Studies into interactive didactic approaches for learning software design using UML

Stikkolorum, D.R.

Part III

Tooling

Chapter 3

Exploring the Application of Game-Based Learning in Software Design Education

Description of contributions of the authors
The research in this chapter was done in collaboration with Oswald de Bruin. At the time of writing Oswald followed the master program on computer science at Leiden University. He took part in the discussions about the game design. He designed the 'look and feel', implemented and tested the game.

The contributions of Dave Stikkolorum were: taking part in the discussions about the game design, research design and writing the publication. Michel Chaudron acted as supervisor to both Dave and Oswald on this project.

This chapter describes an exploratory study on the application of game based learning for learning software design. Based on software design principles we developed a game for training students in software design principles. The game aims to engage and motivate the player for learning, using guided feedback in a challenging level based puzzle game with increasing difficulty. We used Bloom's taxonomy to determine the learning objectives of the different game levels. The

This chapter is based on the following publication: 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

learning objectives let the students learn to cope with the complex balancing problem of interfering design principles during software design. Players are supported while evaluating their design actions by means of the visualisation of control- and data flows. The game based approach supports the educational approach of 'learning by doing'. Initial user tests indicate player's engagement and a possible learning effect.

3.1 Introduction

Designing software systems is one of the difficult tasks in the field of software engineering. It is no surprise that learning software design is a difficult task for students too [80][69][113][141]. They have to face the challenge of abstracting structure and behaviour for possible software solutions. Next to these abstraction skills, students have to learn to integrate them in a software development process. For learning software design skills, students have to practice much in order to get familiar with the tools and the application of their design skills in project settings.

In their literature study Marques et al. conclude there is a need for teaching software engineering in a practical way [87]. They report a shift in studies of teaching approaches from traditional approaches towards learning by doing. For software engineering they report that most studies (until 2014) focus on:

- *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.
- *games* – game development as fun element for learning.
- *traditional* – classroom teaching and lab sessions (constrained assignment under supervision of a TA and/or lecturer).

In our research and educational practice we learnt that universities offer a variety of approaches that enable students to practice as much as possible. Dominant approaches for learning software design are:

- *lab sessions* with practical assignments under the supervision of the lecturer and possibly teaching assistants. – *traditional*
- *group work sessions* with discussions (similar to lab sessions). – *PBL / traditional*
- *homework assignments* that are discussed during the lectures. – *traditional / flipped classroom*
- *projects* in which students have to deliver a solution for a more complex problem than home work assignments or lab session assignments. Projects can solve a real-life problem (even with a 'real' client) or students can be provided with an artificial problem. – *mixed approach*

As introduced in Chapter 1, there is a challenge for lecturers to motivate students for learning software design. A difficulty of learning software design, is that it heavily depends on abstract reasoning. This abstract nature makes it difficult to create assignments that appeal to the students and keep them engaged. On one hand lecturers have to create assignments that focus on a particular issue while in reality one will face the integration of multiple issues. On the other hand more complex problems can be addressed in project-based approaches but can be overwhelming for novice software designers and are often too complex for the classroom setting [23] (engaging students in complete software development processes is discussed in Chapter 11). With both of the approaches there is a risk that the students lack overview of the matter and therefore this could have a negative influence on the engagement.

In this chapter we explore an additional didactic approach for learning software design. We introduce the gamification of a software design environment as part of our bachelors' software engineering course. We aim to motivate students to learn software design by providing them with learning material in the form of an interactive computer game that relates to software design concepts and uses a graphical interface that relates to a modelling tool for making software designs. By slowly introducing concepts we aim to keep the students overview clear. A better overview should support the engagement of the student in a positive way.

In this chapter we discuss our gamification approach, give insight in the game itself and discuss early findings based on think aloud sessions and user testing.

In this chapter we use the term 'design' rather than 'modelling', because we see modelling as a vehicle for designing. The game uses modelling as a vehicle for

- explaining software design principles,
- discovering the usefulness of models for abstract reasoning about a design, and

- finding the proper abstractions for representing a system in a domain.

In this chapter we use the term 'learning by doing' in the sense of experience learning by practising by trial and error. We do not see this as the only right approach. We think that teachers should mix didactic approaches dependent on the context and needs of their students.

We mix the terms 'gamification' and 'game' in this chapter. In our opinion we gamified (applied gamification) the software design tasks by approaching the software design challenge students face as a puzzle. On one hand we applied gamification: we build and environment according to the analogy of an UML design tool and introduced scoring and progress elements. On the other hand we went a bit further and built a game: it contains levels, with clear level goals, feedback guides and gameplay. We build and educational puzzle game by strongly gamifying the software design task. The eventual product could be seen as a proof of concept.

The remainder of this chapter is structured as followed. In Section 3.2 we discuss related work. In Section 3.3 we describe our method, in Section 3.4 we describe the game design. After a 'walk through' of the game in Section 3.5 we evaluate and discuss in Sections 3.6 and 3.7. Finally we conclude and propose future work in Section 3.8.

