Studies into interactive didactic approaches for learning software design using UML

Stikkolorum, D.R.

About the Author

Dave Stikkolorum was born (1976) and raised in the The Hague-region of The Netherlands. After high school he studied electrical engineering with 'technical computer science' as major. After his studies, in 2001, he decided to start as a part-time software developer and as a secondary school teacher in mathematics. He traded the job of software developer for a job as teacher at The Hague University of Applied Sciences (HHS). From 2005 Dave became a full-time member of the 'technical informatics' team at HHS. Because of an interest in the education of software design he started his PhD research at the Leiden Institute of Advanced Computer Science (LIACS, Leiden University). The research was conducted under the supervision of Michel Chaudron and later accompanied by Peter van der Putten. Currently Dave is involved as program director of HBO-ICT at The Hague University of Applied Sciences and teaches in the Game Development and Simulation program.

Appendices

Appendix A

Overview Recommendations for Teaching Software Modelling

Table A.1: *Overview Recommendations for Teaching Software Modelling*

Student Challenges	(Education of) Theory	Tool	Skills		Process Skills
			Abstraction	Bloom levels	
Analysis	**Problem 1**: Students have difficulties with the analysis of text based assignments. Because of incomplete instructions students tend to loose the overall overview. **Recommendation 1**: keep instruction text simple when students start to learn and practice software analysis. **Recommendation 2**: When it is need for practice to have incomplete or poor assignment texts, provide enough feedback moments and/or peer review sessions to overcome students wandering.	**Problem 2**: because UML uses the same notation for Analysis models as well as design models, most tools do not support analysis steps by having a special analysis mode. **Recommendation 3**: emphasise the analysis phase in which a student is modelling and use a simple feature set for making domain models. In best case a teacher can tailor the tool that is used.	**Problem 3**: Students have difficulties to identify the relevant concepts from a problem (domain) in order to abstract them and present them in a (UML) diagram. **Recommendation 4**: use the noun identification technique to train the identification of relevant concepts.	**Problem 4**: Software analysis reaches the higher level bloom tasks: analyse (4) evaluate (5) and create (6). **Recommendation 5**: Students should practice with exercises that are not too complex. Using a domain that students are familiar with can be helpful.	**Problem 5**: Students have difficulties in reflecting to their own analysis models. **Recommendation 6**: students should be trained using an approach where analysis models iteratively evolve. **Recommendation 7**: organise process support were continuously feedback articulated by the lecturer and/or teaching assistants.

Design	**Problem 6**: Students have difficulties with determining how detailed a design model should be. **Recommendation 8**: Offer assignment that make clear which level of detail is expected from the student. **Problem 7**: Because of the open character of most assignments students have difficulties in deciding of their design are "done". **Recommendation 9**: Include feedback moments and design discussions during a lecture. **Recommendation 10**: Offer follow-up assignments that enable students to discover missing elements in their initial design.	**Problem 8**: because UML uses the same notation for analysis models as well as design models, most tools do not support design steps by having a special design mode. **Recommendation 11**: emphasise the design phase in which a student is modelling and use an increasing complex feature set for making domain models. In best case a teacher can tailor the tool that is used.	**Problem 9**: Students have difficulties to include relevant abstractions of the domain concepts (presented in a domain model) in their software design. **Problem 10**: Students have difficulties deciding between classes and attributes (abstraction). *To be explored further.*	idem*	**Problem 11**: Students have difficulties in reflecting to their own design models. **Recommendation 12**: students should be trained using an approach where designs iteratively evolve. **Recommendation 13**: organise process support were continuously feedback articulated by the lecturer and/or teaching assistants.

Analysis vs Design, Analysis and Design		**Problem 12**: there is no mature modelling tool that supports the transition from analysis to design **Recommendation 14**: develop tools that supports the transition from analysis to design **Recommendation 15**: integrate modelling tools with software that supports (agile) software development process tools.	**Problem 13**: Going from analysis to design covers going from a concrete problem domain to an abstraction followed by creating a design from 1) the gathered domain concept abstractions 2) the introduced software concepts from the solution domain. This combining of abstractions of two sources is difficult to grasp for students. **Recommendation 16**: Offer standard abstractions (responsibility driven design, Wirfs-Brock) and standard architectures (layered architecture) as a first start.	idem*	**Problem 14**: Students don't have experience with the development process and therefore find it difficult to see the relation between analysis and design. **Recommendation 17**: students should be trained using an approach where analysis and design are part of iterative cycles (agile UML workshop).

