*, 2019

15. Goda Jusaite, Pim Sanders, Damani Lawson, Koen van Polanen, Hani Al-Ers, and Dave R. Stikkolorum. Improving the quality of online collaborative learning for software engineering students. In *International Academic Conference on Teaching, Learning and E-learning*, pages 8–15, 2020

16. Marcella Veldthuis, Matthijs Koning, and Dave R. Stikkolorum. A quest to engage computer science students:using dungeons & dragons for developing soft skills. In *Proceedings of the Computer Science Education Research Conference 2021*. ACM, 2021

In addition, the author has chaired symposia and conferences in the area of Modelling and Education :

1. Birgit Demuth and Dave R. Stikkolorum, editors. *Proceedings of the MODELS Educators Symposium co-located with the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 29, 2014*, volume 1346 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014

2. Ebrahim Rahimi and Dave R. Stikkolorum, editors. *CSERC '19: Proceedings of the 8th Computer Science Education Research Conference*, New York, NY, USA, 2019. Association for Computing Machinery

3. Dave R. Stikkolorum and Ebrahim Rahimi, editors. *CSERC '20: Proceedings of the 9th Computer Science Education Research Conference*, New York, NY, USA, 2020. Association for Computing Machinery

4. Dave R. Stikkolorum and Ebrahim Rahimi, editors. *CSERC '21: Proceedings of the 10th Computer Science Education Research Conference*, New York, NY, USA, 2021. Association for Computing Machinery

5. Dave R. Stikkolorum and Ebrahim Rahimi, editors. *CSERC '22: Proceedings of the 11th Computer Science Education Research Conference*, New York, NY, USA, 2022. Association for Computing Machinery