3.2 Related Work

Prensky introduced Digital Game Based Learning [104]. He states: "Video games are not the enemy, but the best opportunity we have to engage our kids in real learning" [105]. He suggests to make use of the interest in (video) games of young people for the engagement in learning.

Currently the use of video games in education is widely applied by teachers from different levels of education. Souza et al. show in their literature study of 106 studies a variety of the application of game related approaches in the field of software engineering [126]. The categorised the didactic approaches as follows: Gamification, Game Development Based Learning, Game Based Learning and Hybrid. The authors found a predominance of the application of GDBL for the knowledge area of software design: students learn by developing a game.

Sheth et al. introduced a hybrid form: a code tournament and a gamification of the testing phase [121]. The authors aimed to engage students into software design and testing by the means of competition and collaboration. First the students were involved in a code tournament for learning to make good software design decisions.

The assignment was to create a battle ship game that uses a particular interface that was also used by an AI. The main objective was 'programming to an interface'. They had to compete by playing against the AI in a tournament. Second the students learned about testing by using a gamification of testing techniques such as white - and black box testing. Students are given a number of quests that have to be complete. The quests disguise the testing techniques. The authors report that the qualitative feedback shows that most of the students were very positive about the experience.

Rusu et al. developed a puzzle-based game for the introduction of design patterns to middle school to college junior students [118]. The game is similar to *Lemmings*. In the game robots have to find their way like the lemmings in the original game. The player performs specific actions that resemble a software design pattern, for example: a robot with binoculars is watching a robot that signals that is safe to cross a bridge, the observer pattern. A user study indicated, besides a new appreciation of software engineers, that the students learned about the concepts that were thought. A questionnaire was used where specific scenarios were presented and the students had to choose between metaphorical roles of the robot, such as observer, mediator, etc. The students were not familiar with the topics before and did not specifically learned about the terminology.

In the field of software engineering research can be found about games for learning to program (GBL approach). Most of these games have children as their target audience. Less research is found about games for software design for students.

Lee et al. present their game *Gidget* [74] [75]. Gidget is a game for learning to program. In this game learners learn to program by debugging code that has failures. The failures are made by a robot character that has a damaged chip. By solving puzzles the learner is confronted step-by-step with programming concepts. In addition Lee et al. found that a personification of the feedback (Gidget the robot) had a positive influence on the players' motivation.

Although environments like Alice, Greenfoot or Scratch are not game environments, they do break with more conventional programming environments. In stead of sorting a list of numbers or generating prime numbers they try to connect with the learners interests, such as games, stories and simulation [152]. While Greenfoot and Scratch provide the user a scene that can be filled with game objects from a repository, Alice even provides a 3D game engine.

In the field of architecture modelling Groenewegen et al. [41] introduced a game for validating architecture models. Groenewegen et al. used a board game where the actual models play a role. We also aimed to create a video game where the models itself are game elements.

BlueJ is not a game, but a Java IDE especially designed for an introduction to programming [68]. Later the authors developed Greenfoot. What makes BlueJ worthwhile to mention here is that it is an IDE that gives the UML design a central role. A programmer starts by making a visual design, a class diagram.

By providing the students an environment in which they practice modelling while playing the game, we support a 'learning by doing' approach [33]. This approach is already widely applied for learning students to program [156, 114, 15, 128].

Tenzer et al. present their tool GUIDE that supports the exploration of a software design by playing a game. [149]. This tool supports the process of a multiplayer game in which so called Verifiers and Refuters compete to proof the correctness of a design. By the means of the game a software design can be improved and flaws can be eliminated. The game should help to cope with the difficulty of not being able to apply formal verification techniques on incomplete and/or informal UML models. The tool, and thus the game, aims at designers that are already (basically) skilled in software design. It does not aim at learning in particular. This in comparison to our game, which is meant for novice learners.

Fernandez-Rayes et al. implemented the gamification of an achievement-driven advanced software design course [37]. During the course students compete with each other by attacking and stealing points with the help of virtual cards, represented as text and values in a spreadsheet. They found (partial) evidence that students have higher grades for the course when they are motivated for playing the game. Although game based learning is applied in the field of software engineering, we were at the moment of writing our publication not aware that such a video game as *AoSD* in the field of software design exists. Also, a more recent literature study does not report a similar game [126].

3.3 Method

The aim of this study is to explore if a game supports students in learning software design. The game we have built is created using an iterative approach. The authors came together periodically and discussed and improved their different ideas until there was a first version.

The learning objectives of the game are chosen using Bloom's taxonomy [70] combined with a set of general design principles [88] : coupling, cohesion, information hiding and modularity (discussed in 1.2.2). Bloom's taxonomy is widely used by lecturers. 'Design principles' is a typical topic in a software engineering curriculum. The learning

objectives are explained in Section 3.4.1

During development we tested the game by conducting a user test in the form of a simple version of the 'think aloud' [81] method. We asked the test users to articulate their thoughts while making steps in the game. We improved elements of the game and fixed bugs based on these sessions. Subsequently we addressed a larger group for testing the game and answering a questionnaire before (N=67) and after playing the game (N=13). We are aware that this is not a complete validation. As mentioned before there is no game to compare with.

