Internal Report 2012-10 August 2012

Universiteit Leiden
Opleiding Informatica

The Art of Software Design

Creating an Educational Game

Teaching Software Design

Oswald de Bruin

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Chapter 1

Introduction

In this paper we share our research in which we gamify the learning process of software design. We have made a game which should teach software design to first year computer science students or anyone who is generally interested. We research if it is possible to teach such a difficult subject through a game, we test if we eventually succeeded and we try to shed some light on solutions to the problems we faced.
Although software design is an essential activity in software development, software developers are not always eager to make designs[27]. Although software design is beneficial to the development process, since design does not directly produce code, it is difficult to justify spending large amounts of effort on design [4]. On a small scale, software projects usually do not need a lot of planning. Because of this, software developers do not notice the problems of sloppy designs until a later point. Better education in software design should be a solution to a lot of software projects failing due to sloppy design.
However, teaching software design and with it the object oriented program structure is not easy. One of the reasons is that students usually learn procedural programming first and then learn object oriented programming, so they have to un-learn a pattern they first thought was right. A solution would be to start teaching the object oriented from the start [12, 13]. We want to take it a level higher and start with the UML based software design.
A new problem arises when we want to teach software design or, more broadly, any subject that might be considered boring or that goes against the previously learned methods of the student: the willingness to learn. Traditional ways of learning, reading text books and attending classes, do not always hold the interest of the student and therefore do not communicate subject matter very well. A way to hold interest, to make it 'fun', is to gamify the subject [21].

The boring aspect is not the only part of disinterest for careful software design. If a deadline is tight, one has to come up with a quick solution which is probably not really thought through. One must also not forget the human factor. If someone has an apathetic personality, one will not care to make the right design. If someone is a know it all, one will not want to learn new things in fear of been proven wrong. One may want to look smart and design a very unnecessary complex system. These are all factors that influence a design, even if the right knowledge is readily available [2].
The willingness is not only influenced by the student's interest. Though it is a major factor, lack of confidence for the subject and its seeming complexity are also factors one needs to take into account, as we will explain later in this paper. By making a game, we hope to overcome these human factors and keep the player interested even if there is an initial unwillingness.
Gamifying a subject might hold the interest of the player, but we also need to take into account that the player is a student. In the process of making a subject 'fun' one must not forget the learning aspect. We will show some hurdles one encounters and show solutions one can use in the process of making an educational game, which in our case will handle software design.
In Chapter 2 we will explain underlying theories, show a curriculum we wanted to teach and compare similar projects. In Chapter 3 we will show the implementation of our game and algorithms we devised to evaluate student's actions. In Chapter 4 we discuss the tests we have done to test our theories with our implementation. In Chapter 5 we show the results of our tests and we discuss their meaning in Chapter 6. We conclude with Chapter 7. This paper is supervised by Michel Chaudron and Dave Stikkolorum. The research in this paper has been presented in its preliminary stages on June 09 2012 at the Games And Software design workshop, part of International Conference on Software Engineering 2012 in Zurich. The research has also been submitted to the Models Gamification Contest at MODELS 2012 in Innsbruck.

Chapter 2

Background

Making an educational game has some essential requirements we need to break down. Since it is a game, it needs to be fun, but also engaging. For this we delve into the psychological definition of fun in Section 2.1 and the concept of the Skinner Box 2.2. To construct a method to teach, we refer to Bloom's taxonomy in Section 2.3. The game is educational, so it needs to teach a subject in a curriculum and it needs to do so effectively. We devise a curriculum in Section 2.4. To not repeat previous research, we look at similar projects in Section 2.5 and to have an oak-point, we compiled a philosophy in Section 2.6 to check our game to.

2.1 Psychological definition of fun

For our definition of 'fun' we consider the field of (neuro) psychology. The experience of what can be considered fun lies in the level of specific neurotransmitters in the nerve system. A nerve in the human nerve system has two ways of communicating: a fast, short lived way through pulses and a slow, but more sustaining way by emitting specific substances called neurotransmitters. The neurotransmitters bring a person in a certain state that s/he experiences as fun. This counts for any emotion for that matter. Because neurotransmitters are a substance, the experience of fun can last until after the actions that led to it [10, 17, 20].
This definition alone does not give enough information on how to achieve 'fun' in a game. However, it does help us to understand 'fun' and the structure in which it is experienced, a rush with lasting effects. This gives us a means of structuring our game. For our game this means that when the player has done something s/he finds fun, we should give him/her the opportunity to 'wallow'

in one's fun or to experience the situation that has led to it again. However, we need more theory to structure effectively.

2.2 The Skinner Box

Officially, the Skinner box is called the 'operand conditioning chamber', but is by most named after its creator, Burrhus Frederic Skinner. In the Skinner box are a button, a speaker, a device to dispense food and a floor that can give an electric shock. In the box, an animal is taught by using positive and negative stimuli to push the lever if and only if the speaker makes a sound. If the lever is pushed at the right time, the animal receives food, if it is pushed at the wrong time, the animal gets shocked. By rewarding preferred behaviour and punishing unpreferred behaviour the animal eventually only shows the preferred behaviour, which means it is pushing the lever at the right time [3]. The Skinner box gives us a means of keeping the player engaged, or otherwise on playing the game. This structure is seen, among others, in video games and is considered very effective [6]. However, the Skinner box does not necessarily mean learning, but is more about teaching a subject a trick through something that engages him/her. It gives no guarantee that topics to be learned will stick to the subject's memory. The action itself does not engage the subject, rather the reward does. Even if the topic sticks, the player might see through the reward system and abandon the game. We have to use the effect of the Skinner box sparingly, probably only to spark initial engagement.

2.3 Bloom's taxonomy

To structure the learning aspect of our game, we look at Bloom's taxonomy. It shows a way of dividing educational objectives in 3 domains: The cognitive, the affective and psycho-motor domain [5]. We can use these domains to define the learning objectives of the game and its game play.
The cognitive domain addresses the knowledge skills that are needed for learning. At the first level a person should only be able to recall facts of studied materials.
In the cognitive domain a person obtains information about a problem. S/he comes to understand the problem and can state the problem and possibly its solution in his/her own words. With this understanding the person should be able to apply the knowledge in new situations. The person should be able

to analyze components in a problem and to synthesize a new structure from previous knowledge that is applicable to new situations[1].
The affective domain focuses on the feelings of a person. First, a problem must grasp the person's attention and s/he must be willing to respond to it. When involved in the problem, the person must be able to value aspects of the problem and organize them by priority. At the highest level the person should be able to predict the behaviour of a problem and should be confident on solving it [15].
The psycho-motor domain focuses on a person's development of skills. A person should be willing in solving the problems. They can anticipate a problem and solve it with more ease than before. At the lowest level someone should be able to imitate a solution and adapt routines. This is also known as guided response. A person gradually becomes able to solve more complex problems [26].
There are multiple revisions on Bloom's taxonomy, but we want to focus on this first interpretation, because it defines elements of learning and gives requirements to make an environment in which the elements are possible. Other revisions are simplified where this version defines different sources for a person to learn from.

2.4 Software engineering curriculum

For our game we focused our curriculum on the subjects we describe in this section. We want to make a distinction between concepts and patterns. Where concepts are descriptive of the quality of the model, patterns are good ways to create a model.

coupling Coupling in software design concerns the connectedness between classes in a class diagram. The lower the coupling, the better the design is considered [23, 7, 16].

cohesion The term 'cohesion' in software design concerns the way 'properties' in a certain class share a common aspect. The more similarities are shared, the higher cohesion is considered. High cohesion is preferred [23, 7, 16].

control flow For control flow we consider the way methods or functions call eachother with a model. If function A calls function B, which in turn calls C, then there is a control flow from A to C. [23]

data flow We consider data flow to be the path in which data travels when it is read or written by an operation. Data flow is derived from the control flow, since control flow largely inclines I/O operations. [23]

modularity We consider modularity to be the ease with which a model can be changed [23, 7, 16]. This is a too vague and complicated concept to teach head-on, so we want to teach it by teaching how to construct modular patterns with low coupling and high cohesion.

With these concepts we explain the following patterns we want to teach in our game:

interfaces In order to keep coupling between packages low and cohesion in a model high, one can choose to let one class manage a group of classes. The class that manages the group or package is called an 'interface' [25, 8, 24].

agents When data needs to be read from one place and written to another, it is not wise to put the writing to or reading from a class in the other class. This would lower cohesion. In order to solve this problem, a separate class should be made that reads the data from one class and writes it to the other. This is called an agent.[25, 8, 24]

information hiding When handling classified data, a user should only be able to see the information one is entitled to. If all data is structured to be stored together, regardless of the entitlement of the user, the right data needs to be extracted, so the wrong data is hidden from the user. This concept is called 'information hiding' [25, 8, 24].

We want to focus mostly on teaching the right patterns, instead of showing a bad pattern and telling what is wrong with it. We will discuss this more broadly in Chapter 3. There are ofcourse more patterns and concepts than stated above. We kept our curriculum at this list, because most of the other subjects can be derived from coupling and cohesion and can be explained with control and data flow.

2.5 Similar projects

To look at similar projects, we first want to establish how a game is usually structured. A majority of games are designed with the player controlling an avatar who has to reach a certain goal, usually a place where the avatar has to

hit a button or something similar [28, 19], think of 'Super Mario' (Nintendo, 1985) or 'Rogue' (Michael Toy and Glenn Wichman, 1980). The other majority are puzzle games, which can be mostly described as an environment which is directly changed by the player [19], think of one of the many card game or mah-jong implementations. Our game belongs in the latter category, although we do not just implement cards.
Ofcourse, we are not the first to make an educational game. Games like 'Oregon Trail' (Minnesota Educational Computer Consortium, 1975) and 'Gazillionaire' (Lavamind, 1995-2012) focus on teaching through game-play. In this category we also have games that are not meant to be educational, but are based on a real environment, which makes the player interested in its subject. An example of this is 'Age of Empires' (Microsoft, 1997). Other games, like 'Number munchers' (Minnesota Educational Computer Consortium, 1988) and the 'Magic School bus' series (KnowWonder, 1997-2001), focus on educational content livened up by adding a game to it. Our game tends to fall in this category. Where we stand out from other educational games is that we focus on a more mature audience.
One can also gamify an already existing environment. One that has proven great for teaching is BlueJ. It focuses on teaching object oriented java programming [11, 14]. Although it is a good tool that makes the student produce usable code, we want to simplify our environment to focus more on concepts and patterns. Actual code might distract from the overall picture we want to show the player.

2.6 Main philosophy

As other research shows [18, 22], a person is more engaged when he or she has a freedom of choice. Combining this with what we learned about human behaviour on fun, reward and learning, we put forth our philosophy, which consist of 3 elements: Tolerance, variation and deviation. These elements need to be balanced in order for the game to be effectively fun, engaging and teaching. Let us break down the what and why of each element in relation to educational games:

Tolerance This element concerns the principle of the Skinner box and the cognitive domain of Bloom's taxonomy and considers the width of error ranges. According to the Skinner box principle, a subject starts to show preferred behaviour when the right behaviour is rewarded and the wrong behaviour is punished. To combine this with the cognitive domain of

Bloom's taxonomy, we reward the player when s/he seems to have learned something from the curriculum and punish him/her when s/he does not.

Problems arise when the player tries to understand a subject, but can't grasp the right notion. In this situation the player gets constantly punished, because his/her actions are wrong. If the punishment is too consistent, the player starts to associate the punishment with playing the game instead of the wrong actions. Keep in mind that a player does not have to play the game. Obligation goes against the fun principle of a game. Too harsh punishment will make the player abandon the game and the remainder of the curriculum will not be taught to the player.

Introducing tolerance for wrong actions should keep the player playing when s/he tries to learn a difficult subject, keeping him/her from abandoning the game from frustration. Being tolerant solves the problem, but being fully tolerant gives rise to a new problem: the player skims through the curriculum without learning. The reward aspect of the Skinner box principle keeps the player engaged in playing, but the attention of the player is directed at the reward, not the curriculum. This diminishes the cognitive domain of Bloom's taxonomy.

In order to keep the player playing and learning, there should be not too much punishment and not too much reward. These concepts are barely quantifiable, let alone the balance between them. A rule of thumb might be to not make the rewards and punishment instantly follow up the actions, but the right balance should be found by play-testing the game.

Variation This element concerns all 3 domains in Bloom's taxonomy and the neuro psychological definition of 'fun'. To keep the player engaged, the actions and concepts to be learned need to be varied. When looking at Bloom's taxonomy, the player should learn, like and improve a skill. For a skill to be properly taught, we need to spend a lot of focus on the psycho-motor domain to let the player practice. A player needs to dwell on a certain subject to get it to be memorized.

Concerning the neuropsychological definition of fun, the emotion is experienced with a peak that diminishes over time. If too much time is spent in the psycho-motor domain, if actions are to often repeated, the emotional line diminishes below a certain threshold to which the player is no longer engaged. This is devastating to the affective domain of Bloom's taxonomy. The player no longer pays attention to his/her actions. The elements in the game no longer have a value to the player.