Design decisions	**Problem 15**: students have problems accepting that there are multiple solutions to a problem. **Recommendation 18**: use software design principles as a guide for decision making.	**Problem 16**: there is no mature modelling tool that supports design decision making. Most tools check on syntax level. **Recommendation 19**: develop tools that use feedback based on design principles. **Recommendation 20**: put a low emphasis on syntax.	**Problem 17**: UML offers a very high abstraction for generalisation of (software elements): class. **Recommendation 21**: Make use of stereotypes / roles when designing (responsibility driven design, Wirfs-Brock)	idem*	**Problem 18**: students have problems deciding when a design is better than another. **Recommendation 22**: use software design principles as a guide for decision making. Principles should be used when discussing (pair/collaborated modelling)

Design vs Implementation	**Problem 19**: Students do not see the effect of their design (decisions) on the implementation. There is a difference in time in the development process. **Recommendation 23**: Teach students an agile approach in modelling for having smaller steps in the development process that makes the effect of design decisions clearer.	**Problem 20**: Students don't update their design from the moment they start implementing in code. **Problem 21**: Most tools only offer one way code generation / import or only partial code generation / import. **Recommendation 24**: Develop tools that support complete round-trip code generation - code import (model generation). **Recommendation 25**: Find ways to keep track of code changes and updating students' designs.	**Problem 22**: there is a gap between the completeness of an 'abstract' design and the desired implementation. Students don't feel comfortable with the degree of freedom when they are novice software developers. *To be explored further.*	idem*	**Problem 23**: Students don't update their design from the moment they start implementing in code. **Problem 24**: Students neglect their models when implementing the software. **Recommendation 26**: students should be trained using an approach where design and implementation are part of iterative cycles (agile UML workshop).

Implementation Thinking	**Problem 25**: Because of the direct effect of implementation, students are not motivated to use modelling as a first step during analysis and design. **Recommendation 27**: Use motivating environments such as gaming ("The Art of Software Design") to engage students into modelling.		**Problem 26**: students have difficulties not to think in implementation when making initial designs. It distracts them from using their abstraction skills. *To be explored further.*		**Problem 27**: Students apply implementation knowledge too soon in the development process. *To be explored further.*

Model Complexity	**Problem 28**: Students have difficulties in understanding a design when the design consists of a lot of information. **Recommendation 28**: Consider the layout when preparing assignments. **Recommendation 29**: Teach according to a layout style (Scott Ambler)	**Problem 29**: Tools offer often more information than needed. **Recommendation 30**: When, available use auto layout and change level of detail		**Problem 30**: as model complexity grows, the cognitive load increases on every level. Often models are presented that consist of too many elements (rule of 7 +/- 2). **Recommendation 31**: offer students assignments with increasing complexity of the models. **Recommendation 32**: keep models simple and keep the amount of elements in a diagram low (7+/-2)	

General		**Problem 31**: Students find tools complex. They have a long and steep learning curve and most of the times difficult to install. **Recommendation 33**: Develop easy to use tools that don't require installation, like WebUML			**Problem 32**: Students have difficulties in understanding the role of the different UML diagrams in a development process. **Problem 33**: Students find it difficult to articulate their difficulties during an assignment. (they don't know what they don't know). **Problem 34**: Students do not automatically reconsider their designs. **Recommendation 34**: Train students to use their modelling activities with an agile approach (agile UML workshop), using pair modelling or peer reflection moments, in order to discover diagram relevancy within development cycles and to enable learning discussions. Discussion enables the re-thinking of the delivered design.

* *same kind of problem / solution*

Table A.1 summarises the problems identified in the research of this dissertation. Most of the problems have been investigated and are accompanied by a recommendation.