3.4 Game Design

In this section we discuss the learning objectives, the type of the game, game levels, the role of UML, the game mechanics and the software that was used to implement the game.

3.4.1 Learning Objectives, The Aim of The Game

The main learning objective of the game is to introduce the software design principles coupling, cohesion, information hiding and modularity.

We categorised the learning objectives according to Bloom's taxonomy for the cognitive domain. Because the game is intended for giving an introduction we mostly focused on levels up to the *apply* level.

RECALL

- Students will know what the following concepts mean: classes - associations - methods - attributes - packages - coupling - cohesion - interface - agents - data flow - control flow.
- Students will know the following software design principles: low coupling - high cohesion - information hiding.

UNDERSTAND

- Students understand the relation between the different elements that form a class diagram.

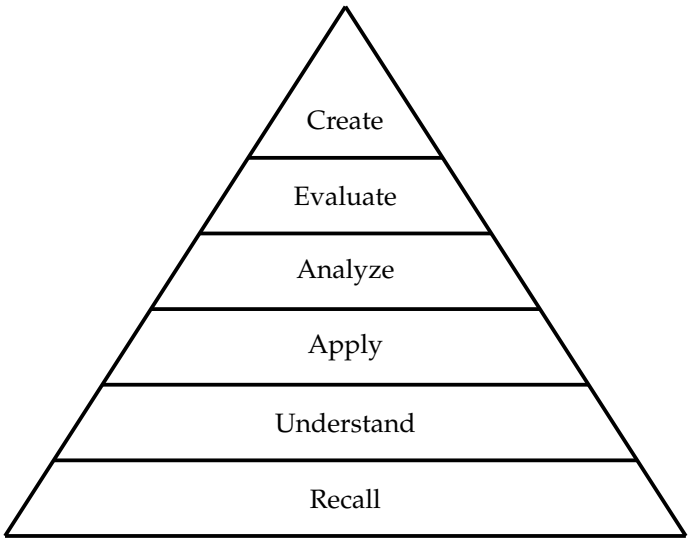

Figure 3.1: *Bloom's Taxonomy – overview of the cognitive domain levels*

- Students understand that there is a trade-off when applying the design principles coupling and cohesion.
- Students understand that there are multiple solutions for creating a software design.

APPLY

- Students can assign methods and attributes to classes in a simple design (4-6 classes) chosen from a list of suggestions.
- Students can associate classes in a simple design (4-6 classes).
- Students can apply the high cohesion principle to a simple class design (4-6 classes).
- Students can apply the high cohesion principle to a simple class design (4-6 classes) with a simple packaging (2-4 packages).
- Students can apply the low coupling principle to a simple class design (4-6 classes).
- Students can apply the low coupling principle to a simple class design (4-6 classes) with a simple packaging with the use of an interface.

3.4.2 Type of Game

We chose to make the game a puzzle game. Software design is a complex balancing problem. The designer has to solve the problem by finding the best trade-off considering different design principles that interfere. This complex task involves abstract reasoning and has multiple solutions.

We see looking for different solutions for balancing design principles as a similar task to solving a puzzle. The type of puzzle is a logic puzzle in which the students synthesise a solution that satisfies multiple logical constraints. An example of this is balancing between coupling and cohesion. In both situations the person uses his/her abstract reasoning skills.

We are aware that in reality the solution space for a software design problem may be wider than a puzzle. We think that is wise to restrict the solution space for novice software designers. Therefore a puzzle game is suitable. It provides a 'closed world' problem that is in control of the lecturer: he/she can choose the width of the solution space.

3.4.3 Game Levels

For each of the design principles, there is a set of levels that offers puzzles of increasing complexity. A puzzle offers a design fragment - typically a set of classes, attributes and methods - and asks the player to complete the design.

Each level starts with an explanation pop-up of the concept that is handled. A somewhat loose definition for the concept is given. The definition is loose because we do not want to emphasise on definitions. In the explanation pop-up also the challenge of the level is given.

The moves that a player can make differ per level. For simple levels, there are predefined classes and methods that the player can move around and connect to existing classes in the design. At more advanced levels, the player can also create new classes.

The initial levels of the game test for understanding the main concepts in isolation. Subsequent levels offer puzzles that require combined understanding of multiple design principles. Subsequent levels unlock when all linked preceding levels are finished (this is explained in Section 3.4.6).

The player receives hints when hovering over an item from the toolbox as well as when hovering over elements that are placed in the drawing area. Hints in the drawing area

Table 3.1: *The implemented levels of 'The Art of Software Design'*

Level	Description
Classes	Introduction to classes and the mechanics of the game.
Associations	Introduction to association.
Methods	Introduction to methods and their functionality.
Attributes	Introduction to attributes and their role.
Packages	Dragging classes to packages.
Coupling	Lower the coupling in a design.
Cohesion	Drag methods and attributes to classes and get an as high as possible cohesion score.
Control flow	A tree-like structure in which the player has to change the control flow in a certain way.
Data flow	Drop an attribute in the right class to let its data flow to the right class.
Package cohesion	Similar to the cohesion puzzle, but now cohesion within packages is taken into account.
Coupling v.s. Cohesion	Put 4 methods in as many or few classes to optimise the coupling and cohesion.
Interfaces	Maintain the data flow between 2 packages while only using 1 association between them.
Agents	Construct an agent between 3 classes to make a data flow.
Information hiding	Extract relevant information from a shielded data flow, without a certain class having access to the shielded data flow.

help players forward to their solutions.