To keep the player paying attention we need to balance introduction and repetition of actions, effectively. Too little variation gets the player bored, too much variation gets the player confused on the subjects we are trying to teach. Again, with this element of our philosophy we need to find the balance with play-testing using the rule of thumb that one should not design only variation or dwell on just one subject.

Deviation This concept entails variation in focus on problems and is a small extension on the previous 2 points of our philosophy. If game progression is linearly structured and a player gets stuck on a problem, engagement is lowered, because the punishment is too frequent. The player then eventually stops playing the game, because playing it gives negative associations. In this situation the player should have the option to try a different approach or to let him/her solve a different problem entirely. This way the player can reflect on his/her previous actions and compare them to the new situation. Too narrow a focus lowers engagement on the difficult parts. Too wide a focus lets the player evade important subjects s/he should learn and devalues the perceived importance of a taught ability. Again, with play-testing the right balance should be found.

This philosophy was our oak-point to the requirements (Section 3.1), game mechanics (Section 3.2.1) and testing (Chapter 4) of our game.

Chapter 3

Implementation

In this section we show the implementation of our educational game. In Section 3.1 we set apart the requirements we've set for the game. In Section 3.2 we describe the final software product we created. In Section 3.3 we show and explain algorithms we devised to evaluate subjects of the curriculum described in Section 2.4.

3.1 Game requirements

To effectively teach our curriculum we compiled the following functional requirements for our game:

1. The player must be able to build and edit a UML-like model.
   (a) The models and problems will be presented in the form of a puzzle.
   (b) The player's resulting model in a puzzle will be evaluated on coupling, cohesion, control flow and data flow
2. The player progresses through different puzzles in a directed acyclic graph with multiple entry points (deviation).
   (a) The player can progress to a puzzle by completing the previous puzzle in the graph.
   (b) A puzzle can be completed by meeting previously stated requirements.
   (c) Besides the requirements to meet, the player will receive a score based on coupling, cohesion and overall structure.

(d) Player's progress will be saved.

3. The game must have possibilities to define coupling, cohesion, control flow and data flow of individual elements in the model.
   (a) These definitions should not be editable by the player.
   (b) A teacher must be able to make his/her own puzzle to fit to their curriculum, using the same tools as us.

In addition we compiled the following non-functional requirements:

- There should be more than one solution to most puzzles (tolerance).
- Instructions to puzzles must be clear.

3.2 Software

We constructed our game in Gamemaker 8.1 Standard. 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 could be used in games. This way we did not have to worry about the engine, if we wanted to add small features, like a simple sound.

3.2.1 Game mechanics

For the final software we designed 3 screens: the starting screen, the DAG (Directed Acyclic Graph) screen and the model screen. The starting screen (Figure 3.1) shows a classroom with tables which gives the player to log in to an account that saves his/her progress. Each table depicts an account. The DAG (Figure 3.2)is shown after a player logs in. In this screen the player can chose an available puzzle to play. Note that not all puzzles are available from the start, as described in Section 3.1.
The model screen (Figure 3.3)is the main screen of focus in our game. This is where the player is confronted with a problem s/he needs to solve. When entering the model screen from the DAG screen the player first gets a pop-up message stating the problem and partly explaining how it should be solved. When the player OK's the pop-up s/he has access to the playfield.
Figure 3.3 shows a screenshot of our final interface, with on the left from top to bottom buttons for creating classes, attributes, methods, associations, deleting parts, restarting the puzzle and exiting the puzzle field. In the middle is the

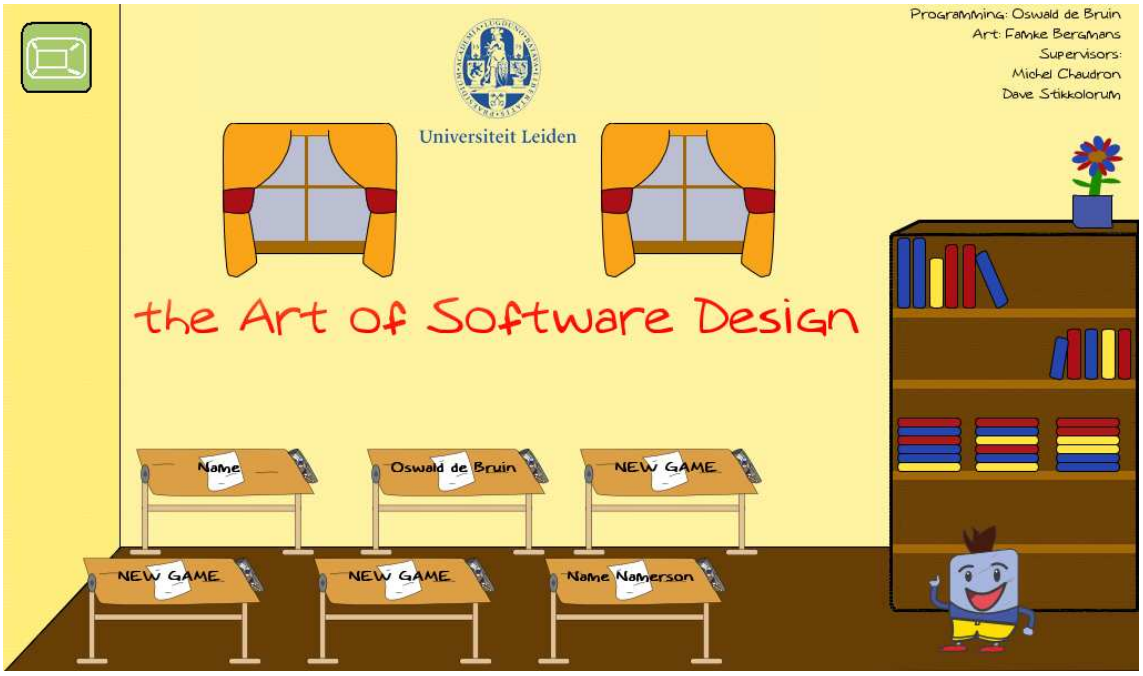

Figure 3.1: Screenshot of the starting screen. Individual graphics by Famke Bergmans

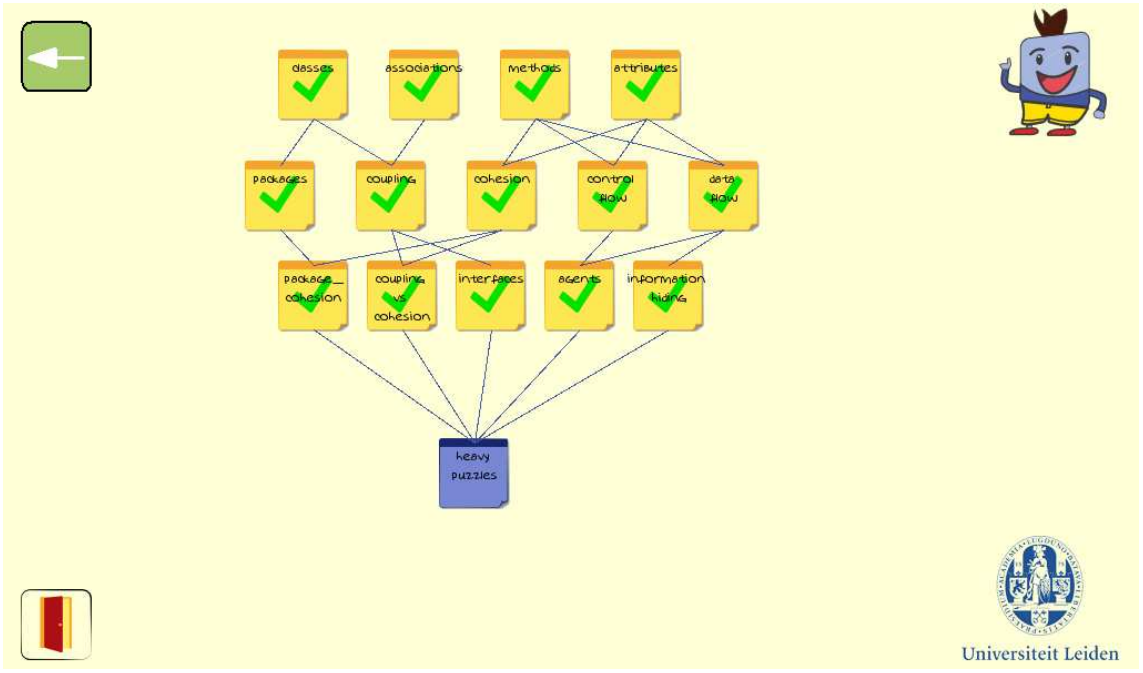

Figure 3.2: Screenshot of the DAG screen. Individual graphics by Famke Bergmans

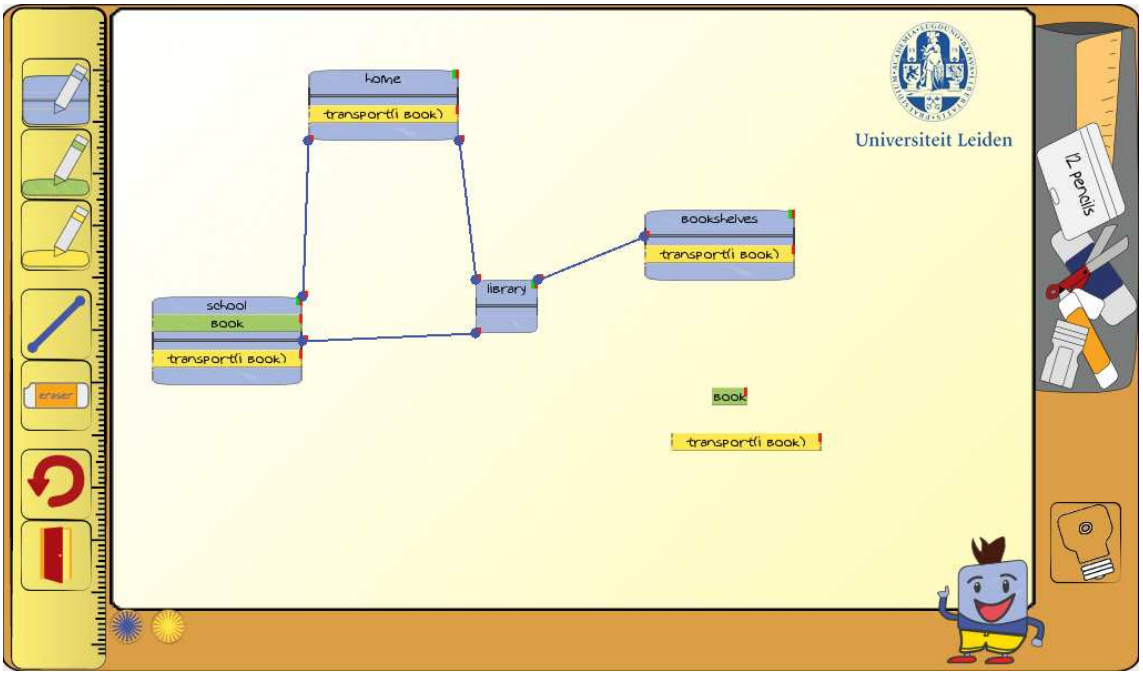

Figure 3.3: Screenshot of our final interface. Also screen with example puzzle 'classes.' Individual graphics by Famke Bergmans

playfield with a partial solved puzzle with a couple of unassigned methods and attributes. Left below the playfield are two buttons to visualize control and data flow. On the bottom right is the mascot who can give hints and repeat the instructions and a button to confirm the solution the player created. This confirm button has a number depicting the score and an image of a light bulb on it. The light bulb fills up as more of the puzzle is solved.

data structures

The data structures are visualized in the meta-model in Figure 3.4. The player can use packages, classes, methods, attributes and associations. All can be edited by the player, except the packages. Packages are bounded areas in the field in which classes can be dropped. Packages can hold more than one class and classes can only be assigned to one package. Methods, attributes and associations can also be dropped in packages, but there is no functionality assigned to this event, unless they are assigned to a class.
Classes can hold more than one method, attribute and/or association. An

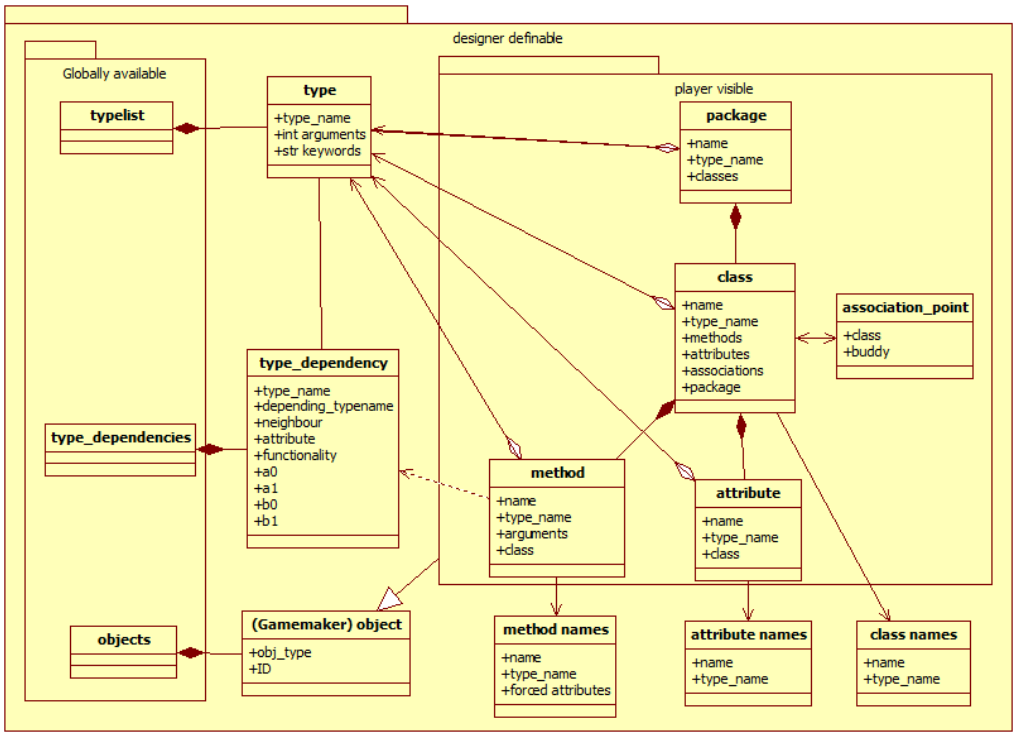

Figure 3.4: Meta model of the game's elements.