Appendix B

Coffee Machine Assignment

a Coffee Machine A coffee machine can make different types of coffee (one at a time): black, with sugar, with sugar and milk. The different types of coffee are poured into a cup. The front of the machine contains a pane with buttons. The buttons are used for selecting the desired drink. One can pay with a chip card or coins. The chipcard is read by a chip sensor and the coins by a coin sensor.

Appendix C

Tank Assignment

The Modelling Task – a Tank Game[1] In this task you will design a game with use of the UML class diagram. You do not need to use packages in this assignment. The description of the game is as follows: A player (user) controls a certain tank. This tank is a Panzer Tank, a Centurion Tank or a Sherman Tank. They fire bullets and Tank shells. Bullets can be Metal, Silver or Gold bullets.

A tank moves around a world (level). The aim is to destroy all other tanks in the world. After a world has been completed the tank advances to the next world. A list of all the worlds visited is kept.

An entire game consists of 8 levels. A world contains a maximum of 20 tanks that compete for victory. Each tank remembers which tanks it has destroyed in the past. The score for each level is kept by a scoreboard that gets notified by the individual tanks each time an opponent is shot. The players control their tanks through an interface allowing for steering, driving (reverse / forward), switching ammo and firing.

[1] text: B. Karasneh

Appendix D

Grading Rubric

Table D.1: *Class Diagram Rubric for Grading Design Modelling*

Grade	Judgement, criteria description
1	The student does not succeed to produce a UML diagram related to the task. He/she is not able to identify the important concepts from the problem domain (or only a small number of them) and name them in the solution/diagram. The diagram is poor and not/poorly related to problem description with a lot of errors: high number of wrong uses of UML elements mostly no detail in the form of attributes or operations.
2	The student is not able to capture the majority of the task using the UML notation. Most of the concepts from the problem domain are not identified. The detail, in the form of attributes or operations, linked to the problem domain is low. Some elements of the diagram link to the assignment, but too much errors are made: misplaced operation / attributes non cohesive classes few operation or attributes are used.
3	The student is able to understand the assignment task and to use UML notions to partly solve the problem. The student does not succeed to identify the most important concepts. A number of logical mistakes could have been made. Most of the problem is captured (not completely clear) with some errors: missing labels on associations missing a couple important classes / operations / attributes Logical mistakes that could have been made: wrong use of different types of relationships wrong (non logical) association of classes.
4	The student captures the assignment requirements well and is able to use UML notations in order to solve the problem. Almost all important concepts from the problem are identified. Some (trivial) mistakes have been made: Just one or two important classes / operations / attributes are missing Design could have been somewhat better (e.g. structure, detail) if the richness of the UML (e.g. inheritance) was used.
5	Student efficiently and effectively used the richness of UML to solve the assignment. The problem is clearly captured from the description. concepts from the domain / task are identified and properly named The elements of the problem are represented by cohesive, separate classes (supports modularity) with a single responsibility In the problem domain needed attributes and operations are present Multiplicity is used when appropriate Naming is well done (consistent and according to UML standard) Aggregation / Composition / Inheritance is well used No unnecessary relationships (high coupling) are included.

Appendix E

Class Diagram Evaluation Criteria

- Syntax – does the diagram follow the formal notation rules of the modelling language? At the time of writing UML 2.4 for example.
- Layout – Is the diagram readable? Does it not mix styles? For example: is an inheritance relationship drawn from top to bottom across the whole diagram?
- Consistency – is the diagram consistent with the other diagrams in the model? For example: does a class name corresponds with a lifeline in a sequence diagram?
- Semantics – Does the diagram represent what it should? For example: a '1 .. *' multiplicity is applied. Which means '1 to many'. Was this meant?
- Requirements satisfaction – does the diagram support the (functional) requirements? For example: Does the class diagram have a set of classes that cover the responsibilities of the system that is designed?
- Design principles – are common design principles applied? For example: did the student take coupling and cohesion into account?
- Design decisions – were the right design decisions made and applied? For example: a pattern is applied to support a quality requirement.