Table 3.1 show the 14 puzzle levels that are created for the proof of concept of the game that was used for this research.

3.4.4 UML Notation

The game uses a notation that is close to UML. Although this notation is used for representing software designs, the game is not intended for learning the syntax of UML. We prefer to address software design principles in a more general manner. Therefore we use a notation that is a very simple subset of UML and tried to stay away from object-orientated dependent principles.

The subset we chose only consists of class diagrams because our research mainly focuses on the structural part of software design. Within the rich collection of class diagram elements we again chose for a restricted set that covers the basics of software design.

For our research we use the following UML elements:

- Package
- Class
- Attribute and Operation
- Visibility (only private and public)
- Data type
- Relationships – Dependency, (directed) Association, Aggregation, Composition and Inheritance
- Multiplicity

Examples of the UML elements that we don't use are: Interface, Realization, Abstract Class, Stereotype, Collection.

For the more high level assignments we use the condensed form of the class notation: without attributes or operations. Other assignments demand more detail in the diagram. The two different notation variations, accompanied with an example of the graphical variant that is used in the game, are illustrated in Figure 3.2.

Differences with UML For learning purposes we chose to use notation elements in additional to the UML subset. For emphasising the input- and/or output roles for method arguments we equipped these with the following prefixes:

- 'io' – a method argument that has both an input and an output role.
- 'i' – a method argument that has an input role.
- 'o' – a method argument that has a output role.

The output role can be compared with a method return result or a method argument that can be manipulated (e.g. a reference)

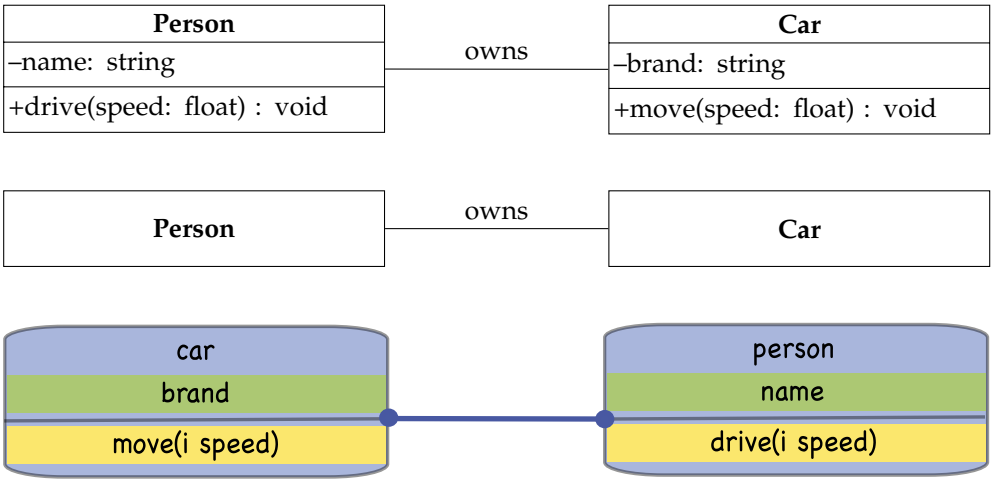

Figure 3.2: *Detailed Class Notation versus Simplified Class Notation and the AoSD version*

3.4.5 Graphical User Interface

Figure 3.3 shows the graphical user interface (GUI) of the game while playing a level. The GUI aims for the analogy of class diagram editors. The GUI consists of the following components:

- *the toolbox* – a collection of draggable items (class, attribute, and method) for making a simple class design. They are represented as crayons. Further, you can put the puzzle in a 'connect mode' with the blue line segment (4th item) to associate classes. In addition, there is an eraser to enable erase mode. By clicking items they can be erased in this mode. For resetting the level there is a curved arrow button. The last item is the exit button represented by a door icon.

  Toolbox items can be hidden by the level designer (e.g. lecturer). It is useful to hide items that are not related to a certain level's goal. In this way levels can be designed in a way that the player is offered a restricted tool set for a better focus on the learning goal.

- *the puzzle area* – a canvas that can contain the draggable items from the toolbox. Also, the game can offer partial designs (unsolved puzzles) in this area to start a level with.

- *the control and data flow area* – two circle shaped buttons for enabling the animation of control - and data flows. The animation of the data flow explains how data is transferred from one object to another object. The control flow animation explains what function is used on an object (e.g. a method call).