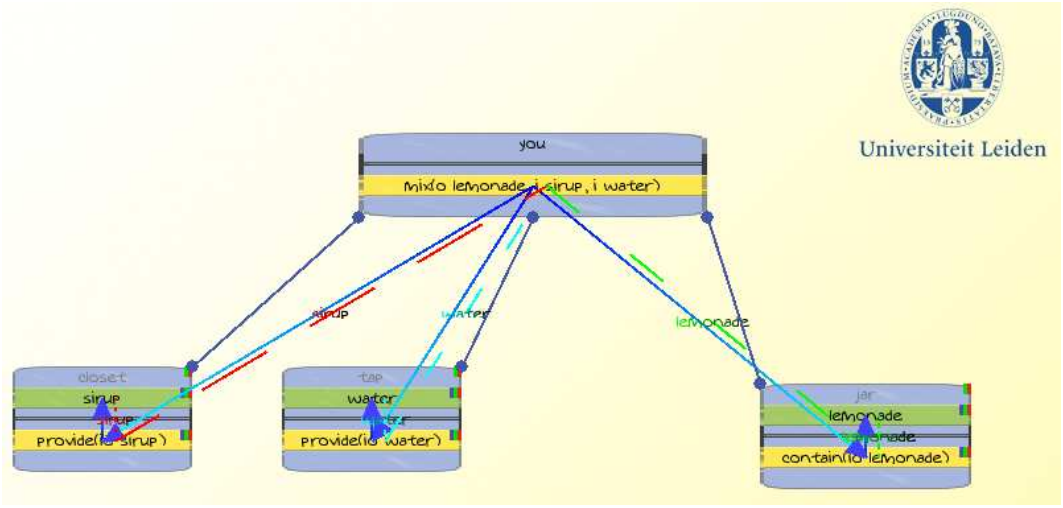

Figure 3.5: A zoom in on a model showing control flow (blue arrows) and data flow (coloured dotted lines). Individual graphics by Famke Bergmans

attribute can be assigned to one class at a time and depicts a variable within its class. Methods are assigned like attributes, with the difference that they depict functionality. With this functionality methods can construct control flow and data flow between classes. Methods hold more than one argument which has a name that can coincide with one or more attributes.
An association is constructed between 2 classes to make control flow and data flow possible between these classes. For control flow and data flow within a class no association is needed within the class. For implementation purposes the association consists of 2 objects, "association points", which individually have to be assigned to one class and one class only. Both points have to be assigned to a different class.
Functionality is defined by 'types' and 'type dependencies', which are invisible to the player, but visible to the creator of a puzzle. A type is a data structure holding a type name, cohesion keywords and an argument-integer. The name is used to assign types to packages, classes, methods and/or attributes. The cohesion keywords are used for calculating cohesion, explained in Section 3.3. The argument-integer is only functional for methods and defines how many arguments a method can have.
A type dependency can be assigned to a type and holds a type name, a "attribute-boolean" depicting dependency on a method (0) or an attribute (1), a "neighbour-boolean" depicting dependency on an object in its own class

(0) or a neighbouring class(1) and 4 integers (a0,a1,b0 and b1) depicting the arguments compatibility range (explained in Section 3.3). The type name depicts the type to which the dependency is assigned. The dependency also holds a functionality-integer, which denotes if a method is reading (0), writing (1) or just touching without initiating a data flow (2). Visualization of control and data flow is shown in Figure 3.5.

3.2.2 Constructing scripts and puzzle editor

The construction, its visible elements and types, and evaluation of a puzzle is done in scripts in a subdirectory of the game directory. These scripts hold scripting commands from Gamemaker that are directly executed when loaded. The construction and evaluation are divided over two scripts: the build script and the evaluate script. The build script is run once at startup of a puzzle. The commands in the build script construct all elements described in the previous section.
The evaluate script is run at every step when playing, which is an average 30 times per second. This script holds a different set of commands that check if the puzzle corresponds to a preferred structure and it calculates a score for the model's coupling and cohesion. The commands in the evaluate script make use of the algorithms depicted in Section 3.3.
To make scripting easier, we included a 'level skeleton editor'. This becomes available when a puzzle is set to 'debug mode'. This can be done by including a debug command in the construction script. When a puzzle is in debug mode, the user has the possibility to generate construction code that constructs the puzzle as it is rendered at that moment on the playfield. We created a special button for this that becomes visible during debug mode. With this button, the creator can easily create an initial structure for a puzzle. We also give the creator the possibility to construct packages, which is not possible for a player. Also for this action we included a button that becomes visible in debug mode. To make debugging even more easier, we included a snapshot function and debug info at the bottom of the screen, see Figure 3.6.
When the initial structure for a puzzle is made, the creator can add functionality and validation in the build and evaluation scripts. For a full list of the commands used in the scripts, please refer to the documentation included with the game, available at aosd.host22.com.

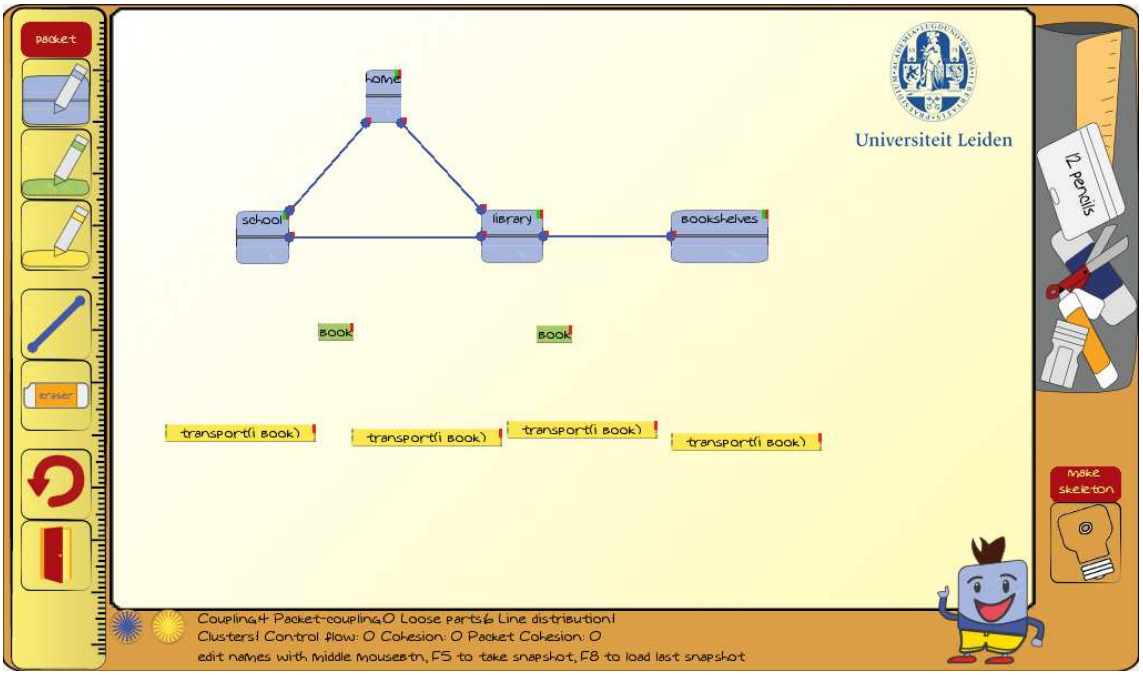

Figure 3.6: The class puzzle in debug mode with the package button top-left, the skeleton button near the validate button and debug information at the bottom. Individual graphics by Famke Bergmans

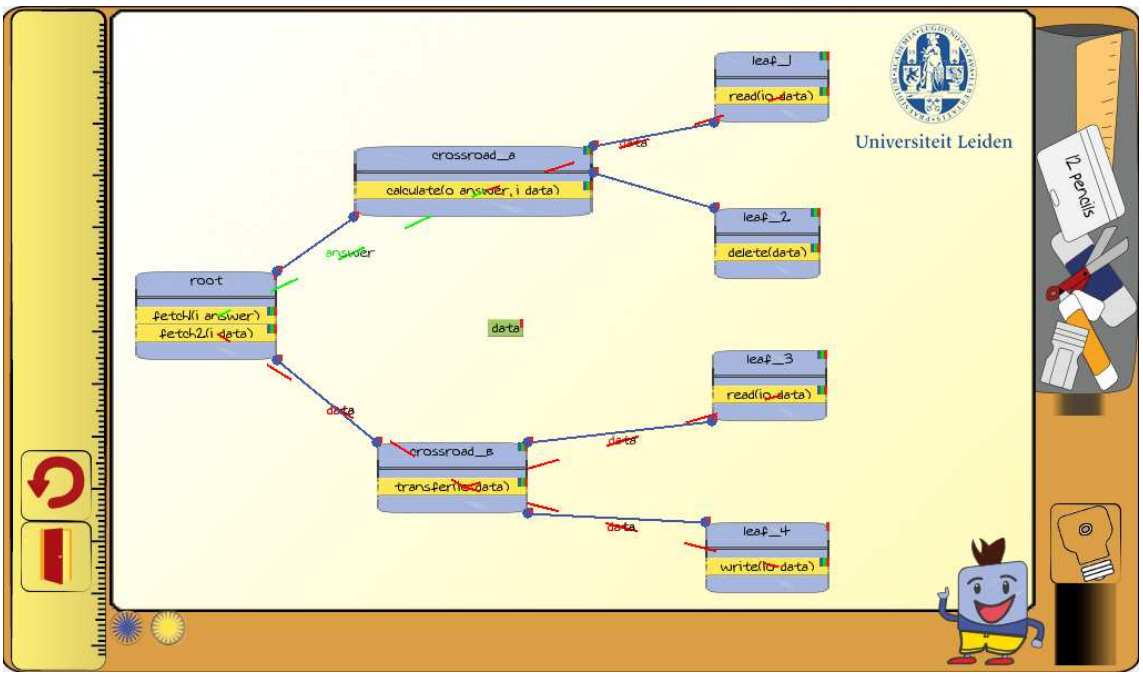

Figure 3.7: Screenshot of the data flow puzzle with data flow view enabled.

3.2.3 Puzzles

Eventually we created 14 puzzles for our game, these are:

Classes Introduction to classes and the mechanics of the game. Example in Figure 3.3

Associations Introduction to associations

Methods Introduction to methods and their functionality

Attributes Introduction to attributes and their role.

Packages A simple puzzle dragging classes to packages

Coupling A simple puzzle where the player must lower the coupling in a puzzle

Cohesion A simple puzzle in which the player drags methods and attributes to classes and has to get an as high as possible cohesion score.

Control flow A puzzle with a tree-like structure in which the player has to change the control flow in a certain way.

Data flow A puzzle in which the player needs to drop an attribute in the right class to let its data flow to the right class. Example in Figure 3.7.

Package cohesion Same as the cohesion puzzle, but now we take the cohesion within packages into account.

Coupling vs Cohesion A puzzle where the player needs to put 4 methods in as many or few classes s/he likes to optimize the coupling and cohesion.

Interfaces A puzzle where the player has to maintain data flow between 2 packages while only using 1 association between them. Example in Figure 3.8.

Agents A puzzle where the player has to construct an agent between 3 classes to make a data flow.

Information hiding A puzzle in which the player needs to extract relevant information from a shielded data flow, without a certain class having access to the shielded data flow.

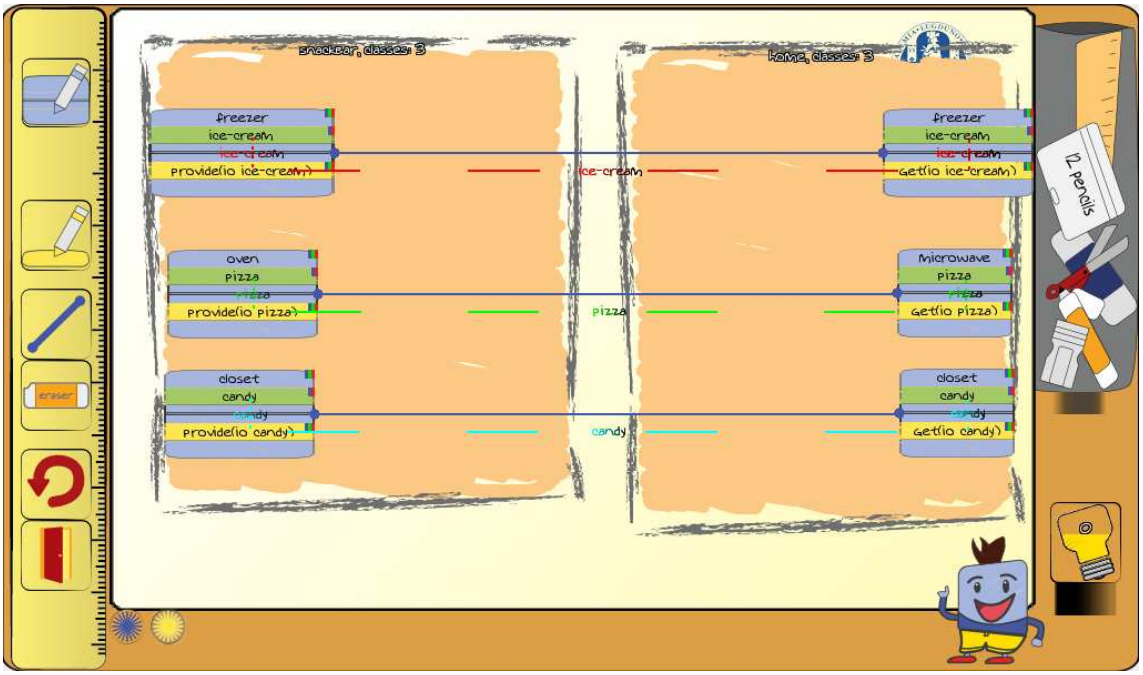

Figure 3.8: Screenshot of the interfaces puzzle with data flow view enabled.

3.3 Algorithms

In this section we explain the algorithms we devised to measure coupling and cohesion and to construct control and data flow. These algorithms were used to check the structure of the model that was constructed by the player. With each of these algorithms, take in mind that the variable 'objects' is a global variable containing an array with all the Gamemaker objects on screen. Although using global variables is not a sign of good software design practice, we had to stick to the implementation of the Gamemaker tool.

3.3.1 Coupling

The algorithms for coupling returns an integer higher or equal to 0. The algorithm is very simple, since it only counts the associations present in the model. In the implementation of the game the associations were constructed out of 2 points which both had its own class and the other point referenced as a 'buddy'.
The coupling between classes is obtained by counting all association points and

dividing that number by 2, see Algorithm 1. Since the game mechanics (Section 3.2) make sure no association connects a class to itself and no association between the same two classes can occur twice, coupling is depicted by this value and no extra checks need to be made.

Algorithm 1 eval_coupling()

```
1: coupling = 0
2: for i= 0 → length(objects) do
3:    if objects_i.obj_type = association_point then
4:       coupling ← coupling +1
5:    end if
6: end for
7: return coupling/2
```

The algorithm for evaluating coupling between packages is a bit more complex. Since classes in packages can have associations to classes within the same package, the algorithm needs a bit of revision.
The package cohesion algorithm returns a value of the same kind, an integer higher or equal to 0. For every package the algorithm checks all association points of all its classes. If one such association point has a buddy who's class is not in the same package, the coupling value is raised by one, see Algorithm 2.
With the implementation of this algorithm, an association going from a class in one package to a class in another package gives a coupling value of 2. This is because both association points have a buddy (the other) in a different package. If an association goes from a class in a package to a class that is not in a package, the association will give a value of 1.

3.3.2 Cohesion

Our cohesion evaluation is an algorithm that can be run on class-level and package level. The cohesion algorithm returns a real value between 0 and 1, depicting the cohesion over the whole model.
Our algorithms for cohesion makes use of the keywords associated with each type, which are associated to an object in the model, being a package, class, attribute or method. The types of each object are stored in a list, as seen in Algorithm 3 and 4, and on this list the cohesion calculation is performed, Algorithm 5. Algorithm 3 and 4 both calculate the average cohesion on a class and package level, respectively.

Algorithm 2 eval_package_coupling()

```
1: package_coupling = 0
2: for i= 0 → length(objects) do
3:    if objects_i.obj_type = class then
4:       for j= 0 → length(objects_i.associations) do
5:          if objects_i.package ≠ objects_i.associations_j.buddy.class.package
            then
6:             package_coupling ← package_coupling +1
7:          end if
8:       end for
9:    end if
10: end for
11: return package_coupling
```

The calc_cohesion algorithm makes use of a "grade", which is directly influenced by the amount of comparisons done in the given typelist and the amount of keywords per type (line 10). This grade has a real value between 0 and 1. If the keyword of one type is present in the keyword of the other, cohesion is raised by the grade. This check is done both ways (line 12 to 21 in Algorithm 5).
The more keywords in two types, the lower the grade will be. This way objects with more keywords will have a lower impact on the cohesion per keyword. The amount of comparisons also influences the grade, but the comparisons-variable is the same during the whole execution, so the eventual cohesion returned by the calc_cohesion algorithm is at maximum exactly 1.
We are aware of the LCOM method of calculating cohesion [9], but we do not fully agree with it. Where LCOM purely takes the amount of objects and their functionality into account, we wanted a metric that took the *intent* of objects into account. Ours is a more feeling-based approach whose outcome can be influenced by changing some keywords when we want a puzzle to have a slightly different outcome. LCOM is a very concrete metric and if, by chance, it has an outcome against the point we want to teach with a puzzle, we need to change the whole puzzle. The choice for our metric was made mostly for modularity purposes, benefiting game development.

Algorithm 3 eval_cohesion()

```
1: cohesion ← 0
2: amount ← 0
3: for i= 0 → length(objects) do
4:    if objects_i.obj_type = class  then
5:       amount ← amount +1
6:       temp_typelist ← {objects_i.type_name}
7:       for j = 0 → objects_i.attribute_amount do
8:          temp_typelist ← temp_typelist ∪{ objects_i.attributes_j.type_name }
9:       end for
10:      for j = 0 → length(objects_i.methods) do
11:         temp_typelist ← temp_typelist ∪{ objects_i.methods_j.type_name }
12:      end for
13:      cohesion ← cohesion + calc_cohesion(temp_typelist)
14:   end if
15: end for
16: return  cohesion / amount
```

Algorithm 4 eval_package_cohesion()

```
1: cohesion ← 0
2: amount ← 0
3: for i= 0 → length(objects) do
4:    if objects_i.obj_type = package  then
5:       amount ← amount +1
6:       temp_typelist ← {object_i.type_name}
7:       for j = 0 → length(objects_i.classes) do
8:          for k = 0 → length(object_i.classes_j.attributes) do
9:             temp_typelist ← temp_typelist ∪{
               objects_i.classes_j.attributes_k.type_name }
10:         end for
11:         for k = 0 → object_i.class_j.method_amount do
12:            temp_typelist ← temp_typelist ∪{
               object_i.classes_j.methods_k.type_name }
13:         end for
14:      end for
15:      cohesion ← cohesion + calc_cohesion(temp_typelist)
16:   end if
17: end for
18: return  cohesion / amount
```

Algorithm 5 calc_cohesion(temp_typelist)

```
1: N ← length(temp_typelist)
2: checks ← 1/2N(N−1)
3: cohesion ← 0
4: for i= 0 → length(temp_typelist)−1 do
5:    for j=i+1 → length(temp_typelist) do
6:       obj_keywords_i ← get_type_keywords(temp_typelist_i)
7:       obj_keywords_j ← get_type_keywords(temp_typelist_j)
8:       grade ← ( length(obj_keywords_i) + length(obj_keywords_j) )× checks
9:       if grade ≠ 0 then
10:         grade ← 1/ grade
11:      end if
12:      for k = 0 → length(obj_keywords_i) do
13:         if is_in_set(obj_keywords_i_k,obj_keywords_j) then
14:            cohesion ← cohesion + grade
15:         end if
16:      end for
17:      for k = 0 → length(obj_keywords_j) do
18:         if is_in_set(obj_keywords_j_k,obj_keywords_i) then
19:            cohesion ← cohesion + grade
20:         end if
21:      end for
22:   end for
23: end for
24: return cohesion
```

3.3.3 Control flow

The control flow algorithm returns a directed graph, which is implemented as a set filled with sets containing a starting point, an ending point and a functionality-variable in that order. The starting and ending point are both a method or an attribute in the model. The functionality-variable is defined by the type-dependency described above and depicts if the control flow holds a reading (0), writing (1) or deleting (2) action. The algorithm is split into two algorithms: the higher level eval_control_flow, which searches for methods and their dependencies, and the lower level search_class, which searches a given class and edits the control flow graph.
Eval_control_flow searches all classes on the playfield for functional methods. Loose methods on the field are ignored, since a method needs assignment to a class to be considered functional. When it has found a method, it searches the type-dependencies list for that method's type. When a dependency is found, the algorithm only checks the neighbour variable of that dependency. If the dependency is needing an object in the current class, the search_class algorithm is run on the current class. If the dependency is needing an object from a neighbouring class, all associations are searched for connected classes and the search_class algorithm is run on those.
If the dependency needs a method, search_class searches all methods of a class, if it needs an attribute, it searches all attributes. In order for a method to be a needed dependency to the control flow, its arguments need to coincide with the requesting method. Which arguments coincide is defined by a the range depicted by a0, a1, b0 and b1 of the type-dependency. The amount of arguments that need to coincide is defined by the shortest range (threshold on line 5). For example: If a0= 0, a1= 3, b0= 0 and b1= 2 then only 2 argument from the first 3 arguments of the source needs to have the same name as the first 2 arguments of the destination. The arguments do not have to be in the same order. If the dependency needs an attribute, only the name of the attribute needs to coincide with the name of an argument in the source in the range a0 $\rightarrow$ a1.
If the destination is found, the branch from source to destination is added to the control flow graph by concatenating the existing control flow set of sets with a set containing one set with the source, destination, and the functionality of the dependency. After this, the algorithm starts looking for another destination in the class, until there are no more objects left to examine in the class.

Algorithm 6 eval_control_flow()

```
1: Cflow ← ∅
2: for i= 0 → length(objects) do
3:   if objects_i.type = class then
4:     for j= 0 → length(objects_i.methods) do
5:       for k= 0 → length(type_dependencies) do
6:         if objects_i.methods_j.type_name = type_dependencies_k.type_name
           then
7:           if type_dependencies_k.neighbour = 0 then
8:             search_class(Cflow, objects_i.methods_j,type_dependencies_k,
               objects_i)
9:           else
10:            for m = 0 → length(objects_i.associations) do
11:              search_class(Cflow, objects_i.methods_j,
                 type_dependencies_k, objects_i.associations_m.buddy.class)
12:            end for
13:          end if
14:        end if
15:      end for
16:    end for
17:  end if
18: end for
19: return Cflow
```

Algorithm 7 search_class(Cflow, method, dependency, class)

```
1: if dependency.attribute = 0 then
2:   for i= 0 → length(class.methods) do
3:     if class.methods_i ≠ method and class.methods_i.typename = dependency.depending_typename then
4:       threshold =min(dependency.a1−dependency.a0, dependency.b1−dependency.b0)
5:       count = 0
6:       for j=dependency.a0→ dependency.a1 do
7:         for k=dependency.b0 → dependency.b1 do
8:           if method.arguments_j = class.methods_i.arguments_k then
9:             count ← count +1
10:            break
11:          end if
12:        end for
13:      end for
14:      if count ≥ threshold then
15:        Cflow ← Cflow ∪{{method,class.method_i,dependency.functionality}}
16:      end if
17:    end if
18:  end for
19: else
20:  for i= 0 → length(class.attributes) do
21:    for j= 0 → length(method.arguments) do
22:      if method.arguments_j = class.attributes_i.name then
23:        Cflow ← Cflow ∪{method, class.attributes_i, dependency.functionality}
24:      end if
25:    end for
26:  end for
27: end if
28: return Cflow
```

3.3.4 Data flow

The _eval_data_flow algorithm (Algorithm 8) constructs a directed graph from the control flow graph, that depicts the data flow. It uses a slightly different representation than the control flow algorithm. Instead of branches, the set of sets returned by the data flow algorithm depict 'stars' with 1 source, depicted by the first element of the set within the set, and an unlimited amount of destinations, depicted by every other element after the first in the set within the set. By using this structure, the algorithm became less complex than when using separate branches.
The algorithm starts with building a set containing a set with the object from which we want to examine the data flow. This data flow has a name, being an attribute's name or a method's argument's name. From this object, it checks the control flow set of sets if it is a source and with writing functionality (line 6) or if it is a destination which is read by another object (line 9). If so, the combination of the current object and the other object is handled by the add_to_dflow algorithm, see Algorithm 9.
In the add_to_dflow algorithm the destination object is being checked for having the requested data of the data flow. If the data name is present in the destination, the algorithm can begin editing the data flow set of sets. Note that it is not checked if the data name actually is a part of the dependency constructing the control flow. This check was left out, because (1) if there is control flow from A to B and they share the same data, one can assume the data flows next to that control line and (2) performing that check would make the algorithm much more complex.
When the destination is considered part of the data flow, the algorithm performs 3 operations on the set of sets for data flow. First it adds the destination object to the current set, then it checks if the destination is a first element in one of the sets within the set. If so, a new set within the set is made with that destination as root.
When the add_to_dflow algorithm has run, the eval_data_flow algorithm keeps checking the whole control flow set until all of the branches have been examined as a possible data flow branch. After every iteration, if there were new data flow destinations, the add_to_dflow algorithm has added new sets containing one element to the set. These new sets are then also checked for their possible data flow. This repeats until no more new unchecked sets remain within the set.