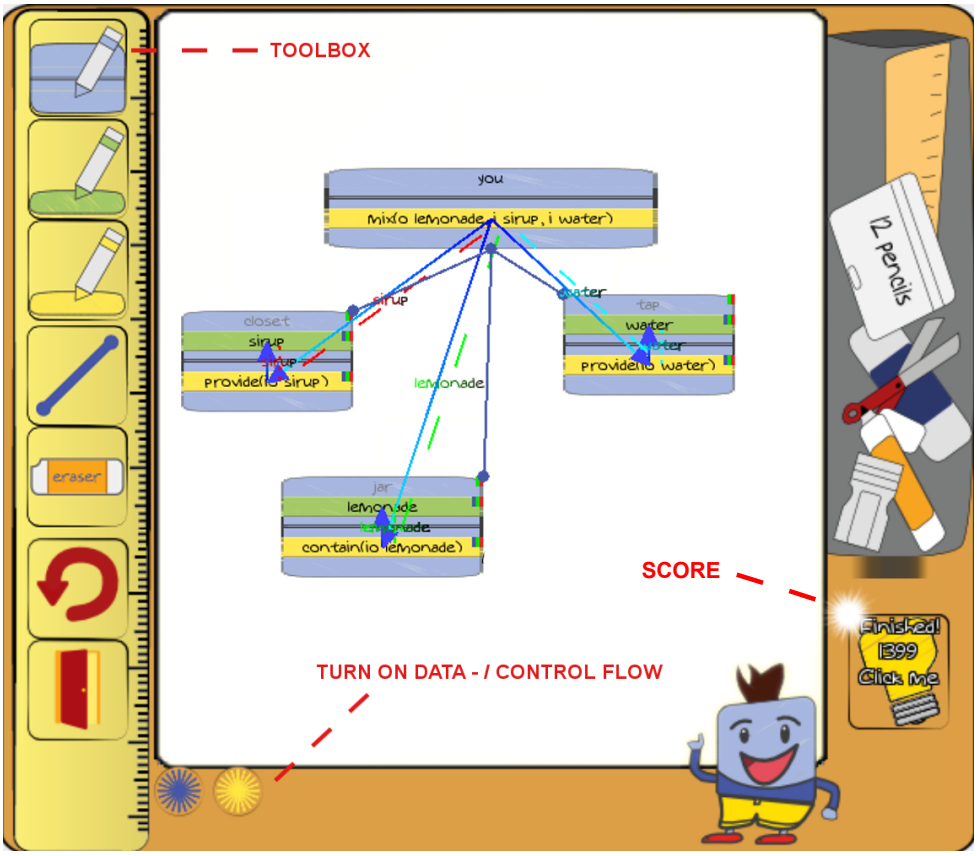

Figure 3.3: *GUI with activated data and control flows. Control flows are represented by an arrow. Data flows are dashed. GUI comments are made in red.*

- *the score indicator* – a light bulb that starts transparent. Whilst making progress the light bulb fills itself with a yellow colour. At the same time the score increases. We chose two indicators. One to give the player feedback about the process. The other one indicates the score of the level, but can vary: some solutions can be a right one, but better than the other.

3.4.6 Game Mechanics

There are several mechanics that we used in order to create an educational and engaging game. In most of the cases the mechanics mentioned below affect both educational and engagement goals.

Direct, Visual and Audio Feedback While feedback is a prerequisite for learning, the game offers feedback through two means:

- each level is scored through an evaluation mechanism supported by audio and visuals. The player receives feedback on the progress because of the intermediate evaluation scores. The player receives confirmational feedback when a solution is a right one.
- the user is given feedback through visualisation of control flow and data flow. To support a player in decision making, the data and control flow of a puzzle can be visualised. This can be seen in Figure 3.3. In this way a player can check the effect of placing an attribute or method in a certain class in comparison with the assignment of the puzzle. We think this is a valuable element of the game from the perspective of self evaluation.

When a player moves or modifies elements, feedback is provided directly and the score adjusts. In addition, sounds are played on user actions (e.g. connecting elements), starting a level and when a level is finished.

Level Unlocking To experience achievement, not all puzzles are playable from the moment the games starts. They have to be unlocked by finishing other puzzles before being able to play that particular one. In this way it is not possible to skip levels. A player needs to learn (by doing) a couple of subjects before they can work on dependent puzzles. By applying the unlock mechanism we aim to create awareness of the dependencies between the learning objectives and the concepts. An example: a player first has to finish the levels about classes and associations before the level about coupling can be played (Figure 3.4).

Choice of Path As mentioned in other research [92] [106] a user is more engaged if the freedom of choice increases. We designed the levels in a way that one can start from different starting points. Even when a player gets stuck he/she is always able to return and take another path.

Multiple Solutions Due to the context of the game, software design, the puzzles do not have one best solution. If a design respects the design principles there still can be a couple of different solutions. A design can be better than another considering the problem domain or other contextual factors. We preserved this real-life situation in the game. This is also expressed with the light bulb - score combination.

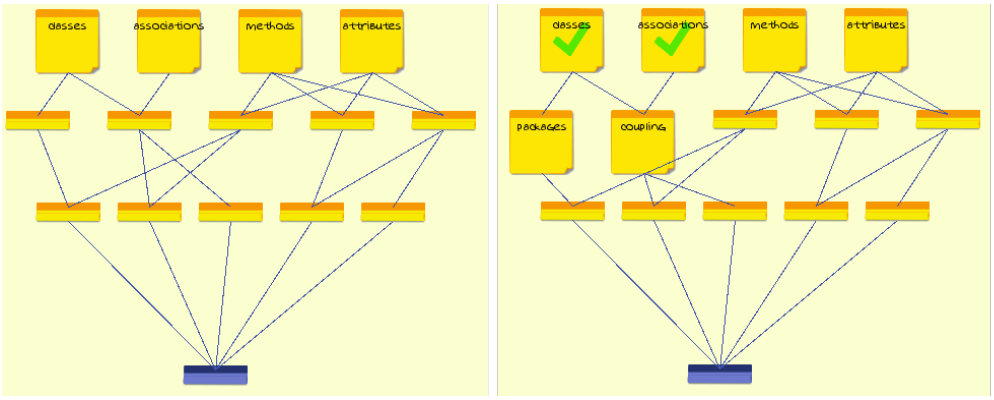

Figure 3.4: *Level unlocking – Left: no levels are unlocked. Right: levels are unlocked because classes and associations levels are completed.*

3.4.7 Scoring Metrics

The player's level score is determined by an evaluation script that scores the puzzle of the level. After every player action this script checks to what degree the created design matches possible solutions based on the design principles: cohesion, coupling, information hiding and modularity. These metrics are used in software design to determine the quality of a software design. The goal of the puzzles is similar: make the design (fragment) as good as possible (high quality). Dependent on the topic of the puzzle one ore more design principle metrics are used. This section explains the calculation of the score in more depth.

Coupling To determine coupling within a puzzle we used a simple approach. We used the CBO (coupling between object classes)[25] metric. This is a commonly used metric in software design. The CBO is a rough metric. There are more precise metrics. However, a more detailed metric would not improve the scoring system for the low to middle complex puzzles the game offers.

CBO per class is: the number of other classes it connects to

Per design the average would be the sum of all the CBO (indexed by *i*) divided by the total of classes (n):

$$averageCBO = \frac{\sum_{i=1}^{n} CBO_i}{n}$$

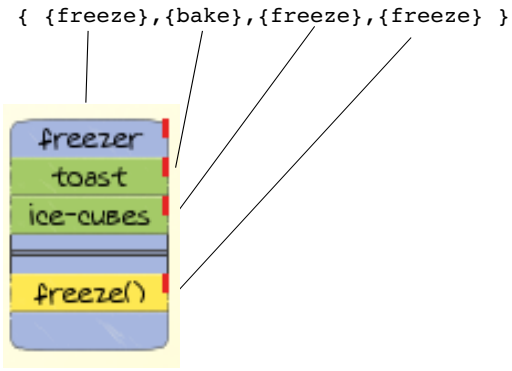

Figure 3.5: *Connected keywords to classes, attributes and methods*

Cohesion Every class, attribute and method has one or more keywords attached to it. We use these keywords to determine if two elements are related, i.e. whether they are cohesive.

The cohesion evaluation script compares if all items of a class (including the class itself) have similar keywords. In the example provided in Figure 3.5 the keyword of the class *freezer* is ***freeze***. The other elements (attributes and methods) also contain keywords: *bake*, *freeze* and *freeze*. In this example only one keyword is attached per element. This can also be two or more.

Cohesion of a class (CC) is determined as follows: i. per comparison (indexed by i), divide the number of keyword matches (m) with all the other elements by the number of keywords (k) that are being considered in the comparison. This gives us the match ratio per comparison. ii. divide the sum of these ratio's by the number of comparisons (n). This gives us the total index of the similarities of a class, the cohesion. The index is a value between 0 and 1. Where 0 means no cohesion and 1 means full cohesion. Below the calculation in formula form:

$$CC = \frac{\sum_{i=1}^{n} \frac{m_i}{k_i}}{n}$$

In the case of Figure 3.5 the CC is determined as follows:

$$CC = \frac{\frac{2}{3} + 0 + \frac{2}{3} + \frac{2}{3}}{4} = 0.5 \text{ (n=4)}$$

Information Hiding and Modularity To evaluate the application of information hiding and modularity we used general design patterns. The player has only a limited

number of choices and is guided to a solution that uses this patterns. The evaluation script checks if the user applied the elements (classes, attributes, methods) in a way that matches the pattern.

3.4.8 Software Platform

The game is constructed with Gamemaker 8.1 Standard[1]. We made this choice so we could rapidly make fully functional prototypes. Gamemaker has a readily available engine with graphics, mouse events, scripting and other features that can be used in games.

3.5 The Game

In this section we provide an overview of the flow of a game session. The main aim of the game is to complete every puzzle and get the best score.