Algorithm 8 eval_data_flow(object, data_name, Cflow)

```
1: Dflow ← {{object}}
2: i ← 0
3: while i < length(Dflow) do
4:    own ← Dflow_{i,0}
5:    for j = 0 → length(Cflow) do
6:       if Cflow_{j,2} = 0 and own = Cflow_{j,0} then
7:          add_to_dflow(object, data_name, i, j, 0, Dflow, Cflow)
8:       end if
9:       if Cflow_{j,2} = 1 and own = Cflow_{j,1} then
10:         add_to_dflow(object, data_name, i, j, 1, Dflow, Cflow)
11:      end if
12:   end for
13: end while
```

Algorithm 9 add_to_dflow(object, data_name, i, j, x ,Dflow, Cflow)

```
1: destination ← Cflow_{jx}
2: found ← false
3: if destination.obj_type = method then
4:    if has_argument( destination, data_name) then
5:       found ← true
6:    end if
7: else
8:    if destination.name = data_name then
9:       found ← true
10:   end if
11: end if
12: if found= true then
13:   Dflow_i ← Dflow_i ∪ {destination}
14:   if is_root(destination) = false then
15:      Dflow ← Dflow ∪{{destination}}
16:   end if
17: end if
18: return Dflow
```

Chapter 4

Tests

To see if our implementation met our requirements, we made two tests. The first is a speak out loud test in which we let people of our target audience play the game and we focus mostly on bug fixes. The second is a survey to a broader audience in which we mostly focus on the enjoyment and the teaching of the game.

4.1 Speak out loud

The speak out loud test was done mostly to prepare the game for the survey. We focused on the accessibility of the interface and on finding bugs. We wanted to know if the instructions were read, if all the buttons in the interface were clear and people understood the goal of every puzzle. In between tests we did some quick fixes on the game if they were crucial for gameplay and possible to fix in a couple of minutes.
The player was to play the game without any help from us, the spectator. Nothing was exactly timed or recorded, since the use of a stopwatch or video-camera would make the player nervous and impair enjoyment of the game. Whilst playing, the player was to narrate his/her own actions, so the spectator could document crucial actions and bugs in the game. The player could ask questions to the spectator, but unless it was about a bug in the game, they were met with "I'm not allowed to help you with that." Afterwards the players were asked about their experience and what they thought about some of the design patterns and elements.

4.2 Survey

After the speak out loud tests we 'polished' the game and sent it out with a survey. The survey consisted of 2 questionnaires and playing the game. First the player was asked to take the 'pre-game' questionnaire in which we tried to verify the player's previous knowledge. Then the player was asked to play the game for around an hour and after that we asked the player to take the 'post-game' questionnaire to verify if his/her knowledge increased after playing the game.
The questionnaires were made with Google docs and the game plus the links to the questionnaires were hosted on our site, http://aosd.host22.com. We let the last 4 speak out loud subjects take the subject to test its deployment. After that we added some questions we deemed necessary and spread the survey. To test educated players, we spread the survey under first year computer sciences students of the LIACS. To test uninformed players, we spread the survey on sub-domains of the 'internet metropolis' www.reddit.com and the gaming website www.escapistmagazine.com.

Chapter 5

Results

In this section we show the results of the previously described tests.

5.1 Speak out loud

We performed the speak out loud test on 6 people divided over 4 sessions (2 individuals and 2 pairs), with knowledge of software design ranging from nothing to fifth year computer sciences students. There was not necessarily a difference in time for completing the game if the player knew more about software design. In one of the paired sessions the person with more knowledge finished the game later and was eventually helped by the less knowing person. However, comparing our individual session with a computer sciences student and a session with an art-history student, the art-history student did not finish the game after 90 minutes of play, while the computer sciences student finished it in 30 minutes.
The tree structure of the progress of puzzles gave players the possibility to deviate from difficult puzzles, but not all of them used this possibility. If a puzzle was too hard for a player, some abandoned the puzzle to try another one, but others were stubborn and did not quit until the puzzle was solved. Usually the latter kind of player took longer to finish the game.
This does not mean the longer playing group understood less of the subject. Notably, people who understood the subject better took longer to play the game, but also had more trouble solving puzzles. We will discuss this problem later in Section 6.
The instructions at the beginning of the puzzles were often, but not always, skipped. The longer the instructions, the more likely they would be skipped. Usually when a instruction was longer than 3 paragraphs, it was not read.

This does not mean that short instructions did better. At the first tests the cohesion puzzle only had the instruction "put the methods in the right classes." This was easy to follow and was one of the fastest solved puzzles. However, the players got stuck at the follow up puzzle, since they did not learn what cohesion was.
The cohesion puzzle gave us more points for consideration. It was easily solvable and it had a mechanic in it that showed what cohesion could be, but it did not stick in the player's memory. It was too easy and gave the player no incentive to think about the subject.
Other puzzles that were easily solvable and had longer instructions had the same problem. A puzzle needs to give the player a phase of consideration and validation of what is learnt in the instructions. The easier the puzzle, the less the player learnt from it, but that does not mean a puzzle should be extremely hard.
Difficult puzzles with open solutions, like the information hiding puzzle, also did not effectively teach the player. In a difficult puzzle, the player is constantly met with negative feedback, so the player can not effectively evaluate what was told in the instructions. What worked most effectively was a puzzle with not too long instructions, 3 paragraphs if around 4 lines, and a puzzle that gave the player thought for consideration, but had a clear goal. The best example from the game are the interfaces puzzle and the agents puzzle.
In Figure 5.1 we visualized the reactions brought up in our speak out loud session. We categorized the reactions under our main focal points: fun, education and bug reports. These categories were broadened by their positive and negative side, making 6 groups of categorized reactions: fun, not fun, educational, not educational, features and bugs. We made sub categories for the educational and bug-report categories. The educational categories are divided in application, engagement and practice, corresponding to the cognitive, affective and psycho motor domain of Bloom's taxonomy. The bug reports were divided in categories for errors, interfaces anomalies and missing parts in puzzles. The fun categories are barely divided, since the related reactions did not fit in specific categories and were already small in number.
The most reactions fall under bug reports, since these were the most easily identifiable. Positive bug-reports (features that appear to be working) were much smaller in number, because one does not expect features to be broken. Positive and negative educational reactions were evenly represented. It's disappointing that there were more negative than positive engagement reactions, but this is countered by the larger amount of positive fun reactions.

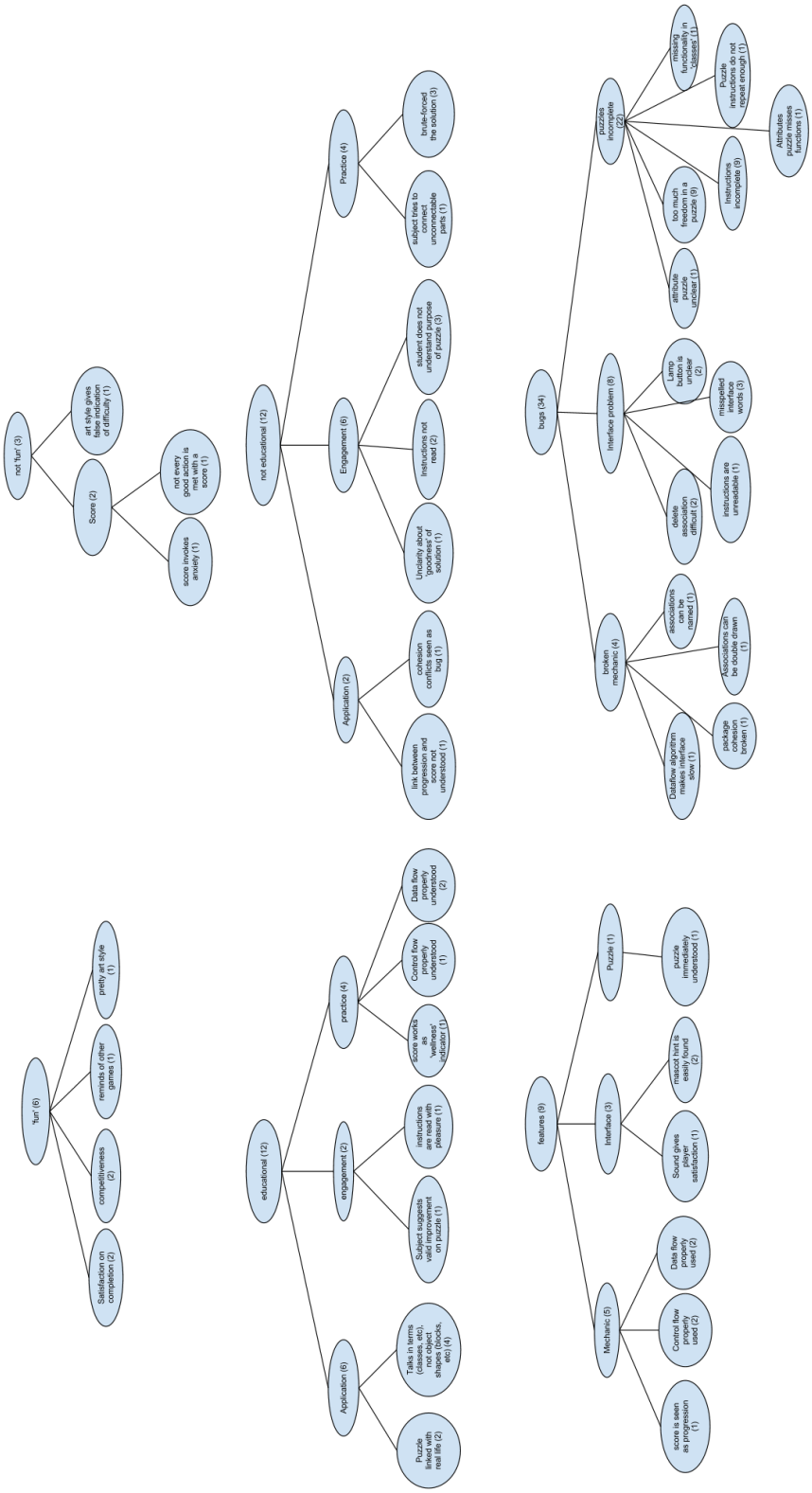

Figure 5.1: Visualization of all reactions brought up during the speak out loud session

5.2 Questionnaire

The survey was not the success we had hoped for. A lot of people did the pre-game survey, but almost all of them skipped the crucial post-game survey. From the 67 people who took the pre-game survey, only 13 took the post-game survey, 4 of which were from the speak out loud sessions when the survey was in its preliminary stages. The surveys of these people are incomplete, since we added more questions later, but we had to take these into the data pool to have a bit more significant data.
The average age of people playing the game (according to the pre-game survey) is 23 years old, with participants ranging from 18 to 62 years of age. They played the game on average for 35 minutes. From the pre-game survey 24 out 67 people were in IT, in the post-game survey 4 out of 13. 10 out of 13 on the post-game survey finished all puzzles.
7 People agreed on that they enjoyed playing the game versus 2 who did not and 4 who were neutral. The graphical design did just as good, as 8 liked it, 3 did not and 2 were neutral. Asking if they felt like they were in school, 9 disagreed, 1 was neutral and 3 agreed. The visualization of data flow made 8 people understand the concept, but there were still 4 who thought the visualization did not contribute and 1 who was neutral on the subject. Control flow did about the same, 7 agreed, 2 neutral 4 disagreed in its contribution. There is a general distinction between people in IT and those who are not, as the first group is more positive in general about the game. In the suggestion box IT people mostly state positive aspects whereas those who are not tend to note things they dislike about the game.
The light bulb icon in the bottom right proved to be a valuable part of feedback. Most (7) agreed that the sound with its filling light-bulb made them "feel I did right", 1 was neutral and 2 disagreed. The score indication in this light-bulb was less fortunate, since 5 out of 13 disagreed on its "feel I did right" factor and 2 were neutral. This last notion might be due to the fact that we did not implement the score number as careful as we wanted to. The score was only a true indication in puzzles considering coupling and cohesion and only became 1000 on completion at the other puzzles.
The questions about what people thought they knew and learnt were added later, so only 10 people got to answer those post-game. However, we got enough data from the pre-game survey and since roughly the same amount of IT and non-IT did the pre- and post-game survey we can still take percentages as an indicator.
The concepts people recalled best after playing the game were classes, associ-

ations and methods, all scoring 8 participants that recalled versus 2 who did not. Classes and methods were relatively well known pre-game, 33 do versus 31 not and 27 do versus 37 not recalling respectively, but associations were very obscure with 13 known versus 54 not. This being well known after the game is not a surprise, since classes, methods and associations are the most often used elements in our puzzles.
Patterns and pattern concepts were not so well recalled after playing the game. Information hiding, coupling and cohesion were all met with barely half of the post game participants recalling what they were. Coupling and cohesion were not a complete disaster, since they were known by only a few pre-game (10 vs. 57 and 7 vs 57), but interfaces were claimed to be known by 29 of all participants. This part, however had a different notion of interfaces, as they often claimed keyboards and mice to be interfaces, which is true in computer interaction, but in software design we mean something slightly different.
Despite it being a pattern, the best learned concept was 'agents', which was claimed to be not known by 58 out of 64 before playing the game and known by 6 out of 10 after playing the game. Checking those who explained in their own words learned us that they really understood the subject. There was also one who claimed not to know what agents were, but still managed to summarize its function in his/her own words: "Transport data."