'The Art of Software Design' [2,3] starts with an opening screen and gives the player the opportunity to personalise the game by entering the name of the player. This name is used as a saved game and by clicking it, it will resume after the last finished puzzle. After entering the welcome screen a puzzle-tree appears. This tree consists of puzzles based on software design principles or combination of those principles. Players have to unlock upper level puzzles before they can do the next puzzles.

Inside a puzzle a welcome message is shown (as e.g. Figure 3.6). This message contains the assignment of the puzzle. Typical tasks a player has to fulfil are e.g. placing attributes and operations in the right classes (responsibility driven approach), connect the right classes (associate) given a typical design principle. A toolbox is present for adding these items when needed. A progress indicator shows how close the player is to the solution of the puzzle. As mentioned before the game offers the opportunity to complete a puzzle with different solutions. One can be better than the other. This is shown by the players' score as shown in Figure 3.7.

After completing the puzzle, the player returns to the main screen where certain puzzles are unlocked. From there new challenges are possible.

[1] https://yoyogames.com/gamemaker

[2] The Art of Software Design : https://gamejolt.com/games/the-art-of-software-design/21373

[3] Trailer : https://www.youtube.com/watch?v=xn1E2dU-_zg

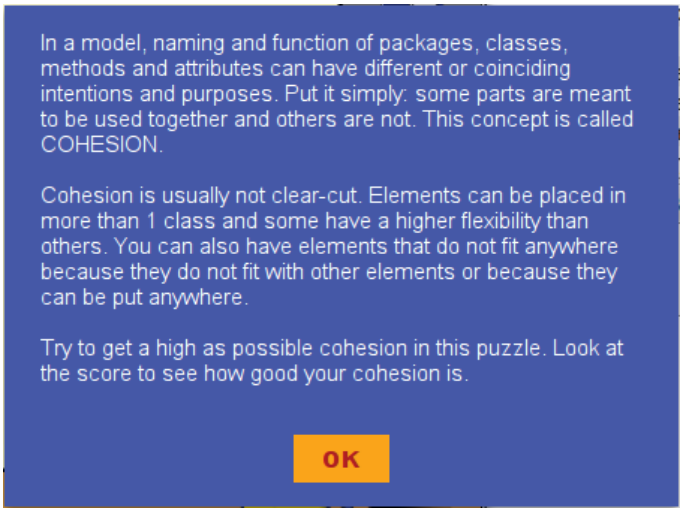

Figure 3.6: *Pop-up with assignment for the cohesion puzzle*

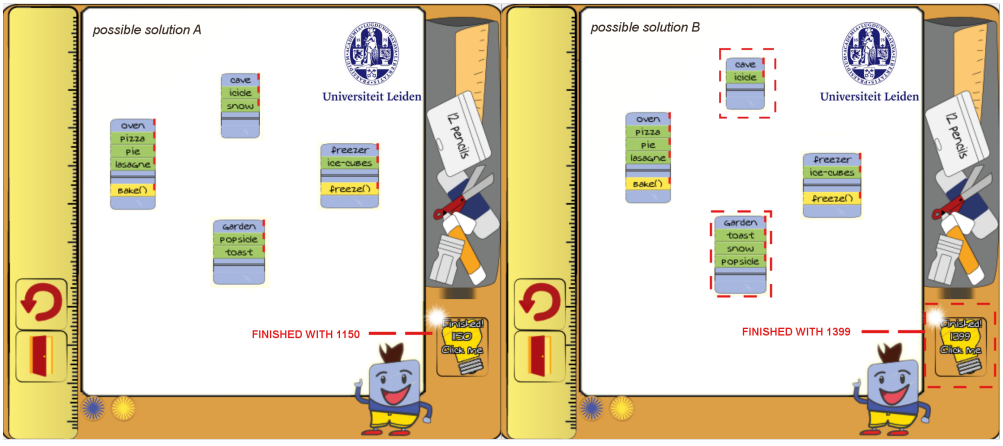

Figure 3.7: *Two possible solutions for the cohesion puzzle. The dashed red boxes show the differences.*

The game ends when all the puzzles about basic design principles have been fulfilled. After that, in future versions of the game, series of more complex puzzles can be offered.

3.6 Evaluation

We evaluated the game in two steps. First we conducted a think aloud session with 6 participants in order to get feedback for improvement. Our aim with these sessions was to: receive feedback on the accessibility of the user interface, finding bugs, checking if all text was read and for checking if the puzzles were understood. We selected 6 students with different backgrounds: from students with no software engineering experience to students who do (e.g. psychology and computer science). From first to fifth year students. Second we addressed a larger group for testing. Players with a software engineering background were approached within our own programs. General players were approached via the discussion website *reddit.com*[4] and via the gaming website *escapistmagazine.com*[5]. We recorded the answers on a survey that consisted of a pre-game and post-game questionnaire. 67 subjects responded to the pre-game questionnaire. 13 (of which 4 subjects were involved in the think aloud session) answered the post-game questions.

3.6.1 Think Aloud Evaluation

Players spent 30 (computer science student) to 90 (medical student) minutes.

Game Mechanics We noticed that giving freedom of choice was of value. When people got stuck on a puzzle, they tried out different solutions or tried another puzzle and returned to the more difficult puzzle later on.