Chapter 6

Discussion

Creating a game about software design was not easy, since we had to reinvent a core principle in game-design: the goal depicting success. The goal never changes within a regular game, but in our game the goal was to create a certain pattern, which was not a regular goal for a game, and the type of pattern also changed per puzzle. This was hard to implement modularly, because every pattern needed a different script and the difference in goals also confused the player. This is not to say we failed in making a clear goal for the player. If the instructions were short (3 paragraphs max), the player was mostly willing to learn about a new goal.
There was also a big meta problem involved when creating this specific educational game about software design. It is very difficult to discuss parts of the game without explicitly stating on what level one is discussing. For example: the word "class" could imply the sprite seen on the screen, the data structure within the program's code, but also an object within the software design for the code of this game. All levels of development, being the GUI, the code and the design were all referred to by the same definitions.
We found some arguments against educational gaming. The first came from our speak out loud sessions. The way people play a game is by trial and error, with the emphasis on getting to the goal as fast as possible. The purpose of education is to teach the 'why' and 'how' of a subject. These interests do not necessarily coincide. As we've seen in the speak out loud tests, when players know nothing about the subject, they try to find the solution by brute force, test every viable combination. When the player does know something about the subject we find an opposite situation, as we had with the 'coupling vs cohesion' puzzle. If the player is taught wrong, s/he will take longer solving a puzzle than someone who did not pay attention, because the former wrongly

avoids possible solutions.
We also found out that not everything is teachable solely through games. One can learn a skill, but if one needs to learn definitions, the designer needs to include instructions with the game. Since instructions are text-reading, which is part of the traditional way of teaching, it is impossible to teach everything through pure gaming. However, the right balance of instructions and gameplay make the player more willing to read an otherwise uninteresting topic. Adding gameplay to the instructions gives the player a means to test an otherwise intangible concept. A video game might not be a replacement to traditional teaching, but it is a great addition to teaching as a whole.
The third argument in favour of traditional teaching is our findings on engagement from our survey. Gamification is said to enhance engagement [21], which is beneficial for the affective domain in Bloom's taxonomy. We found from our survey that people who did not partake in the speak out loud session and who were not in IT did not really care about the game and its subject. Note that the speak out loud sessions were done with acquaintances, who already had a small interest in our work, and that people from IT should already have an interest in the subject. This was not the case with the non-IT strangers. We even had to remove some post-game survey results who were filled with profanities and obviously not accurately filled in. Although this is an unpleasant way of gathering data, it is proof that a game, even when carefully crafted, does not necessarily stimulate the affective domain or improve engagement.
To evaluate our game we need to look at it in 2 ways: as a game with the content it currently has and as a tool for teachers to expand and use in their curriculum. As a game it was mostly successful. Players learned concepts and patterns with great ease in a relatively short amount of time. The best part is that they understood the reason for the use of the concepts and patterns, even if they had no previous knowledge or were particularly interested in the subject.
The game failed in some aspects to catch the interest of players who were unwilling to begin with. If a player does not want to learn, s/he will not read instructions and randomly try to find a solution, effectively not learning anything.
The game also needed a lot more play testing if we wanted to eliminate puzzles that did not effectively teach certain topics to some players. We could not fully anticipate what was learned to whom, since this was different per player. A topic can stick very well with one player while another can not grasp what it is about.
As we established in Section 2.5, there is no one way to make an educational

game. There is also a difference in interest per player. Different people like to play different games and some methods of gamification will work better on certain people than others. As a result, our game failed to grasp the attention of some people, while other people were very positive about it. This is a problem probably any educational game will face.
Lastly, as a game we slightly overlooked a requisite of the psycho-motor domain in Bloom's taxonomy: repetition. It should come as no surprise that the patterns we tried to teach did not stick with half of the players. Most patterns were handled only once and not mentioned afterwards. Since classes, methods and associations played a major role in every puzzle, these were better recalled.
As a tool, the game suffered from the short spiral in the development. We hoped that creators would be able to make a puzzle-model for the game and that the game as a tool would teach software design to the player without the creator needing much thought of it. This is unfortunately not the case. As it is now, there are a lot of useless functions that can be used by creators and they have too much freedom to bypass the teaching principles we established in this paper. Creators can make instructions longer than 3 paragraphs. They can (and probably will) make puzzles with very narrow solutions. It is also possible to teach anti-patterns.
In order to effectively create a good puzzle in this game, the creator needs to know the content of this paper and read the first chapter of the instruction manual with the game. This is a lot to expect from a user, if one just wants to go ahead and make a puzzle. The creator as a user was not the first priority during design, so this is not a big issue. Even though a creator can render the game useless as a tool when uninformed, if a creator is well informed and does enough play-testing on potential students, the game can be greatly and effectively expanded.

Chapter 7

Conclusion and future work

One of the citations used in the introduction of this paper from Marc Prensky ([21]) claimed that with gamification learning would finally become fun and effective. After our research we beg to differ. When done right, a game can indeed be fun and effective at teaching, but there are some hurdles that one needs to take into account, most notably: development.
Just as any piece of software, (especially) an educational game needs to be carefully checked if its functionalities meet the requirements. Things go wrong when the requirements are incomplete, the development was sloppy or the content was not carefully presented in the user interface. These are all viable threats to development [2] and if one of these threats is not handled right, the game will not teach effectively and Prensky's statement becomes invalid. This brings some irony to our paper: We started out making an educational game about software design and now we show why software design can be a bottle neck for educational games.
Another problem that works to the traditional education's advantage is development time and cost. An educational game might relieve students from study-efforts, but it is very taxing on effort for the one making the game. When going into specialized topics learnt by few people, the effort put in making the game will most likely not weigh up to the effort saved by the students. To illustrate: the educational part of our game has been explained in a couple of pages in Chapter 2 of this paper. It might have taken a day to research and write. The game we've developed touches the same topics, but took the same author 6 months to fully develop.
The disappointing part of this conclusion is that both types of education, traditional and gamified, need a teacher to check the student if s/he has effectively learnt the topic. A game might save the teacher time, but if the student is

not willing to learn (Bloom's taxonomy's affective domain) the student will try (and most likely succeed) to cheat the game. Again we emphasize: finishing the game does not mean the player has learned something. We must see gamification as an addition to traditional education, not a replacement.
With that in mind, let's take a look at our game. The game we have developed for this paper is effective, but incomplete. First of all it needs a lot more puzzles for teaching more patterns and to repeat the execution of those old and new patterns to enhance the psycho-motor domain. The puzzles that are included now mostly teach concepts and only a few patterns. This needs to be expanded with more patterns and 'real life' situations in which a player should be able to choose a pattern.
In order for the game to get more puzzles, the interface for the creator or teacher needs to be altered or changed entirely. The new interface should refrain the creator of teaching the topic in a wrong way (see Chapter 6). This might mean the game needs to be completely remade, but not completely redesigned. The algorithms can and should all be reused along with the hierarchy of model elements. It is mostly the interface that needs to improve on accessibility for student (player) and teacher (creator).
A final improvement on the game might be customizable written feedback. Right now we give feedback in the form of a score and a light-bulb. This is not as good as actually telling the player what is wrong or right with the puzzle s/he is solving. Because the goal is different per puzzle and because the creator should be able to create his/her own goals, it was difficult to come up with a way to actually explain to the user how a puzzle should be solved aside from the introductory instructions. One must also note that if a written feedback is implemented, it is not guaranteed that the player will read it, since the instructions were also often not read.
Our game was very effective for our target audience, but to say our game is proof that one can teach anything through a video game is a bit overconfident. The sample size of our survey was not nearly big enough to make any such bold claims. We might want to do a separate research using a bigger survey. Furthermore, software design is just one topic of all existing difficult topics. However, the tests on the game clearly gave indication of effectively teaching skills needed for software design. It taught these skills in a relatively short time (around half an hour) and it made the player engaged in the topic, which gives way to effective supplementary teachings of a more traditional kind. If anything, our game is a great supplement to a software design curriculum.

Bibliography

[1] Benjamin S. Bloom. *Taxonomy of Educational Objectives Book 1: Cognitive Domain*. David McKay Co inc., 1956.

[2] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns; Refactoring Software, Architectures and Projects in Crisis*. Robert Ipsen, 1998.

[3] R. Carlson. *Psychology-the science of behavior*. Pearson Education, 4th edition, 2009.

[4] Xiwen Cheng, Wouter Eekhout, Martin Iliev, Frank van Smeden, Chengcheng Li, Oswald de Bruin, and Jaron Viëtor. Empirical study on influences of modeling during requirements and maintenance phases.

[5] Donald Clark. Bloom's taxonomy of learning domains: The three types of learning, october 2001. http://www.nwlink.com/~donclark/hrd/bloom.html [online; Accessed 15-jan-2012].

[6] Dennis Coon. *Psychology: A modular approach to mind and behavior*. Thomson Wadsworth, 2005.

[7] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[9] R Harrison, S J Counsell, and R V Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[10] William A. Kahn. Psychological conditions of personal engagement and disengagement at work. *The Acedemy of Management Journal*, 33-4:692–724, 1990.

[11] Michael Kölling. *The design of an object-oriented environment and language for teaching*. PhD thesis, University of Sydney, 1999.

[12] Michael Kölling. The problem of teaching object-oriented programming part i: Languages. *Journal of Object-Oriented Programming*, 11(8):8–15, 1999.

[13] Michael Kölling. The problem of teaching object-oriented programming part ii: Environments. *Journal of Object-Oriented Programming*, 11(9):6–12, 1999.

[14] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):1–12, 2003.

[15] D. R. Krathwohl, B. S. Bloom, and B. B. Masia. *Taxonomy of Educational Objectives, the Classification of Educational Goals. Handbook II: Affective Domain*. David McKay Co., Inc., 1973.

[16] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, 2001.

[17] John Liggins, Robert O. Pihl, Chawki Benkelfat, and Marco Leyton. The dopamine augmenter l-dopa does not affect positive mood in healthy human volunteers, January 2012.

[18] Robert Moser. A fantasy adventure game as a learning environment. why learning to program is so difficult and what can be done about it. *ITiCSE 97 Proceedings of the 2nd conference on Integrating technology into computer science education*, 29(3):114–116, 1997.

[19] James Newman. *Videogames*. Routledge, 2004.

[20] Lauren A. O'Connell and Hans A. Hoffmann. The vertebrate mesolimbic reward system and social behavior network: A comparative synthesis, 2011.

[21] Marc Prensky. *Digital Game-Based Learning*. McGraw-Hill, 2001.

[22] Andrew K Przybylski, C Scott Rigby, and Richard M Ryan. A motivational model of video game engagement. *Review of General Psychology*, 14(2):154–166, 2010.

[23] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 1996.

[24] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000.

[25] Alan Shalloway and James R. Trott. *Design Patterns Explained; A New Perspective on Object Oriented Design*. Addison-Wesley Professional, 2005.

[26] E. J. Simpson. *The Classification of Educational Objectives in the Psychomotor Domain*. Gryphon House, 1972.

[27] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley and Sons Ltd, 2008.

[28] Giovanni Viglietta. Gaming is a hard job, but someone has to do it!, march 2012.

Appendix A

Software design

In this appendix section we show the complete class diagram of the software we made, see Figure A.1. Relevant sequence diagrams are shown in Figure A.2 to A.6.

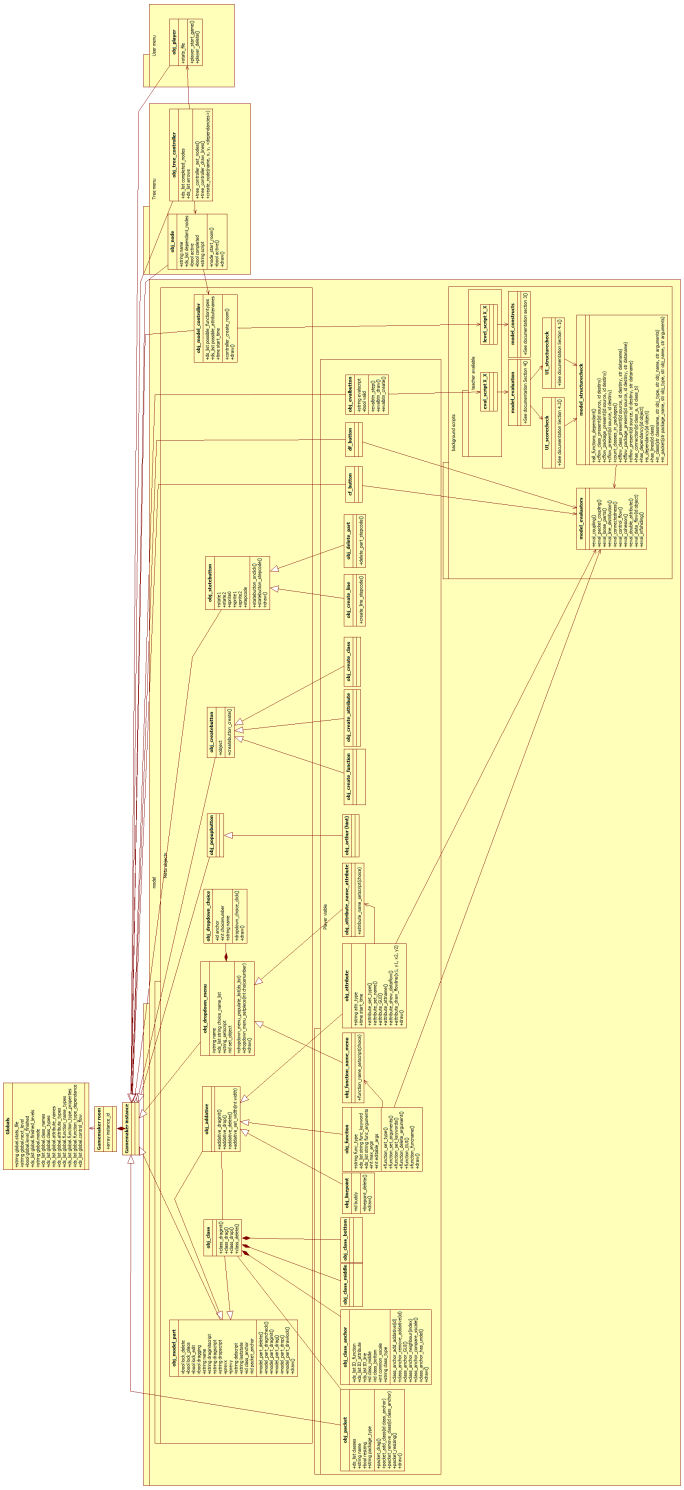

Figure A.1: Class diagram of the game's software architecture

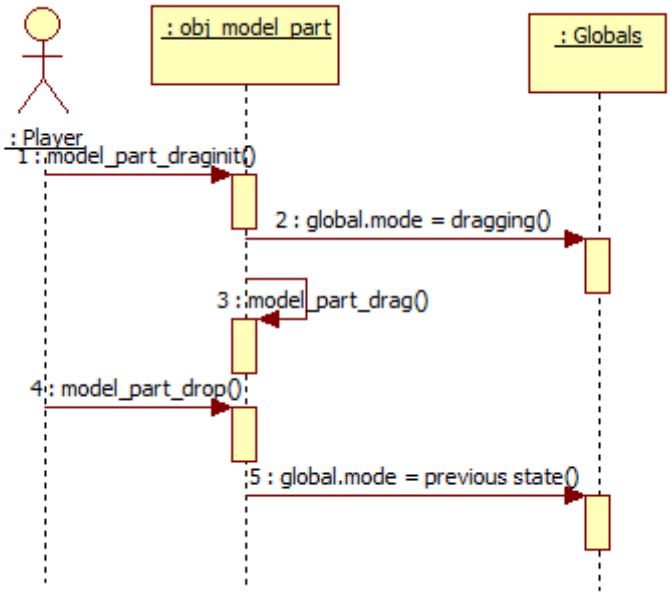

Figure A.2: Sequence diagram showing how an object is dragged.

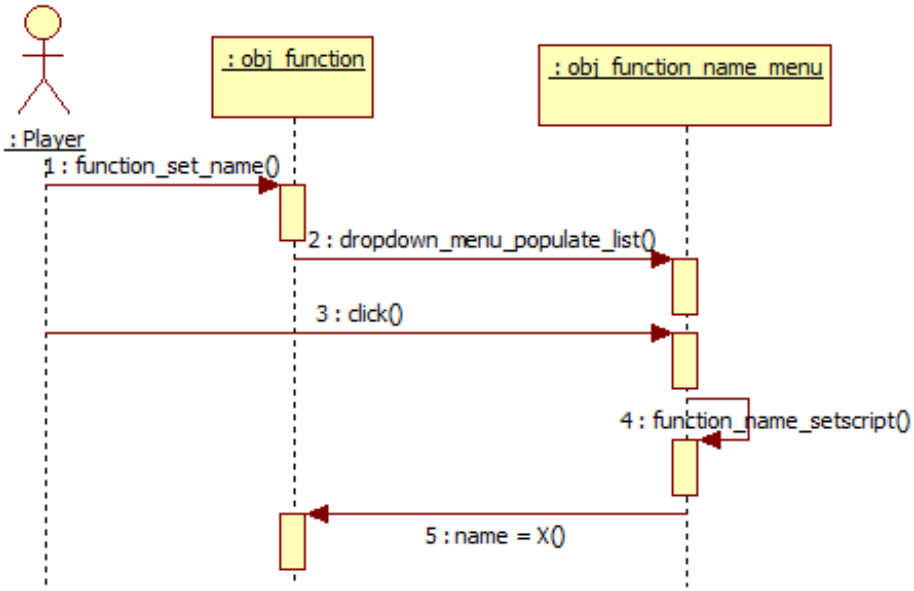

Figure A.3: Sequence diagram showing how the player can set a new name for an object.

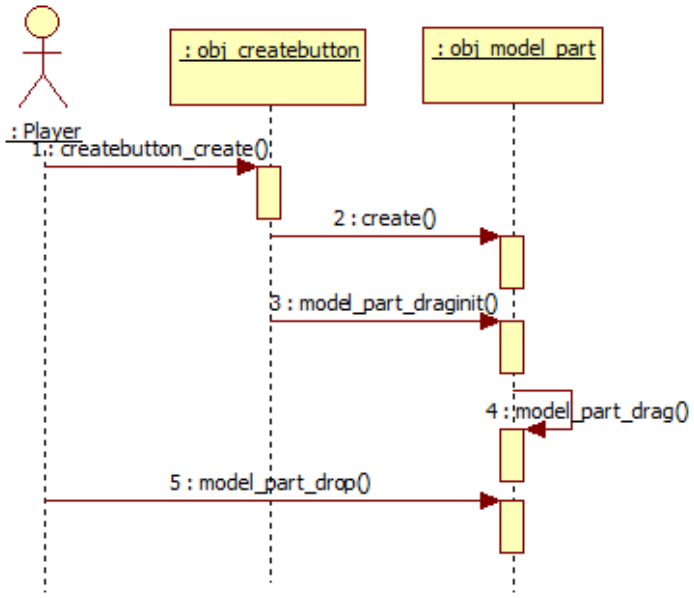

Figure A.4: Sequence diagram showing how the player can create a new part with a create button.

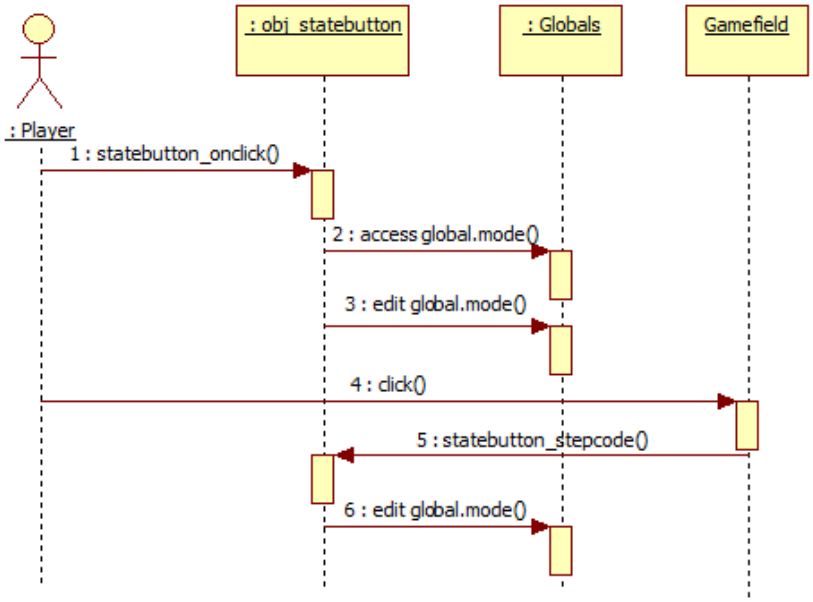

Figure A.5: Sequence diagram showing how the player can interact with the field using a state button.

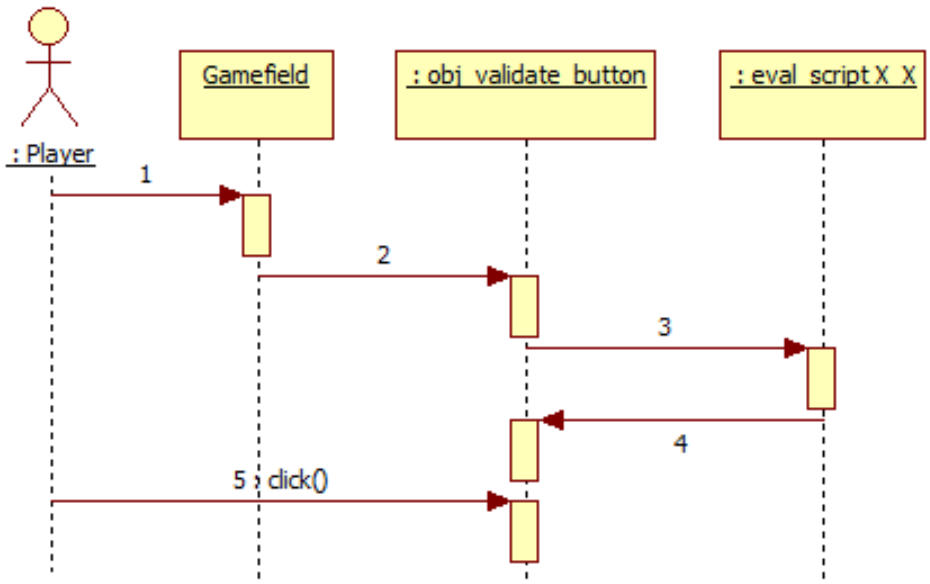

Figure A.6: Sequence diagram showing how the player can validate a solution.

Appendix B

Speak out loud condensed notes

B.1 Test 1: 21-jun-2012

Subject: 5th year computer sciences university student. Had already heard of the game, but was not involved with development. Playtime: 30min from start to finish Notabilities:

- Puzzles weren't worked out enough, a lot of the problems could be broken or worked around.
- Control flow was used as hint for the solution, this was intended.
- Subject tried to delete associations by clicking on a line, but this was only implemented on the start and end-point of an association's line, so this didn't work.
- Subject noted that it was disappointing that not every good action is met with a score.
- The package cohesion puzzle had a broken mechanic in the package cohesion scoring.

Conclusions: Nothing out of the ordinary. Subject had more trouble with bugs than with the puzzle itself. Quick fixes: Fixed the noted bugs in the puzzles plus the package cohesion mechanic.

B.2 Test 2: 22-jun-2012, 11:00

Subject: 2nd year medical HBO student. Knows nothing about software design or computer sciences. Never saw the game before. Claims not to be good with video games. Playtime: 1 hour and 30 minutes from start until 8 out of 14 puzzles were solved Notabilities:

- The subject did not know when a puzzle was solved and that she had to click the validate-model button, even when it was pointed out after 5 minutes of her searching.
- When one puzzle was too hard, the subject switched puzzles and tried again later. This was intended. (NB: Toleration, Variation, Deviation!)
- The instructions were too vague at times and the subject did not always understand
- The buddy hints were very useful and immediately found.
- The create buttons had conflicting icons. According to the subject, a pencil depicts drawing a line instead of picking up a part and putting it on the field.
- Drop down menu names were not always accurate
- Within half an hour the subject began to talk about methods, attributes and classes.
- Subject saw the data flow and control flow hints as a hint to drag one class to the other. This resulted in the subject dragging all classes to the same place, which was not our intention.
- The game has a bug in which more than one association can be mad between the same classes. Their lines are drawn on top of each other, so the subject did not know there were too many lines.
- The cohesion puzzle was the least well instructed. Drag the methods to their corresponding classes is an instruction that is easily understood, but does not explain cohesion. As a result the cohesion puzzle was finished the fastest, but understood the least.
- If a method or attribute is locked in a class and the subject, knowing about the lock, tries to move the class by dragging the method or attribute, nothing happens. This was counter-intuitive.