User Interface Players paid more attention to the instructions prior to a puzzle when there were 3 paragraphs of 4 lines of text at maximum. The text was best understood if it had an introductory paragraph, an explanatory paragraph and an assigning paragraph in that order. Some puzzles seem too simple. They were solved without reading the instructions.

[4] https://www.reddit.com
[5] https://www.escapistmagazine.com

Observations For us it was satisfying to see, that after a while subjects started talking about the puzzles in terms of 'classes, methods and associations', instead of 'boxes, blocks and lines', which seems to indicate some unconscious learning.

3.6.2 Survey Group Evaluation

Players spent 35 minutes on average.

67 participants answered the pre-game questionnaire (shown in Table 3.2). Unfortunately only 13 participants answered the post-game questionnaire (shown in Table 3.3). From the 13 post-game participants 10 finished the game. We cannot explain the reason for the low amount of participants in the post-game questionnaire. We assume that participants lost interest after answering the pre-game questionnaire and did also not play the game.

Table 3.2: *Pre-Game Questionnaire Results*

N = 67	
Mean age	23 (min: 18, max: 62)
Working in IT	24
Familiar with classes	33
Familiar with methods	27
Familiar with associations	13
Familiar with coupling	10
Familiar with cohesion	7

3.7 Discussion

In this section we discuss the topics that were introduced in the introduction of this chapter: engagement, learning, interactive feedback for learning, games supporting 'learning by doing' and implementation issues.

3.7.1 Engagement

Supported by the answers on the questionnaire and player observation, we believe that a game can engage students into software design. The accessible platform invites students to start without any prior software design knowledge and gets the player

Table 3.3: *Post-Game Questionnaire Results*

N = 13			
Working in IT	4		
Finished all puzzles	10		
	yes	not	neutral
Enjoyed playing	7	2	4
Liked the graphics	8	3	2
Understood data flow	8	4	1
Understood control flow	7	4	2
Light bulb "feel I did right"	7	2	1
Scoring "feel I did right"	6	5	2
Recall of classes, methods and associations	8	2	-
Recall principles	5	5	-

engaged in learning the basic principles of software design. This positive experience should motivate them to learn about more in depth and specific knowledge.

3.7.2 Learning Yield

We acknowledge that further study is needed to demonstrate the learning effect, but we think the unconscious learning of concepts such as classes, attributes and methods at least indicates a certain learning effect. During the think aloud sessions we observed that our test players adopted the software design terminology and started reasoning about possible solutions using the appropriate terms. Further, most players that participated in the post-questionnaire mention that the understood the concepts of control- and data flow and recall classes, methods and associations. Half of them recall the software design principles.

3.7.3 Interactive Visual Feedback

Several authors emphasise the importance of the presence of feedback mechanisms in educational approaches [75] [49] and games in general [106]. We believe that the game introduces a valuable form of feedback. Next to the progress feedback that tells the player how well he/she performs, AoSD guides the students toward their solutions by supporting them in decision making. The visualisation of the control- and data flow gives students a better understanding of the relationship between the important

elements (classes).

3.7.4 Specific Implementation Issues

coupling – Determining the average coupling of the design could be a too rough measure for a part of the quality score and thus the player's score. The average coupling can turn out to be relatively low, while a certain class in the design can have a very high coupling. It may be wise to indicate the user that the coupling of a certain class is too high. In addition having a class that has a very high coupling could lead to a penalty.

cohesion – Although we find the simulation of associative thinking with keywords a very plausible solution for determining cohesion, we have no data that validates that approach. It may be wise to explore correlation between metrics such as (L)COM [46] to validate our keywords method.

3.8 Conclusion

For addressing the challenge to get students motivated for learning software design and keep them engaged, we explored an additional educational approach with the help of an educational puzzle game. In this chapter we explored if a game could support students in learning software design. Although we were not able to completely validate our results, we think that the game *The Art of Software Design* supports students in learning. We offered the players a restricted solution space in which they can be trained in the topic of design principles, the basis of the complex balancing skill a good software designer should eventually master. By practising making design decisions, the players learn about balancing between the different software design quality metrics. Players are challenged by unlocking their levels of increasing complexity by choice of path and have the freedom of solving the puzzles with multiple solutions. An important instrument in the game is the visual feedback system that guides students towards their solution. The preliminary evaluation results support our conclusion.

Current courses on the topic of software design can adopt our game as an enrichment of their didactic approaches for offering students an interactive way in order to keep them engaged. With the help of the level editor lecturers can extend the levels and introduce additional topics of different complexity levels. We see the game as an opportunity to support students' self-assessment possibilities by using the built-in feedback mechanism.

Future research and improvement of this approach. A deeper study of the validity of

the score metrics is needed. Further research is needed to demonstrate the learning effect of the game. To demonstrate the learning effect of the game we see a challenge in future research. One potential approach is to use the design skills test that was described in Chapter 2. We suggest to study both validation and learning effect in a case study that uses *The Art of Software Design*.