- Subject claims a lot of puzzles are solved by guessing.

Conclusions:

- The 'Toleration, Variation, Deviation' principle was an invaluable asset on letting the subject play by itself.
- Things were learned when both the instructions were elaborate and the puzzle was easily solvable. The instructions are linked with the action and solidified in the brain with a reward. If the instructions are not clear, the subject can not recall and if the action is not satisfying, the subject can not evaluate.

Quick fixes: Fixed more bugs and puzzle elements. Changed instructions at some puzzles. Added a sound and firework on the validate model button when a puzzle was right to attract attention.

B.3 Test3: 22-jun-2012 17:56

Subjects: Simultaneously 2 people. A 4th year law university student (depicted as LS) and a VWO working person (WP). Both have no experience in computer sciences, but are eager at playing video games. Both played on a different computer, thus had their own game. We finished in 47 minutes. LS was done, WM did not finish the last 4 puzzles. Notabilities:

- The mascot/buddy does not give hints at the save and tree screens. This was confusing.
- The validate button was immediately found now
- More bugs were found in puzzles
- Drawing lines (associations) gave problems again
- Subjects claimed to need more explanation. On the other hand: not all instructions were read.
- Game reminded subjects of 'Myst', because that game also gave little instructions and left the player to 'try everything'.
- Game sparked competitiveness between players.

- The art-style put the subjects off-trail, since the style was happy, but the game more difficult than they expected.
- Different colours in data flow were immediately understood by WP as 'different data'. This was as intended.
- LS realized after around 8 puzzles she was learning software design.
- LS claimed the points were scary (as in: judging), but also did not like it when a puzzle did not score her actions.
- LS tried solving puzzles on trying everything out. WM tried solving puzzles by understanding its mechanics.
- WM got stuck on coupling vs cohesion. When LS was finished with all her puzzles, she helped WM with the solution.

Conclusions:

- Competition worked well
- Too easy solutions and too long instructions lead to not reading and not evaluating. The instructions shouldn't be too long (around longer than 3 medium paragraphs) and the puzzles should not be too easy, which is hard to quantify.
- Implementing fireworks on completion worked better than expected.

Quick fix: None, implementing large fixes.

B.4 Test 4: 29-jun-2012 15:05

Subjects: A 3rd year Psychology student and a graduated Archaeologist. Notabilities:

- Unblocked difficult puzzles have to be blocked, because people are eager to try those first, even though they don't have the necessary knowledge
- The first puzzle about classes was used to explain all game mechanics. Instructions were too long and the evaluation too lenient.
- Subjects started making models without difficulty, even when they weren't instructed to before.

- Narrow solution puzzles are faster solved than wide solution puzzles
- The puzzle that explains cohesion only shows 1 on 1 cohesion, but not conflict between methods while trying to achieve cohesion. This gave trouble in the coupling vs cohesion puzzle.
- Still, people who understand more advance slower.

Conclusions:

- A lot of puzzles work counter-productive by learning in a wrong way. In the case of the cohesion puzzle we started with a puzzle that works in a perfect environment. Unfortunately, because we ignored the possible conflicts in the puzzle, people who understood the puzzle had trouble with the follow-up puzzle.

Quick fixes: Again none, we go make the big survey

Appendix C

Survey questions

These are the survey questions we asked our test subjects. The questions with multiple choice options have the statistics included.

C.1 Pre-game survey

- What is your age? (avg: 23)
- At what point are you in your career? (Pre-high school/ High school/ Student/ working) (1/ 15/ 35/ 16)
- Is your profession in Information Technology (IT)? (24/ 43)
- How high do you rate your skill with computers compared to average people? (1, I am really bad with computers $\rightarrow$ 10, I am very good with computers) (avg: 8.0)
- Have you played the game 'the Art of Software Design? (yes/no)(1/ 66)
- Please rate your interest in software design. (1, could not care less $\rightarrow$ 10, Awesome!)(avg: 6.6)
- Do you know what 'classes' are in software design? (yes/no)(33/ 31)
- Do you know what 'associations' are in software design? (yes/no)(13/ 54)
- Do you know what 'methods' are in software design? (yes/no)(27/ 37)
- Do you know what 'attributes' are in software design? (yes/no)(29/ 35)

- Do you know what 'coupling' are in software design? (yes/no)(10/ 57)
- Explain in your own words what 'coupling' means in software design.
- Do you know what 'cohesion' are in software design? (yes/no)(7/ 57)
- Explain in your own words what 'cohesion' means in software design.
- Do you know what 'control flow' are in software design? (yes/no)(12/ 52)
- Explain in your own words what 'control flow' means in software design.
- Do you know what 'data flow' are in software design? (yes/no)(11/ 53)
- Explain in your own words what 'data flow' means in software design.
- Do you know what 'agents' are in software design? (yes/no)(6/ 58)
- Explain in your own words what 'agents' means in software design.
- Do you know what 'interfaces' are in software design? (yes/no)(29/ 35)
- Explain in your own words what 'interfaces' means in software design.
- Do you know what 'information hiding' are in software design? (yes/no)(16/ 48)
- Explain in your own words what 'information hiding' means in software design.
- How often do you think you will use the above mentioned concepts if you were to be in software design? (Not/ Almost never/ Average/ Often/ Always)
  - Classes (4/ 0/ 11/ 23/ 26)
  - Associations (7/ 1/ 24/ 22/ 10))
  - Methods (5/ 1/ 16/ 19/ 23))
  - Attributes (5/ 1/ 16/ 19/ 23)
  - Coupling (5/ 3/ 31/ 15/ 10)
  - Cohesion (5/ 4/ 27/ 19/ 9)
  - Control flow (4/ 2/ 22/ 21/ 15)

  - Data flow (6/ 1/ 22/ 20/ 15)
  - Information hiding (5/ 5/ 29/ 18/ 7)
  - Interfaces (4/ 4/ 15/ 21/ 20)
  - Agents (6/ 5/ 31/ 15/ 7)

C.2 Post-game survey

- What is your age? (avg: 29)
- At what point are you in your career? (Pre-high school/ High school/ Student/ working) (0/ 2/ 4/ 7)
- How high do you rate your skill with computers compared to average people? (1, I am really bad with computers $\rightarrow$ 10, I am very good with computers) (avg: 7.6)
- Have you finished the game 'the Art of Software Design? (yes/no) (9/ 1)
- For approximately how many minutes have you played the game, finished or not? (avg: 34.8)
- Which puzzles have you finished? (Finished/ Only tried/ Never seen)
  - Classes (10/ 0/ 0)
  - Associations (9/ 1/ 0)
  - Methods (10/ 0/ 0)
  - Attributes (9/ 1/ 0)
  - Packages (9/ 1/ 0)
  - Coupling (9/ 1/ 0)
  - Cohesion (9/ 0/ 1)
  - Control flow (9/ 0/ 1)
  - Data flow (9/ 0/ 1)
  - Package cohesion (9/ 0/ 1)
  - Coupling vs Cohesion (9/ 0/ 1)
  - Interfaces (7/ 2/ 1)

  - Agents (8/ 1/ 1)
  - Information hiding (8/ 1/ 1)

- About your experience; Please select how much you agree with these statements (Completely disagree/ Slightly disagree/ Neutral/ Slightly agree/ Completely agree)
  - I liked the graphical design of the game. (4/ 4/ 2/ 1/ 2)
  - The puzzle objectives were clear. (3/ 4/ 1/ 3/ 2)
  - I had trouble assigning parts of the puzzle. (0/ 2/ 3/ 5/ 3)
  - I enjoyed playing the game. (4/ 3/ 4/ 0/ 2)
  - I feel like I learned something from playing this game. (2/ 5/ 3/ 1/ 2)
  - The game was hard to finish. (1/ 2/ 5/ 1/ 4)
  - The game taught me what software design is. (1/ 1/ 8/ 1/ 2)
  - While playing the game I felt like I was at school. (0/ 3/ 1/ 6/ 3)
  - The dotted lines in the game helped me understand data flow. (2/ 6/ 1/ 1/ 3)
  - The blue arrows in the game helped me understand control flow. (2/ 5/ 2/ 1/ 3)
  - I think I can make a software model because of this game. (0/ 4/ 6/ 1/ 2)
  - The indicated score number made me feel I did right. (4/ 2/ 2/ 4/ 1)
  - The light bulb made me feel I did right. (5/ 2/ 3/ 3/ 0)
  - The finishing sound made me feel I did right. (3/ 4/ 1/ 1/ 1)
  - I just dragged parts around to see if it was the right solution. (2/ 2/ 0/ 1/ 5)
  - I could finish puzzles easier if I understood the concept. (3/ 5/ 1/ 0/ 1)
  - Even if I did not understand the concept, I could finish the puzzles easily. (0/ 4/ 1/ 4/ 1)
  - After I finished a puzzle, I usually played around to find more solutions. (2/ 1/ 0/ 1/ 6)

  - The instructions were too long at times. (1/ 0/ 4/ 2/ 3)
  - This game is a perfect example of how school should be gamified. (1/ 3/ 2/ 1/ 3)
  - The puzzles often frustrated me. (1/ 1/ 2/ 3/ 3)
  - I am going to show this game to a friend. (2/ 1/ 3/ 1/ 3)
- Do you know what the following concepts mean in software design? (Yes/ No)
  - Classes (8/ 2)
  - Associations (8/ 2)
  - Methods (8/ 2)
  - Attributes (6/ 4)
  - Packages (7/ 3)
  - Coupling (5/ 5)
  - Cohesion (6/ 4)
  - Control flow (5/ 5)
  - Data flow (5/ 5)
  - Information hiding (6/ 4)
  - Interfaces (6/ 4)
  - Agents (6/ 4)
- Explain in your own words what 'coupling' means.
- Explain in your own words what 'cohesion' means.
- Explain in your own words what 'control flow' is.
- Explain in your own words what 'data flow' is.
- Explain in your own words what 'interfaces' do.
- Explain in your own words what 'agents' do.
- Rate the importance of all these concepts in software design. (Not important/ A little important/ Useful/ Important/ Very important)
  - Classes (9/ 1/ 3/ 0/ 0)

  - Associations (8/ 3/ 2/ 0/ 0)
  - Attributes (8/ 3/ 2/ 0/ 0)
  - Packages (3/ 5/ 5/ 0/ 0)
  - Methods (10/ 1/ 2/ 0/ 0)
  - Coupling (2/ 5/ 6/ 0/ 0)
  - Cohesion (2/ 6/ 5/ 0/ 0)
  - Control flow (6/ 4/ 2/ 1/ 0)
  - Data flow (6/ 5/ 2/ 0/ 0)
  - Information hiding (3/ 2/ 8/ 0/ 0)
  - Interfaces (3/ 5/ 2/ 0/ 0)
  - Agents (3/ 3/ 3/ 0/ 1)

- Can you explain why these software design concepts are used? What are their benefits?

- How often did you use the concepts that were taught to you? (Not/ Almost never/ Average/ Often/ Always)

  - Classes (5/ 1/ 1/ 1/ 2)
  - Attributes (5/ 1/ /1/ 0/ 3)
  - Methods (5/ 1/ 1/ 1/ 2)
  - Associations (4/ 2/ 1/ 1/ 2)
  - Coupling (0/ 1/ 4/ 2/ 3)
  - Cohesion (0/ 3/ 3/ 1/ 3)
  - Control flow (1/ 2/ 4/ 0/ 3)
  - Data flow (1/ 2/ 4/ 0/ 3)
  - Information hiding (1/ 1/ 2/ 2/ 4)
  - Interfaces (0/ 1/ 3/ 3/ 3)
  - Agents (1/ 1/ 4/ 1/ 3)
  - Packages (0/ 4/ 1/ 2/ 3)

- Which concepts did you find interesting? (Very boring/ Boring/ Neutral/ Interesting/ Very interesting)

  - Classes (4/ 3/ 1/ 0/ 2)
  - Attributes (5/ 2/ 1/ 0/ 2)
  - Methods (5/ 2/ 1/ 0/ 2)
  - Packages (3/ 2/ 3/ 0/ 2)
  - Associations (3/ 2/ 3/ 0/ 2)
  - Coupling (1/ 3/ 3/ 1/ 2)
  - Cohesion (2/ 2/ 3/ 1/ 2)
  - Control flow (2/ 3/ 1/ 2/ 2)
  - Data flow (2/ 3/ 1/ 2/ 2)
  - Information hiding (2/ 2/ 4/ 0/ 2)
  - Interfaces (1/ 3/ 4/ 0/ 2)
  - Agents (3/ 0/ 5/ 0/ 2)
- Please rate your current overall interest in software design (1,boring $\rightarrow$ 10, Awesome) (avg: 7.4)
- Did you enjoy the game? (Yes/ No) (9/ 4)
- If you would give this game a review, what score would you give it? (1, Bad $\rightarrow$ 10, good) (avg: 5.8)
- Any suggestions on improvement?